Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation

Renton, WA, USA *April 9–11, 2018*

ISBN 978-1-939133-01-4

conference proceedings

15th USENIX Symposium on Networked Systems Design and Implementation

Renton, WA, USA
April 9–11, 2018

Sponsored by

In cooperation with SIGCOMM and SIGOPS

NSDI ’18 Sponsors

Gold Sponsors

facebook

Silver Sponsors

Google

Bronze Sponsors

General Sponsor

Industry Partners and Media Sponsor

ACM *Queue*

Distributed Management Task Force (DMTF)

No Starch Press

© 2018 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author’s employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person’s exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-939133-01-4

USENIX Supporters

USENIX Patrons

Facebook Google Microsoft
NetApp Private Internet Access

USENIX Benefactors

Amazon Oracle Squarespace VMware

USENIX Partners

Booking.com Can Stock Photo Cisco Meraki
Dealslands Fotosearch

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation

April 9–11, 2018
Renton, WA, USA

Symposium Organizers

Program Co-Chairs
Sujata Banerjee, *VMware Research*
Srinivasan Seshan, *Carnegie Mellon University*

Program Committee
Rachit Agarwal, *Cornell University*
Mohamed Alizadeh, *MIT*
Katerina Argyraki, *EPFL*
Ranjita Bhagwan, *Microsoft Research*
Marco Canini, *KAUST*
Kai Chen, *HKUST*
Romit Roy Choudhury, *University of Illinois at Urbana-Champaign*
Mike Freedman, *Princeton University*
Roxana Geambasu, *Columbia University*
Shyam Gollakota, *University of Washington*
Ramesh Govindan, *USC*
Dongsu Han, *KAIST*
Jon Howell, *Google*
Kyle Jamieson, *Princeton University*
Michael Kaminsky, *Intel Labs*
Srikanth Kandula, *Microsoft*
Srinivas Keshav, *University of Waterloo*
Kyu-Han Kim, *Hewlett Packard Labs*
Eddie Kohler, *Harvard University*
Arvind Krishnamurthy, *University of Washington*
Wyatt Lloyd, *Princeton University*
Boon Thau Loo, *University of Pennsylvania*
Jay Lorch, *Microsoft*
Harsha V. Madhyastha, *University of Michigan*
Dahlia Malkhi, *VMware Research*
KyongSoo Park, *KAIST*
Amar Phanishayee, *Microsoft*
Raluca Ada Popa, *University of California, Berkeley*
George Porter, *University of California, San Diego*
Lili Qiu, *UT Austin*
Costin Raiciu, *University Politehnica of Bucharest*
Timothy Roscoe, *ETH Zurich*
Michael Schapira, *Hebrew University*
Cole Schlesinger, *Barefoot Networks*
Vyas Sekar, *Carnegie Mellon University*
Emin Gun Sirer, *Cornell University*
Alex C. Snoeren, *University of California, San Diego*
Hakim Weatherspoon, *Cornell University*
Keith Winstein, *Stanford University*
Tim Wood, *George Washington University*
Minlan Yu, *Yale University*
Matei Zaharia, *Stanford University*
Lin Zhong, *Rice University*

Poster Session Co-Chairs
Rachit Agarwal, *Cornell University*
Dongsu Han, *KAIST*

Test of Time Awards Committee
Aditya Akella, *University of Wisconsin–Madison*
Jon Crowcroft, *University of Cambridge*
Mike Dahlin, *Google*
Nick Feamster, *Princeton University*

Steering Committee
Aditya Akella, *University of Wisconsin–Madison*
Katerina Argyraki, *EPFL*
Paul Barham, *Google*
Nick Feamster, *Princeton University*
Casey Henderson, *USENIX Association*
Jon Howell, *Google*
Arvind Krishnamurthy, *University of Washington*
Jeff Mogul, *Google*
Brian Noble, *University of Michigan*
Timothy Roscoe, *ETH Zurich*
Alex C. Snoeren, *University of California, San Diego*

External Reviewers

Aditya Akella
Gustavo Alonso
David Andersen
Les Atlas
Suman Banerjee
Daniel S. Berger
Sol Boucher
Prabal Dutta
Giulia Fanti
Ian Goldberg
Chris Hodsdon
Anuj Kalia
Charles Killian
Dejan Kostić
Hyeontaek Lim
Haonan Lu
Bruce Maggs
David Naylor
Ravi Netravali
Khiem Ngo
Matthew Mukerjee
Jitendra Padhye
Theano Stavrinos
Ion Stoica
Ki Suh Lee
Christina Vlachou

Message from the NSDI ’18 Program Co-Chairs

Welcome to NSDI ’18!

Over the years, NSDI has established itself as the top venue for work on networked and distributed systems. This year’s iteration is no exception, and we have an excellent program that showcases exciting research advances and operational experience on topics including distributed systems, cloud, NFV, web systems, video systems, congestion control, traffic management, fault tolerance, configuration management, and diagnosis.

NSDI ’18 received 255 submissions of which we accepted 40 papers. Our Program Committee consisted of 45 members with a mix of research and industry experience. This year’s review process was double-blind as it was with NSDI ‘17. The papers were reviewed in two rounds, with papers that advanced to the second round receiving at least five reviews. In total, the Program Committee and external reviewers generated 1,003 reviews. Once we completed reviewing, the committee discussed online and selected 77 papers that were discussed further at a 1.5-day PC meeting held in Palo Alto, CA. The program committee strived to produce valuable feedback; we hope it benefited authors of every submission.

It has been a great pleasure working with many other people to put this program together. We would like to thank the authors of all submitted papers for choosing to send work of such high caliber to NSDI. Thanks also to the program committee for their professionalism, diligence and enthusiasm. Special thanks to Dongsu Han and Rachit Agarwal for serving as poster chairs and to Lin Zhong, George Porter, Dahlia Malkhi, Dongsu Han, Lili Qiu, Srinivasan Keshav and Tim Wood for their help in selecting the Best Paper and Community Awards. Thanks also to the members of the Test of Time Awards Committee, Aditya Akella, Jon Crowcroft, Mike Dahlin, and Nick Feamster. We are also very grateful to the USENIX staff, including Casey, Hilary and Michele, for their exceptional support and help. We are grateful for the help from Keith Winstein and Mary Jane Swenson in organizing the PC meeting at Stanford University.

Finally, NSDI wouldn’t be what it is without the attendees, so thank you very much for being here. We hope you enjoy the conference!

Srinivasan Seshan, *Carnegie Mellon University*
Sujata Banerjee, *VMWare Research*
NSDI ’18 Program Co-Chairs

NSDI '18: 15th USENIX Symposium on Networked Systems Design and Implementation

April 9–11, 2018
Renton, WA, USA

New Hardware

Approximating Fair Queueing on Reconfigurable Switches 1
Naveen Kr. Sharma and Ming Liu, *University of Washington;* Kishore Atreya, *Cavium;* Arvind Krishnamurthy, *University of Washington*

PASTE: A Network Programming Interface for Non-Volatile Main Memory 17
Michio Honda, *NEC Laboratories Europe;* Giuseppe Lettieri, *Università di Pisa;* Lars Eggert and Douglas Santry, *NetApp*

NetChain: Scale-Free Sub-RTT Coordination 35
Xin Jin, *Johns Hopkins University;* Xiaozhou Li, *Barefoot Networks;* Haoyu Zhang, *Princeton University;* Nate Foster, *Cornell University;* Jeongkeun Lee, *Barefoot Networks;* Robert Soulé, *Università della Svizzera italiana;* Changhoon Kim, *Barefoot Networks;* Ion Stoica, *UC Berkeley*

Azure Accelerated Networking: SmartNICs in the Public Cloud 51
Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg, *Microsoft*

Distributed Systems

zkLedger: Privacy-Preserving Auditing for Distributed Ledgers 65
Neha Narula, *MIT Media Lab;* Willy Vasquez, *University of Texas at Austin;* Madars Virza, *MIT Media Lab*

Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization 81
Yilong Geng, Shiyu Liu, and Zi Yin, *Stanford University;* Ashish Naik, *Google Inc.;* Balaji Prabhakar and Mendel Rosenblum, *Stanford University;* Amin Vahdat, *Google Inc.*

SnailTrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows 95
Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe, *ETH Zurich*

Traffic Management

Balancing on the Edge: Transport Affinity without Network State 111
João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa, *Fastly*

Stateless Datacenter Load-balancing with Beamer 125
Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu, *University Politehnica of Bucharest*

Larry: Practical Network Reconfigurability in the Data Center 141
Andromachi Chatzieleftheriou, Sergey Legtchenko, Hugh Williams, and Antony Rowstron, *Microsoft Research*

Semi-Oblivious Traffic Engineering: The Road Not Taken 157
Praveen Kumar and Yang Yuan, *Cornell;* Chris Yu, *CMU;* Nate Foster and Robert Kleinberg, *Cornell;* Petr Lapukhov and Chiun Lin Lim, *Facebook;* Robert Soulé, *Università della Svizzera italiana*

NFV and Hardware

Metron: NFV Service Chains at the True Speed of the Underlying Hardware 171
Georgios P. Katsikas, *RISE SICS and KTH Royal Institute of Technology;* Tom Barbette, *University of Liege;* Dejan Kostic, *KTH Royal Institute of Technology;* Rebecca Steinert, *RISE SICS;* Gerald Q. Maguire Jr., *KTH Royal Institute of Technology*

G-NET: Effective GPU Sharing in NFV Systems 187
Kai Zhang, *Fudan University;* Bingsheng He, *National University of Singapore;* Jiayu Hu, *University of Science and Technology of China;* Zeke Wang, *National University of Singapore;* Bei Hua, Jiayi Meng, and Lishan Yang, *University of Science and Technology of China*

SafeBricks: Shielding Network Functions in the Cloud 201
Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy, *UC Berkeley*

Web and Video

Vesper: Measuring Time-to-Interactivity for Web Pages 217
Ravi Netravali and Vikram Nathan, *MIT CSAIL;* James Mickens, *Harvard University;* Hari Balakrishnan, *MIT CSAIL*

Towards Battery-Free HD Video Streaming 233
Saman Naderiparizi, Mehrdad Hessar, Vamsi Talla, Shyamnath Gollakota, and Joshua R Smith, *University of Washington*

Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs 249
Ravi Netravali, *MIT CSAIL;* James Mickens, *Harvard University*

Salsify: Low-Latency Network Video through Tighter Integration between a Video Codec and a Transport Protocol 267
Sadjad Fouladi, John Emmons, and Emre Orbay, *Stanford University;* Catherine Wu, *Saratoga High School;* Riad S. Wahby and Keith Winstein, *Stanford University*

Performance Isolation and Scaling

ResQ: Enabling SLOs in Network Function Virtualization 283
Amin Tootoonchian, *Intel Labs;* Aurojit Panda, *NYU, ICSI;* Chang Lan, *UC Berkeley;* Melvin Walls, *Nefeli;* Katerina Argyraki, *EPFL;* Sylvia Ratnasamy, *UC Berkeley;* Scott Shenker, *UC Berkeley, ICSI*

Elastic Scaling of Stateful Network Functions 299
Shinae Woo, *KAIST, UC Berkeley;* Justine Sherry, *CMU;* Sangjin Han, *UC Berkeley;* Sue Moon, *KAIST;* Sylvia Ratnasamy, *UC Berkeley;* Scott Shenker, *UC Berkeley, ICSI*

Iron: Isolating Network-based CPU in Container Environments 313
Junaid Khalid, *UW-Madison;* Eric Rozner, Wesley Felter, Cong Xu, and Karthick Rajamani, *IBM Research;* Alexandre Ferreira, *Arm Research;* Aditya Akella, *UW-Madison*

Congestion Control

Copa: Practical Delay-Based Congestion Control for the Internet 329
Venkat Arun and Hari Balakrishnan, *MIT CSAIL*

PCC Vivace: Online-Learning Congestion Control 343
Mo Dong and Tong Meng, *UIUC;* Doron Zarchy, *The Hebrew University of Jerusalem;* Engin Arslan, *UIUC;* Yossi Gilad, *MIT;* Brighten Godfrey, *UIUC;* Michael Schapira, *The Hebrew University of Jerusalem*

Multi-Path Transport for RDMA in Datacenters 357
Yuanwei Lu, *Microsoft Research and University of Science and Technology of China;* Guo Chen, *Hunan University;* Bojie Li, *Microsoft Research and University of Science and Technology of China;* Kun Tan, *Huawei Technologies;* Yongqiang Xiong, Peng Cheng, and Jiansong Zhang, *Microsoft Research;* Enhong Chen, *University of Science and Technology of China;* Thomas Moscibroda, *Microsoft Azure*

(continued on next page)

Cloud

Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization 373
Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat, *Google, Inc.*

LHD: Improving Cache Hit Rate by Maximizing Hit Density 389
Nathan Beckmann, *Carnegie Mellon University;* Haoxian Chen, *University of Pennsylvania;* Asaf Cidon, *Stanford University/Barracuda Networks*

Performance Analysis of Cloud Applications 405
Dan Ardelean, Amer Diwan, and Chandra Erdman, *Google*

Diagnosis

007: Democratically Finding the Cause of Packet Drops 419
Behnaz Arzani, *Microsoft Research;* Selim Ciraci, *Microsoft;* Luiz Chamon, *University of Pennsylvania;* Yibo Zhu and Hongqiang (Harry) Liu, *Microsoft Research;* Jitu Padhye, *Microsoft;* Boon Thau Loo, *University of Pennsylvania;* Geoff Outhred, *Microsoft*

Efficient and Correct Test Scheduling for Ensembles of Network Policies 437
Yifei Yuan, Sanjay Chandrasekaran, Limin Jia, and Vyas Sekar, *Carnegie Mellon University*

Distributed Network Monitoring and Debugging with SwitchPointer 453
Praveen Tammana, *University of Edinburgh;* Rachit Agarwal, *Cornell University;* Myungjin Lee, *University of Edinburgh*

Stroboscope: Declarative Network Monitoring on a Budget 467
Olivier Tilmans, *Université Catholique de Louvain;* Tobias Bühler, *ETH Zürich;* Ingmar Poese, *BENOCS;* Stefano Vissicchio, *University College London;* Laurent Vanbever, *ETH Zürich*

Fault-Tolerance

Plover: Fast, Multi-core Scalable Virtual Machine Fault-tolerance 483
Cheng Wang, Xusheng Chen, Weiwei Jia, Boxuan Li, Haoran Qiu, Shixiong Zhao, and Heming Cui, *The University of Hong Kong*

Odin: Microsoft's Scalable Fault-Tolerant CDN Measurement System 501
Matt Calder, *Microsoft/USC;* Manuel Schröder, Ryan Gao, Ryan Stewart, and Jitendra Padhye, *Microsoft;* Ratul Mahajan, *Intentionet;* Ganesh Ananthanarayanan, *Microsoft;* Ethan Katz-Bassett, *Columbia University*

Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure 519
Qiao Zhang, *University of Washington;* Guo Yu, *Cornell University;* Chuanxiong Guo, *Toutiao* (*Bytedance*)*;* Yingnong Dang, Nick Swanson, Xinsheng Yang, Randolph Yao, and Murali Chintalapati, *Microsoft;* Arvind Krishnamurthy and Thomas Anderson, *University of Washington*

Physical Layer

LiveTag: Sensing Human-Object Interaction through Passive Chipless WiFi Tags 533
Chuhan Gao and Yilong Li, *University of Wisconsin-Madison;* Xinyu Zhang, *University of California San Diego*

Inaudible Voice Commands: The Long-Range Attack and Defense 547
Nirupam Roy, Sheng Shen, Haitham Hassanieh, and Romit Roy Choudhury, *University of Illinois at Urbana-Champaign*

PowerMan: An Out-of-Band Management Network for Datacenters Using Power Line Communication 561
Li Chen, Jiacheng Xia, Bairen Yi, and Kai Chen, *The Hong Kong University of Science and Technology*

Configuration Management

NetComplete: **Practical Network-Wide Configuration Synthesis with Autocompletion** 579
Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev, *ETH Zürich*

Automatically Correcting Networks with ***NEAt*** 595
Wenxuan Zhou, Jason Croft, Bingzhe Liu, Elaine Ang, and Matthew Caesar, *University of Illinois at Urbana-Champaign*

Net2Text: **Query-Guided Summarization of Network Forwarding Behaviors** 609
Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev, *ETH Zürich*

Approximating Fair Queueing on Reconfigurable Switches

Naveen Kr. Sharma[*] *Ming Liu*[*] *Kishore Atreya*[†] *Arvind Krishnamurthy*[*]

Abstract

Congestion control today is predominantly achieved via end-to-end mechanisms with little support from the network. As a result, end-hosts must cooperate to achieve optimal throughput and fairness, leading to inefficiencies and poor performance isolation. While router mechanisms such as Fair Queuing guarantee fair bandwidth allocation to all participants and have proven to be optimal in some respects, they require complex flow classification, buffer allocation, and scheduling on a per-packet basis. These factors make them expensive to implement in high-speed switches.

In this paper, we use emerging reconfigurable switches to develop an approximate form of Fair Queueing that operates at line-rate. We leverage configurable per-packet processing and the ability to maintain mutable state inside switches to achieve fair bandwidth allocation across all traversing flows. Further, present our design for a new dequeuing scheduler, called Rotating Strict Priority scheduler that lets us transmit packets from multiple queues in approximate sorted order. Our hardware emulation and software simulations on a large leaf-spine topology show that our scheme closely approximates ideal Fair Queueing, improving the average flow completion times for short flows by 2-4x and 99th tail latency by 4-8x relative to TCP and DCTCP.

1 Introduction

Most current congestion control schemes rely on end-to-end mechanisms with little support from the network (e.g., ECN, RED). While this approach simplifies switches and lets them operate at very high speeds, it requires end-hosts to cooperate to achieve fair network sharing, thereby leading to inefficiencies and poor performance isolation. On the other hand, if the switches were capable of maintaining per-flow state, extracting rich telemetry from the network, and performing configurable per-packet processing, one can realize intelligent congestion control mechanisms that take advantage of dynamic network state directly inside the network and improve network performance.

One such mechanism is Fair Queueing, which has been studied extensively and shown to be optimal in several aspects. It provides the illusion that every flow (or participant) has its own queue and receives a fair share of the bandwidth under all circumstances, regardless of other network traffic. Having the network enforce fair bandwidth allocation offers several benefits. It simplifies congestion control at the end-hosts, removing the need to perform slow-start or complex congestion avoidance strategies. Further, flows can ramp up quickly without affecting other network traffic. It also provides strong isolation among competing flows, protects well-behaved flows from ill-behaving traffic, and enables bounded delay guarantees [34].

A fair bandwidth allocation scheme is potentially well suited to today's datacenter environment, where multiple applications with diverse network demands often co-exist. Some applications require low latency, while others need sustained throughput. Datacenter networks must also contend with challenging traffic patterns – such as large incasts or fan-in, micro-bursts, and synchronized flows, – which can all be managed effectively using a fair queueing mechanism. Fair queueing mechanisms can also provide bandwidth guarantees for multiple tenants of a shared cloud infrastructure [35].

Over the years, several algorithms for enforcing fair bandwidth allocation have been proposed [25, 27, 28, 33], but rarely deployed in practice, primarily due to their inherent complexities. These algorithms maintain state and perform operations on a per-flow basis, making them challenging to implement at data rates of 3-6 Tbps in hardware. However, recent advances in switching hardware allow flexible per-packet processing and the ability to maintain limited mutable state at switches without sacrificing performance [12, 6]. In this paper, we explore whether an efficient fair queueing implementation can be realized using these emerging reconfigurable switches.

We present *Approximate Fair Queueing (AFQ)*, a fair bandwidth allocation mechanism that approximates the various components of an ideal fair queueing scheme using features available in emerging programmable switches, such as the ability to maintain and mutate switch state on a per-packet basis, perform limited computation for each packet, and dynamically determine which egress queue to use for a given packet. We describe a variant of the packet-pair flow control protocol [24], designed to work with AFQ, that achieves close to optimal performance while maintaining short queues. We further prototype an AFQ implementation on a Cavium networking processor and study its feasibility on upcoming reconfigurable switches. Using a real hardware testbed and large-scale simulations, we demon-

[*]University of Washington
[†]Cavium Inc.

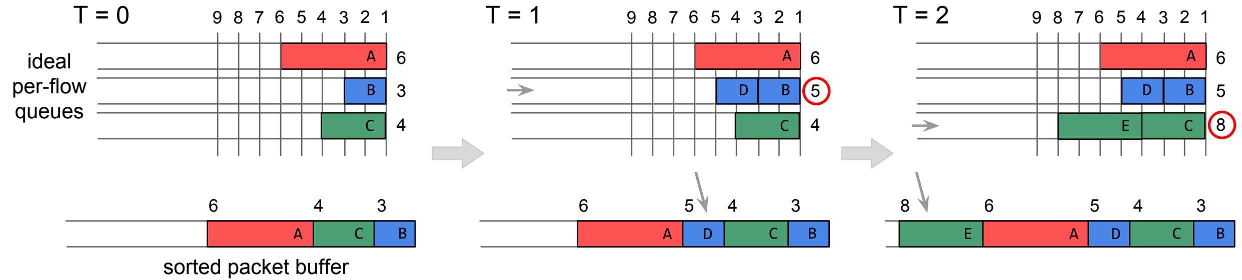


Figure 1: An example of the bit-by-bit round robin Fair Queueing algorithm. The algorithm buffers all packets in sorted order based on their departure round. When a blue packet D of size 2 arrives at T = 1, its departure round is calculated as 5 and is placed between packets A and C in the sorted buffer. Similarly, when a green packet of size 4 arrives at T = 2, its departure round is 8, and it is placed at the end of the departure queue.

strate AFQ's utility, showing it achieves fair bandwidth allocation for common datacenter workloads and traffic patterns, significantly improving performance over existing schemes. Specifically, AFQ reduces the average flow completion time of common workloads by 2-4x compared to TCP and DCTCP, and 99th percentile tail latency for short flows by up to 5-10x. We measure its overhead programmable switches by implementing AFQ in the P4 language and compiling it to a realistic hardware model, demonstrating that the resource overhead is modest.

2 Background

The idea of enforcing fair bandwidth allocation inside the network has been well studied and shown to offer several desirable properties. A straight-forward way of achieving such allocation is to have per-flow queues, as proposed by Nagle [31], serviced in a round robin manner. This is clearly impractical given today's network speeds and workload complexities. An efficient algorithm, called *bit-by-bit round robin (BR)*, proposed in [18], achieves ideal fair queueing behavior without requiring expensive per-flow queues. We describe this approach next since it forms the basis of our AFQ mechanism. We then provide background on the reconfigurable switch architecture.

2.1 Bit-by-Bit Round Robin (BR)

The *bit-by-bit round robin* algorithm achieves per-flow fair queueing using a round robin scheme wherein each active flow transmits a single bit of data every round. Then, the round ends, and the round number is incremented by one. Since it is impractical to build such a system, the BR algorithm "simulates" this scheme at packet granularity using the following steps.

- For every packet, the switch computes a *bid number* that estimates the time (round) when the packet would have departed.
- All packets are then buffered in a *sorted priority queue* based on their bid numbers, which allows dequeuing and transmission of the packet with the lowest bid number at any time.

Figure 1 shows a simple example of this approach. Although the BR algorithm achieves ideal fair queuing behavior, several factors make it challenging to implement given today's line-rate, 3-6 Tbps switches. First, to compute bid numbers for each packet, the switch must maintain the *finish round number* for each active flow. This is equal to the round when the flow's last byte will be transmitted and must be updated after each packet's arrival. Today's switches carry hundreds to thousands of concurrent flows [7, 37]. Their limited amounts of stateful memory makes it difficult to store and update per-flow bid numbers. Second, inserting packets into an ordered queue is an expensive $O(logN)$ operation, where N is the maximum buffer size in number of packets. Given the 12-20MB packet buffers available in today's switches, this operation is challenging to implement at a line-rate of billions of packets per second. Finally, switches need to store and update the current round number periodically using non-trivial computation involving: (1) time elapsed since last update, (2) number of active flows, and (3) link speed, as described in [25]. Today's line-rate switches lack the capability to perform such complex computations on a per-packet basis.

As noted, emerging reconfigurable switches allow flexible packet processing and the ability to maintain limited switch state, therefore we explore whether we can implement fair-queuing on these new line-rate switches. Further, recent work [38] has shown approximation to be a useful tool for implementing a broad class of in-network protocols for congestion control, load balancing, QoS and fairness.

2.2 Reconfigurable Switches

Reconfigurable switches provide a *match+action (M+A)* processing model: *match* on arbitrary packet header fields and then perform simple packet processing *actions*. In our work, we assume an abstract Reconfigurable Match Table (RMT) switch model, as described in [8, 9] and depicted in Figure 2. A reconfigurable switch begins packet processing by extracting relevant packet headers via a user-defined parse graph. The extracted header fields and packet metadata are passed onto a pipeline of

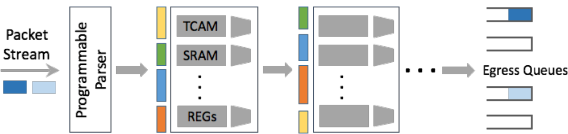


Figure 2: The architecture of a reconfigurable switch. Packets are processed by a pipeline of match+action stages with local state.

user-defined M+A tables. Each table matches on a subset of extracted headers and can apply simple processing primitives to any field. After traversing the pipeline stages, packets are deposited in one of multiple queues associated with the egress port for future transmission. The parser and the M+A pipeline can be configured using a high-level language, such as P4 [8] or PoF [43].

A reconfigurable switch provides several hardware features to support packet processing on the data path: (1) a limited amount of *stateful memory*, such as counters, meters, and registers, which can be accessed and updated to maintain state across packets, and (2) *computation primitives*, such as addition, bit-shifts, hashing, and max/min, which can perform a limited amount of processing on header fields and data retrieved from stateful memory. Further, switch metadata, such as queue lengths, congestion status, and bytes transmitted, can also be used in packet processing. Crucially, the pipeline stages can determine which transmit queue to use for a given packet based on packet header content and local state. Finally, a switch-local *control plane CPU* can also perform periodic bookkeeping tasks. Several such reconfigurable switches, – Cavium XPliant [12], Barefoot Tofino [6] and Intel Flexpipe [32] – are available today.

3 Approximate Fair Queueing

Any fair queuing router must perform per-flow management tasks to guarantee fair bandwidth allocation. These tasks include *packet classification* – which flow this packet belongs to, *buffer allocation* – whether this flow's packet should be enqueued or dropped, and *packet scheduling* – decide which flow's packet to transmit next. The key idea behind AFQ is to approximate the various components of a fair queueing scheme using features available in programmable switches.

Our design goals for AFQ include achieving per-flow max-min fairness [20], where a flow is defined as a unique 5-tuple. Our design should be implementable in high-speed routers running at line-rate. It must also be able to handle several thousand flows with varying packet sizes. We next provide an overview of our design.

3.1 Design Overview

Our design emulates the ideal BR algorithm described earlier. Like that algorithm, AFQ proceeds in a round robin manner, where every flow transmits a fixed number of bytes in each round. On arrival, each packet is assigned a departure round number based on how many bytes the flow has sent in the past, and packets are scheduled to be transmitted in increasing round numbers. Implementing this scheme requires AFQ to store the finish round number for every active flow at the switch and schedule buffered packets in a sorted order. It must also store and update the current round number periodically at the switch.

We approximate fair queueing using three key ideas. First, we store approximate flow bid numbers in sub-linear space using a variant of the count-min sketch, letting AFQ maintain state for a large number of flows with limited switch memory. This is made feasible by the availability of read-write registers on the datapath of reconfigurable switches. Second, AFQ uses coarser grain rounds that are incremented only after all active flows have transmitted a configurable number of bytes through an output port. Third, AFQ schedules packets to depart in an approximately sorted manner using multiple FIFO queues available at each port on these reconfigurable switches. Combining these techniques yields schedules that approximate those produced by a fair queueing switch. However, we show that AFQ provides performance that is comparable to fair queueing for today's datacenter workloads despite these approximations. Figure 3 shows the pseudocode describing AFQ's main components, which we explain in more detail in the next three sections.

3.2 Storing Approximate Bid Numbers

A flow's bid number in the BR algorithm is its finish-round number, which estimates when the flow's last enqueued byte will depart from the switch. The bid number of a flow's packet is a function of both the current active round number as well as the bid number associated with the flow's previous packet, and it is used to determine the packet's transmission order. AFQ stores each active flow's bid number in a count-min sketch-like datastructure to reduce the stateful memory footprint on the switch since such memory is a limited resource.

A count-min sketch is simply a 2D array of counters that supports two operations: (a) `inc(e,n)`, which increments the counter for element `e` by `n`, and (b) `read(e)`, which returns the counter for element `e`. For a sketch with `r` rows and `c` columns, `inc(e,n)` applies `r` independent hash functions to `e` to locate a cell in each row and increments the cell by `n`. The operation `read(e)` applies the same `r` hash functions to locate the same `r` cells and returns the minimum among them. The approximate counter value always exceeds or equals the exact value, letting us store flow bid numbers efficiently in sub-linear space. Theoretically, to get an ϵ approximation, – i.e., $error < \epsilon \times K$ with probability $1 - \delta$, where K is the number of increments to the sketch, – we need `c` = e/ϵ and `r` = $log(1/\delta)$ [17].

```
/* AFQ parameters */

S[][] : sketch for bid numbers
nH    : # of hashes in sketch
nB    : # of buckets in sketch
nQ    : # of FIFO queues
BpR   : bytes sent in each round

/* Count-min sketch functions */
func read_sketch(pkt):
   val = INT_MAX
   for i = 1 to nH:
      h = hash_i(pkt) % nB
      val = min(S[i][h], val)
   return val

func update_sketch(pkt, val):
   for i = 1 to nH:
      h = hash_i(pkt) % nB
      S[i][h] = max(S[i][h], val)
```

```
/* Enqueue Module */

R : Current round (shared w/ dequeue)

On packet arrival:
   bid = read_sketch(pkt)

   // If flow hasn't sent in a while,
   // bump it's round to current round.
   bid = max(bid, R * BpR)

   bid = bid + pkt.size
   pkt_round = bid / BpR

   // If round too far ahead, drop pkt.
   if (pkt_round - R) > nQ:
      drop(pkt)
   else:
      enqueue(pkt_round % nQ, pkt)
      update_sketch(pkt, bid)
```

```
/* Dequeue Module */

R : Current round number (shared)
i : Current queue being serviced

while True:
   // If no packets to send, spin.
   if buffer.empty()
      continue;

   // Drain i'th queue till empty.
   while !queue[i].empty():
      pkt = dequeue(i)
      send(pkt)

   // Move onto next queue,
   // increment round number.
   i = (i + 1) % nQ
   R = R + 1
```

Figure 3: Pseudocode for AFQ

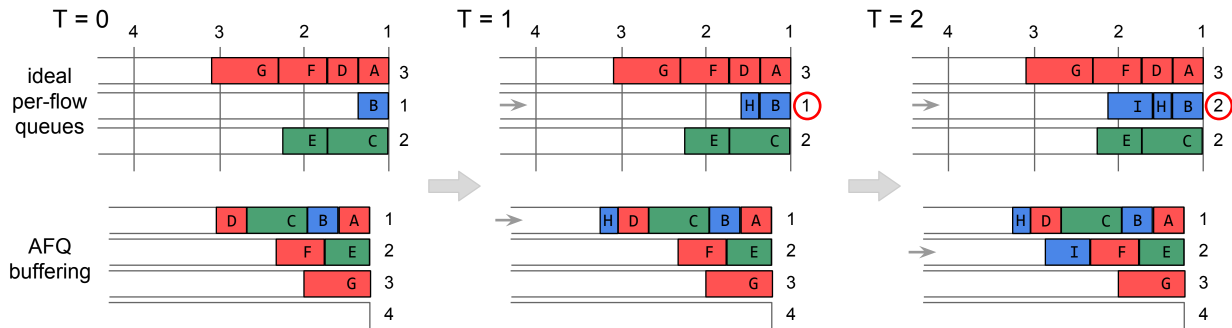


Figure 4: An example of the AFQ enqueue mechanism. As packets arrive, their bid numbers are estimated, and they are placed in an available FIFO queues. When a blue packet H arrives at T = 1, its bid number falls within round 1 and is placed in the first FIFO queue servicing round 1. When a subsequent blue packet I arrives at T = 2, its bid number falls in round 2; hence, it is placed is the second FIFO queue. For both packets H and I, we can see the approximation effects of using a large quantum of bytes per round and FIFO queues. An ideal FQ scheme using BR would transmit packet H before packets C and D, and packet I before E and F, as their last bytes are enqueued before the other packets in the per-flow queue. However, this reordering is upper-bounded by the number of active flows multiplied by the round quantum.

In hardware, a sketch is realized using a simple *increment by x* primitive and predicated read-write registers (as described in [40]), both of which are available in reconfigurable switches. On packet arrival, `r` hashes of the flow's 5-tuple are computed to index into the register arrays and estimate the flow's finish round number, which is used to determine the packet's transmission schedule. In practice, AFQ re-uses one of several hashes that are already computed by the switch for Link Aggregation and ECMP. Today's devices support up to 64K register entries per stage and 12-16 stages [22], which is sufficient for a reasonably sized sketch per port to achieve good approximation, as we show in Appendix E.

3.3 Buffering Packets in Approximate Sorted Order

The BR fair queuing algorithm ensures that the packet with the lowest bid number is transmitted next at any point of time using a sorted queue. Since maintaining such a sorted queue is expensive, AFQ instead leverages the multiple FIFO queues available per port to approximate ordered departure of buffered packets, similar to timer wheels [46].

Assume there are N FIFO queues available at each egress port of the switch. AFQ uses each queue to buffer packets scheduled to depart within the next N rounds, where in each round, every active flow can send a fixed number of bytes, i.e., `BpR` bytes (bytes per round). We next describe how packets are enqueued and dequeued in approximate sorted order using these multiple queues.

3.3.1 Enqueue Module

The enqueue module decides which FIFO queue to assign to each packet. On arrival, the module retrieves the bid number associated with the flow's previous packet from the sketch. If it is lower than the starting bid number for the current round, the bid is pushed up to match the current round. The packet's bid number is then obtained by adding the packet's size to the previous bid number, and the packet's departure round number is computed as the packet's bid number divided by `BpR`. If this departure round exceeds N rounds in the future, the packet is dropped, else it is enqueued in the queue corresponding to the computed round number. Note that the current round number is a shared variable that the dequeue mod-

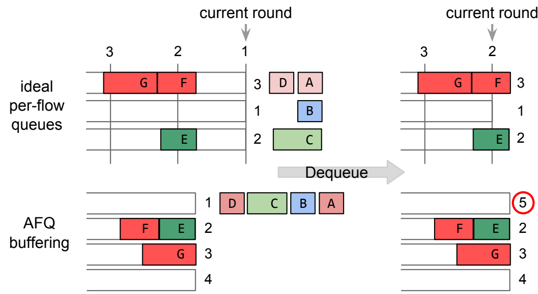


Figure 5: An example of the AFQ dequeue mechanism.

ule updates after it finishes draining a queue. Finally, the enqueue module updates the sketch to reflect the bid number computed for the current packet. Figure 4 shows an example of how AFQ works when various flows with variable packet sizes arrive at the same egress port.

Clearly, having more FIFO queues leads to finer ordering granularity and a better approximation of fair queuing. Switches available today support 24-32 queues per port [9, 12], which we show is sufficient for datacenter workloads. AFQ assumes that the total buffer assigned to each port can be dynamically assigned to any queue associated with that port. This lets AFQ to absorb a burst of new flow arrivals when several packets are scheduled for the same round number. Most switches already implement this functionality via dynamic buffer sharing [16].

3.3.2 Dequeue Module

The dequeue module transmits the packet with the smallest departure round number. Since the enqueue module already stores packets belonging to a given round number in a separate queue, AFQ must only drain the queue with the smallest round number. This is achieved by arranging all queues in strict priority, with the queue having the lowest round number assigned the highest priority. However, once empty, the queue must be bumped down to the lowest priority and the current round number incremented by 1. Note that this round number is shared with the enqueue module, which can then adjust its queueing behavior. The just-emptied queue is then used to store packets belonging to a future round number that is N higher than the current round number. Figure 5 shows the priority change and round assignment that occurs when a queue is drained to empty by a *Rotating Strict Priority* (RSP) scheduler. We describe in Section 4 how to implement this scheduler on reconfigurable switches.

An important implication of this design is that updating the current round number becomes trivial – increment by 1 whenever the current queue drains completely. Unlike the BR fair queuing algorithm, which must update the round number on every packet arrival, this coarse increment does not involve any complex computations or extra packet state, making it much more feasible to implement on reconfigurable switches.

3.4 Discussion

Several approximations govern how closely the AFQ design can emulate ideal fair queueing and present a fairness versus efficiency trade-off.

Impact of approximations: First, using a count-min sketch means that AFQ can over-estimate a packet's bid number in case of collisions. As the number of active flows grows beyond the size of the sketch, the probability of collisions increases, causing packets to be scheduled later than expected. However, as we show in the Appendix E, the sketch must be sufficiently large to store state only for active flows that have a packet enqueued at the switch, not all flows traversing the switch, including dormant ones that have not transmitted recently.

Second, unlike the BR fair queueing algorithm, which transmits one bit from each flow per round, AFQ lets active flows send multiple bytes per round. Since this departure round number is coarser than the bid number and AFQ buffers packets with the same round number in FIFO order, packets with higher bid numbers might be transmitted before packets with lower bid numbers if the switch received them earlier. This reordering can lead to unfairness within the round, but is bounded by number of active flows times `BpR` in the worst case.

`BpR` trade-off: Since AFQ buffers packets for the next N rounds only, the `BpR` must be chosen carefully to balance fairness and efficient use of the switch buffer. If `BpR` is too large, a single flow can occupy a large portion of the buffer, causing unfair packet delays and drops. If it is too small, AFQ will drop packets from a single flow burst despite having sufficient space to buffer them. The choice of `BpR` depends on network parameters, such as round trip times and link speeds, switch parameters, such as number of FIFO queues per port and total amount of packet buffer, as well as the endhost flow control protocol. We discuss how to set the `BpR` parameter after we describe the end-host transport protocol, which prescribe the rate adaptation mechanisms and determine the desired queue buildups on the switch.

4 Rotating Strict Priority (RSP) Scheduler

Given the Figure 3 pseudocode, implementing the dequeue module appears to be trivial. However, some hardware constraints make it more challenging than it seems. First, the two modules are generally implemented as separate blocks in hardware, which drives considerations regarding the sharing of state and synchronization issues between them. This is important since the decision of which queue to insert the packet into, or whether to drop the packet altogether, depends on the current round number. Second, the RSP scheduler, a custom mechanism, requires a queue's priority to be adjusted with respect to all other queues after it is completely drained by the

dequeue module. This mechanism is currently not supported, so we explore multiple ways to implement the RSP scheduler on today's hardware.

Synchronizing the enqueue and dequeue modules. Our design requires the current round number to be shared and synchronized between the two modules. The RMT architecture outlined in [9] does not permit the sharing of state across pipelines stages or pipelines due to performance considerations, but other reconfigurable switches that support the disaggregated RMT model [15, 12] do not impose this constraint. However, a workaround on the RMT architecture is possible if we make the following modifications to the enqueue module. Instead of explicitly receiving a signal regarding round completion (through the increment of the round number), the enqueue model can maintain a local estimate of the round number and infer round completion by obtaining queue metadata regarding its occupancy.

An empty queue corresponding to a given round number implies that the queue has been completely drained, and the enqueue module then locally increments its estimate of the round number and tries adding the packet to the next queue. Eventually, the enqueue module will identify a queue that is either not empty or that corresponds to a round number that it has not previously assigned to any incoming packet; it then assigns the packet to this queue. Note that we have replaced explicit signaling by providing access to queue occupancy data, which is supported on reconfigurable switches such as Barefoot's Tofino and Cavium Xpliant, at least at a coarse-grain level (i.e., pipeline stages have access to a coarse-grain occupancy level for each queue, if not the exact number of bytes enqueued).

Emulating RSP using a generic DRR scheduler. Deficit Round Robin (DRR) is a scheduling algorithm that guarantees isolation and fairness across all queues serviced. It proceeds in rounds; in each round it scans all non-empty queues in sequence and transmits up to a configurable *quantum* of bytes from each queue. Any deficit carries over to the next round unless the queue is empty, in which case the deficit is set to zero. We note that RSP is simply a version of DRR with the quantum set to a large value that is an upper-bound on the number of bytes transmitted by flows in a round. With a very high quantum, a queue serviced in DRR is never serviced again until all other queues have been serviced. This is equivalent to demoting the currently serviced queue to the lowest priority. Crucially, the DRR emulation approach indicates that the hardware costs of realizing RSP should be minimal since we can emulate its functionality using a mechanism that has been implemented on switches. However, we note that many modern switches implement a more advanced version of DRR, called Shaped DWRR, a variant that performs round robin scheduling of packets from queues with non-zero deficit counters in order to avoid long delays and unfairness. Unfortunately, the RSP mechanism cannot be emulated directly using DWRR due to its use of round robin scheduling across active queues.

Emulating RSP using strict priority queues. We now consider another emulation strategy that uses periodic involvement of the switch-local control plane CPU to alter the priority levels of the available egress queues. When the priority level for a queue is changed, typically through a PCIe write operation, the switch hardware instantaneously uses the queue's new priority level to determine packet schedules. The challenge here is that the switch CPU cannot make repeated updates to the priority levels given its clock speed and the PCIe throughput. We therefore designed a mechanism that requires less frequent updates to the priority levels (e.g., two PCIe operations every 10us) using hierarchical schedulers.

Our emulation approach splits the FIFO queues into two strict priority groups and defines hierarchical priority over the two groups. All priority level updates are made by switching the upper-level priority of the two sets of queues; these updates are made only after the system processes a certain number of rounds. Suppose we have $2 \times n$ queues, split into two groups (G^1, G^2) of n queues each. In each group, all n queues are serviced using strict priority. Initially, G^1 has strict priority over G^2. Packets with round number $1 \rightarrow n$ are enqueued in $G^1_{1\rightarrow n}$, whereas packets with round $(n+1) \rightarrow 2n$ are enqueued in $G^2_{1\rightarrow n}$. Packets with a round number greater than that are dropped. After a period τ, or when all queues in G^1 are empty, we switch the priorities of G^1 and G^2, making all queues of G^2 higher priority than G^1. Queues in each group retain their strict priority ordering. After the switch, we allow packets to be enqueue on G^1's queues for rounds corresponding to $(2n+1) \rightarrow 3n$.

This approach is feasible using hierarchical schedulers available in most ToR switches today. It reduces the number of priority transitions the switch must make and is implementable with the help of the management/service CPU on the switch. The time period τ depends on the link-rate and number of queues. Our experiments with the Cavium Xpliant switch indicate that $\tau = 10\mu s$ is both sufficient and supportable using the switch CPU. The disadvantage of this emulation approach is that the number of active queues the system can use could drop from $2n$ to n at certain points in time. However, our evaluations show that AFQ can perform reasonably well even with a small number of queues (viz., 8 queues for 40 Gbps links).

5 End-host Flow Control Protocol

Although AFQ is solely a switch-based mechanism that can be deployed without modifying existing end-hosts to achieve significant performance improvement, a network-enforced fair queuing mechanism lets us optimize the end-host flow control protocol to extract even more gains. This section describes our approach, adapted from literature, for performing fast ramp ups and keeping queue sizes small at switches. If all network switches provided fair allocation of bandwidth, the bottleneck bandwidth could be measured using the packet-pair approach [26], which sends a pair of packets back-to-back and measures the inter-arrival gap.

Packet-pair flow control. We briefly describe the packet-pair flow control algorithm. At startup, two packets are sent back-to-back at line-rate, and the returning ACK separation is measured to get an initial estimate of the channel RTT and bottleneck bandwidth. Normal transmission begins by sending packet-pairs paced at a rate equal to the estimated bandwidth. For every packet-pair ACK received during normal transmission, the bandwidth estimate is updated and the packet sending rate adjusted accordingly. If the bandwidth estimate decreases, a few transmission cycles are skipped, proportional to the rate decrease, to avoid queue buildup. Similarly, when the bandwidth estimate increases, a few packets, again proportional to the rate increase, are injected immediately to maintain high link utilization as described in [24], which also studies the stability of such a control-theoretic flow control.

Although this approach works well for an ideal fair-queuing network, we need to make some modifications for it to be robust against approximations introduced by AFQ. The complete pseudocode of our flow control protocol is available in Appendix A.

Robust bandwidth estimation. Since AFQ transmits multiple bytes in a single round, the packet-pair approach can incorrectly estimate bottleneck bandwidth if two back-to-back packets are enqueued in the same round and transmitted one after the other. This is not an issue if the BpR is less than or equal to 1 MSS, where MSS is the maximum segment size of the packets in the pair, and it holds true for our testbed and simulations. However, if the BpR is greater than twice the MSS, we must ensure that the very first packet-pair associated with a flow maps onto different rounds to get a reasonable bandwidth estimate using the inter-arrival delay. We accomplished this by adding a delay of `BpR-MSS` bytes at line-rate in between the packet-pairs at the end-host. This careful spacing mechanism, described in [45] measures the cross-traffic observed in a short interval and extrapolates it to identify the number of flows traversing the switch at that juncture. The protocol records the packet-pair arrival gap at the receiver and piggybacks on the acknowledgment to avoid noise and congestion on the reverse path. To further reduce variance, the protocol keeps a running EWMA of bandwidth estimates in the last RTT and uses the average for pacing packet transmission.

Per-flow ECN marking. Unlike an ideal fair-queueing mechanism, where the packet with the largest round number is dropped on buffer overflow, AFQ never drops packets that have already been enqueued. As a result, AFQ must maintain short queues to absorb bursty arrival of new flows. To dissipate standing queues and keep them short, we rely on a DCTCP-like ECN marking mechanism. Each sender keeps track of the fraction of marked packets and instead of transmitting packets at the estimated rate, the protocol sends packets at estimated rate times $(1 - \alpha/2)$. This optimization ensures that any standing queue is quickly dissipated. Further, unlike simple drop-tail queues, AFQ lets us perform per-flow ECN marking, which we exploit by marking packets when the enqueued bytes for a flow exceed a threshold round number. We set this number to 8 rounds in our simulations, which keeps per-flow queues very short without sacrificing throughput.

Bounding burstiness. Finally, since we have a fairly accurate estimate of the base RTT and the fair-share rate for each flow, we bound the number of inflight packets to a small multiple of the available bandwidth delay product (BDP) – similar to the rate based TCP BBR [10], – currently set to 1.5x the BDP in our implementation. This reduces network burstiness, especially when new flows arrive, by forcing older flows to stop transmitting due to their reduced BDP. This optimization keeps queues short, avoiding unnecessary queue buildup and packet drops.

We now perform a simple back of the envelope calculation to determine how to set the `BpR` parameter. As noted, we can use any end-host mechanism with AFQ, including standard ones such as TCP and DCTCP. Prior work has shown that DCTCP requires a queue of size roughly $1/6^{th}$ of the bandwidth delay product for efficient link utilization [3]. If the average round-trip latency of the datacenter network is d and the peak line rate is l, then we require $d \times l/6$ amount of buffering for a single flow to ensure maximum link utilization. Further, if we have `nQ` queues in the system, then we set `BpR` to $d \times l/(6 \times nQ)$. In practice, this is less than a MSS for a 40 Gbps link, 20 us RTT, and 10-20 queues. Further, the amount of buffering required by a single flow can be even lower by using an end-host protocol that leverages packet-pair measurements (such as that described above). Section 6.2.3 provide empirical data from our experiments to show that our end-host protocol does indeed maintain lower levels of per-flow buffer buildup than traditional protocols and that packet drops are rare.

6 Evaluation

We evaluated AFQ's overall performance, fairness guarantees and feasibility using: (1) a hardware prototype based on a Cavium network processor within a small cluster, (2) large-scale packet-level simulations, and (3) a programmable switch implementation in P4.

6.1 Hardware Prototype

Existing reconfigurable switches do not expose the programmability of internal queues, we therefore built a prototype of an AFQ switch using a programmable network processor. The Cavium OCTEON platform [14] has a multi-core MIPS64 processor with on-board memory and 4x10Gbps network I/O ports alongside several hardware-assisted network/application acceleration units, such as a traffic manager, packet buffer management units, and security co-processors. All of these components are connected via fast on-chip interconnects providing high performance, low latency, and programmability for network applications ranging from 100Mbps to 200Gbps.

6.1.1 AFQ Switch Implementation

We built a 4-port AFQ switch on top of the network processor using the Cavium Development Kit [13]. Figure 6 shows the high-level architecture, which includes 4 ingress pipelines, 4 egress pipelines, 32 FIFO packet queues, and a count-min sketch table containing 4 rows and 16K columns. The number of ports was fixed due to hardware limitations while all other individual components, such as ingress/egress pipelines, queue and table sizes were configured based on available resources.

Each ingress and egress pipeline instance runs on a dedicated core, sharing access to packet buffer queues and the count-min sketch stored on the on-board DRAM. The ingress pipeline implements most of the AFQ functionality. First, it parses the packet and computes multiple hashes using on-chip accelerators for indexing into the count-min sketch. Next, it estimates the current round number for the packet using the algorithm shown in Figure 3. Finally it updates the count-min sketch and enqueues the packet in the queue corresponding to the estimated round number. The egress simply dequeues packets from the queue corresponding to the current round being serviced, re-encapsulates the packets and transmits them to the specific port based on a pre-loaded MAC table.

Each packet queue maintains a shared lock to avoid race conditions arising from concurrent accesses of the ingress and egress cores. Other queue state updates and sketch table reads/writes use lock-free operations. We use the software reference counting technique to avoid TOCTOU race conditions.

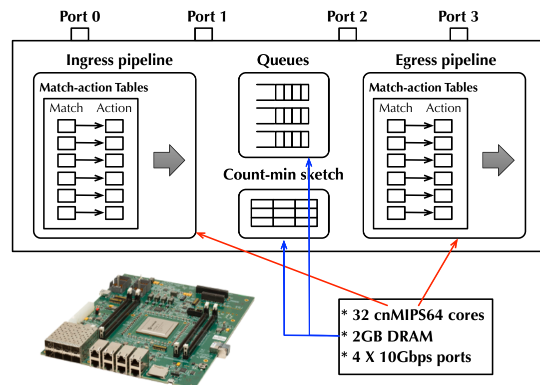


Figure 6: High-level architecture of the AFQ switch prototype.

6.1.2 End-host Protocol Implementation

We implemented the packet-pair flow control protocol (Section 5) in user-space on top of UDP and integrated it with our workload generator. The implementation uses hardware timestamps from the NIC to measure the spacing between packet-pairs to accurately obtain bandwidth estimate and RTT samples, similar to prior work [30]. The flow control re-implements standard TCP sequencing, fast retransmit, and recovery in user-space atop UDP.

6.1.3 Hardware Testbed and Workload

Our testbed includes 8 Supermicro servers, 2 Cavium XPliant switches and the prototype AFQ switch atop the network processor described above. All servers are equipped with 2x10Gbps port NICs. We created a 2-level topology using VLANs to divide the physical ports on the two switches. We integrated the prototyped AFQ switch into the aggregation switch which runs the AFQ mechanism at the second layer of the topology. The end-to-end latency is approximately $200\mu s$, most of which is spent inside the network processor.

We set up 4 clients and 4 servers that generated traffic using the enterprise workload described in [1], such that all traffic traversed the AFQ switch in the aggregation layer. Each client opened 25 concurrent long-running connections to each server, and requested flows according to a Poisson process at a rate configured to achieve desired network load. We compared four schemes,

- Default Linux TCP CUBIC with droptail queues
- DCTCP [2] with ECN marking droptail queues
- DCTCP with our AFQ mechanism
- Our packet-pair flow control with AFQ mechanism

For DCTCP, we enabled the default kernel DCTCP module and set the ECN marking threshold to $K = 65$ packets. For a fair comparison, we relayed the TCP and DCTCP traffic through our emulated switch.

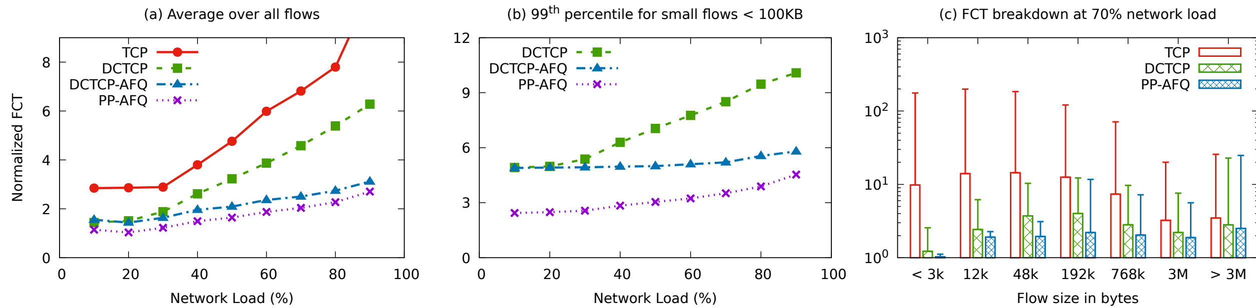


Figure 7: FCT summary for the enterprise workload on our hardware testbed. (a) average FCT for all flows, (b) tail latency for short flows, and (c) average and 99th percentile (using error bar) for various flow sizes. Note, TCP does not appear in (b) as its performance is outside the plotted range.

6.1.4 Overall Performance

We use flow completion time (FCT) as the evaluation metric and report the average and 99th percentile latency over a period of 60 seconds. Figure 7 shows FCT statistics for various flow sizes as we increase the network load; data points are normalized to the average FCT achieved in an idle network. AFQ improves DCTCP performance by 2x and TCP performance by 10x for both average and tail flow completion times. The benefits of AFQ are more visible at high network loads when there is substantial cross-traffic with high churn. In such a scenario, TCP and DCTCP take multiple RTTs to achieve fair bandwidth allocation, and suffer long queueing delays behind bursty traffic; whereas AFQ lets new flows achieve their fair share immediately and isolates them from other concurrent flows, leading to significantly more predictable performance.

Figure 7(b) also shows the improvement from our packet-pair end-host flow control over DCTCP, as the packet-pair approach avoids slow-start and begins transmitting at fair bandwidth allocation immediately after the first ACK. This fast ramp-up along with fair allocation at AFQ switches translates to significant FCT improvement, especially for short flows, as shown in Figure 7(c).

6.2 Software Simulation

We also studied AFQ's performance in a large-scale cluster deployment using an event-driven, packet-level simulator. We extended the mptcp-htsim simulator [36] to implement AFQ and several other comparison schemes.

6.2.1 Simulation Topology and Workload

We simulated a cluster of 288 servers connected in a leaf-spine topology, with 9 leaf and 4 spine switches. Each leaf switch is connected to 32 servers using 10Gbps links; and each spine switch is connected to each leaf using 40Gbps links. All leaf and spine switches have a fixed-sized buffer of 512KB and 1MB per port respectively. The end-to-end round-trip latency across the spine (4 hops) is $\approx 10\mu s$. All flows are ECMP load balanced across all spine switches. We use a small value of min-RTO = $200\mu s$ for all schemes, as suggested in [4].

We used both synthetic and empirical workloads derived from traffic patterns observed in productions datacenters. The synthetic workload generates Pareto distributed ($\alpha = 1.1$) flows with mean flow size 30KB. The empirical workload is based on an enterprise cluster reported in [1]. Flows arrive according to a Poisson process at randomly and independently chosen source-destination server pairs from all servers. The arrival rate is chosen to achieve a desired level of utilization in the spine links. Both workloads are heavy-tailed with majority bytes coming from a small fraction of large flows; both also have a diverse mix of short and long flows, with the enterprise workload having more short flows.

6.2.2 Comparison Schemes

- **TCP**: Standard TCP-Reno with fast re-transmit and recovery, but without SACKs, running on switches with traditional drop-tail queues
- **DCTCP**: The DCTCP [2] congestion control algorithm with drop-tail queues supporting ECN marking on all switches; marking threshold set to 20 packets for 10Gbps links and 80 packets for 40Gbps links
- **SFQ**: Same TCP-Reno as above with Stochastic Fair Queueing [29] using DRR [39] on all switch ports; with 32 FIFO queues available at each switch port
- **AFQ**: Our packet-pair flow control with AFQ switches using 32 FIFO queues per port, a count-min sketch of size 2x16384, and a `BpR` of 1 MSS
- **Ideal-FQ**: An ideal fair queueing router that implements the BR algorithm (described in [18]) and uses our packet-pair flow control at the end-host

6.2.3 Overall Performance

We compared the overall performance of various schemes in the simulated topology by measuring the FCT of all flows that finished over a period of 10 seconds in the simulation. Figures 8 and 9 show the normalized FCT (normalized to the average FCT achieved in an idle network) for all flows, short flows (<100KB) and flows bucketed across different sizes at varying network loads and workloads.

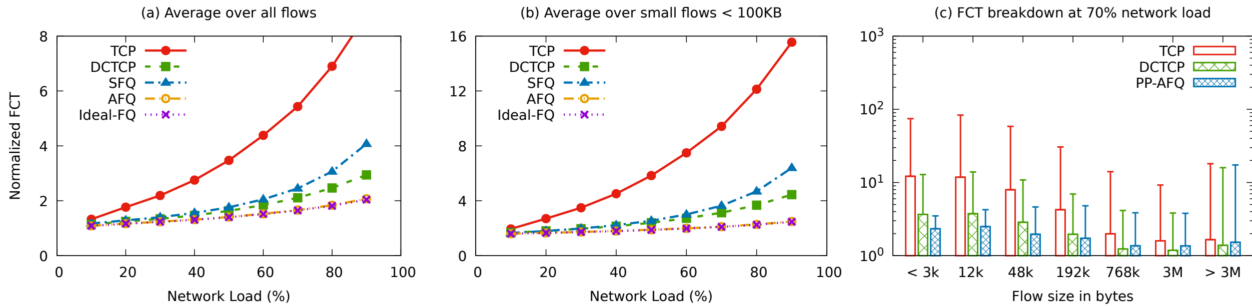


Figure 8: Flow completion times for synthetic workload in the cluster. (a) average FCT for all flows, (b) average FCT for flows shorter than 100KB, and (c) average and 99th percentile (using error bar) for various flow size buckets at 70% network load.

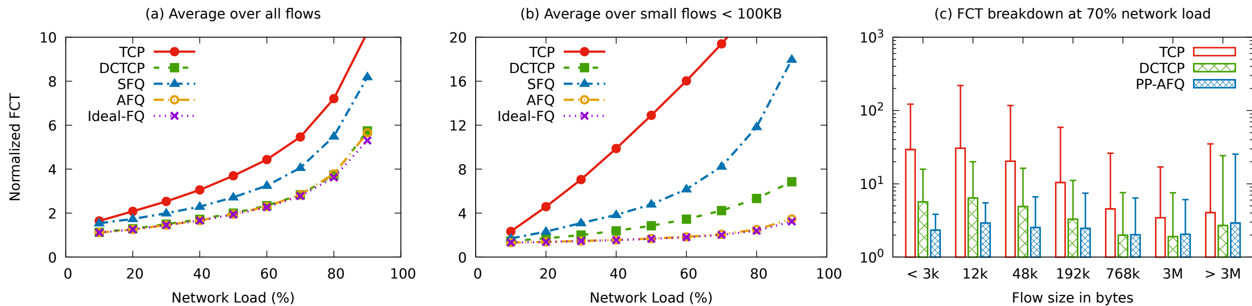


Figure 9: Flow completion times for enterprise workload, with each graph showing the same metrics as in Figure 8.

Our simulation results match previous emulated observations. As expected, most schemes perform close to optimal at low network load, but quickly diverge as network traffic increases. Traditional TCP keeps switch buffers full, leading to long queueing delays, especially for shorter flows. DCTCP improves the performance of short flows significantly since it maintains shorter queues, but is still a factor of 2-4x away from ideal fair-queuing behavior. SFQ works very well at low network loads when the number of active flows is comparable to number of queues, however as network traffic increases, collisions within a single queue become more frequent leading to poor performance. AFQ achieves close to ideal fair queuing performance for all network load, which is 3-5x better that TCP and DCTCP for tail latency of short flows: irrespective of other network traffic, all flows immediately get their fair share of the network without waiting behind other packets. This leads to significant performance benefit for shorter flows, which do not have to wait behind bursty traffic.

To further understand the performance gains, we measured several other metrics, such as packet drops, retransmissions, average queue lengths, and buffer occupancy distribution during the experiment. Figure 10(a) shows the average bytes dropped per flow for each scheme. As expected, standard TCP drops on average one packet per flow, and DCTCP has negligible drops at low network load. However, at higher loads, drops are more frequent, leading to occasional re-transmission and performance penalty. This is also reflected in the average queue length shown in Figure 10(b). Both DCTCP and packet pair with AFQ are able to maintain very short queues, but with an interesting difference in the buffer occupancy distribution as shown in Figure 10(c). We took periodic snapshots on the queue every $100\mu s$, to count how many packets belong to each flow in the buffer and plotted the CCDF of number of packets per flow across all snapshots. AFQ with packet-pair flow control rarely has more than 5 packets enqueued per flow at the core links, whereas DCTCP and TCP have many more packets buffered per flow. This can lead to unfairness when bursty traffic arrives, such as during an incast, which we discuss next. In summary, AFQ achieves similar performance to DCTCP for all flows, and 2x better performance for short flows while maintaining shorter queues and suffering fewer drops by ensuring fair allocation of bandwidth and buffers.

6.2.4 Incast Patterns

Incast patterns, common in datacenters, often suffer performance degradation due to poor isolation. In this setup, we started a client on every end-host which requests a chunk of data distributed over N other servers. Each sender replies back with $1/N$ of the data at the same time. We report the total transfer time of the chunk of data with a varying number of senders averaged over multiple runs. Simultaneous multiple senders can cause unfair packet drops for flow arriving later, causing timeouts that delay some flows and increase overall completion time. An ideal fair-queuing scheme would allocate

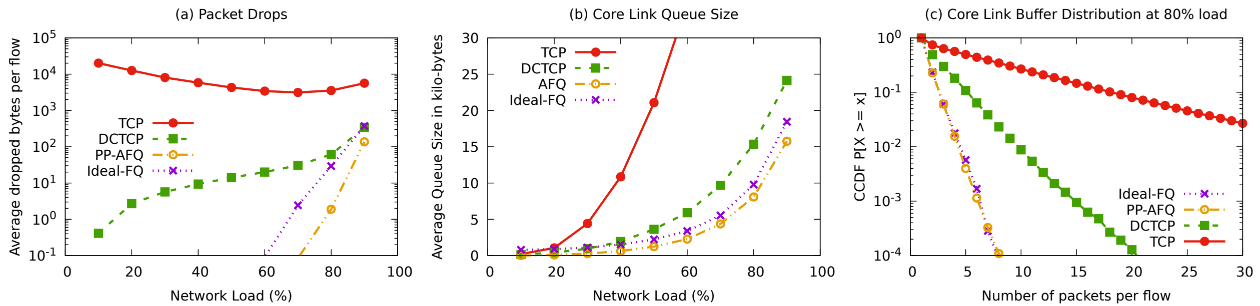


Figure 10: Packet drops, queue lengths and buffer occupancy distribution for enterprise workload in the cluster.

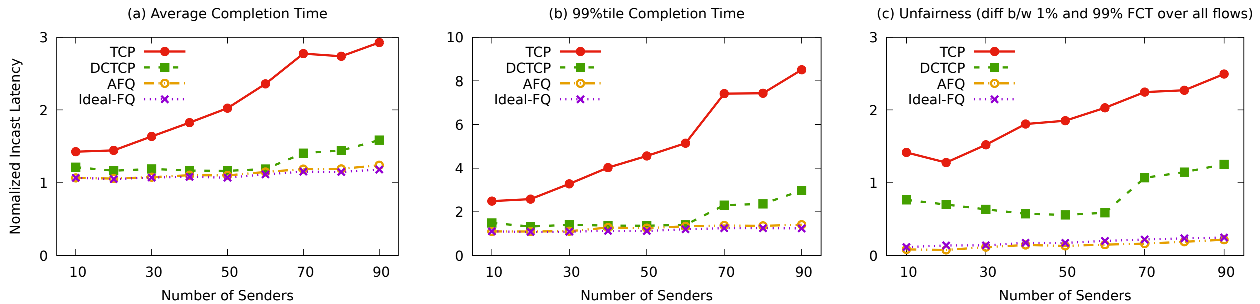


Figure 11: Completion time summary for an incast request of size 1.5MB from a varying number of senders.

equal bandwidth and buffer to each sender, hence finishing all transfers at roughly the same time.

Figure 11 shows the total completion time of various schemes for a total chunk size 1.5MB with varying number of senders. The receiver link has approximately 300KB of buffer, roughly around 200 packets. Most schemes perform well with few senders, but degrade when the number of senders overwhelms the receiver link buffer. This leads to packet drops for flows arriving later in a traditional droptail queue, sending them into timeouts. AFQ achieves close to optimal request completion time, even with large senders because it ensures each flow gets fair buffer allocation regardless of when it arrives. As a result packet drop are minimal, leading to fewer re-transmissions and lower completion time. Figure 12 shows the number of packet drops observed during the incast, packet re-transmissions, and buffer occupancy, which confirm the preceding observation. As expected, TCP drops several packets throughout the incast experiment, causing several re-transmissions. DCTCP performs much better and suffers zero packet drops until the number of senders exceeds the link buffer capacity. AFQ has even fewer drops than DCTCP because it distributes the available buffer space in a fair manner among all flows, as shown in Figure 12(c).

6.3 P4 Implementation

To evaluate the overhead of implementing AFQ on an actual reconfigurable switch, we expressed AFQ in the P4 programming language and compiled it to a production switch target. The P4 code ran on top of a baseline switch implementation [41] that provides common functionality of today's datacenter switches, such as basic L2 switching (flooding, learning, and STP), basic L3 routing (IPv4, IPv6, and VRF), link aggregation groups (LAGs), ECMP routing, VXLAN, NVGRE, Geneve and GRE tunneling, and basic statistics collection. The compiler implements the functionality proposed in [23] and compiles to the hardware model described in Section 2.2. It reports the hardware usage of various resources for the entire implementation.

Resource	Baseline	+AFQ		+AFQ-Large	
Pkt Header Vector	187	191	+2%	191	+2%
Pipeline Stages	9	12	+33%	12	+33%
Match Crossbar	462	465	+1%	465	+1%
Hash Bits	1050	1082	+3%	1092	+4%
SRAM	165	178	+8%	190	+15%
TCAM	43	44	+2%	44	+2%
ALU Instruction	83	90	+8%	90	+8%

Table 1: Summary of resource usage for AFQ.

Table 1 shows the additional overhead of implementing two variants of AFQ as reported by the compiler. AFQ uses a count-min sketch of size 2x2048, while AFQ-Large uses a sketch of size 3x16384. We can see the extra overhead is small for most resources. We need more pipeline stages to traverse the count-min sketch and keep a running minimum, and more SRAM to store all the flow counters. We also use extra ALU units to perform per-packet increments and bit-shifts to divide by `BpR`.

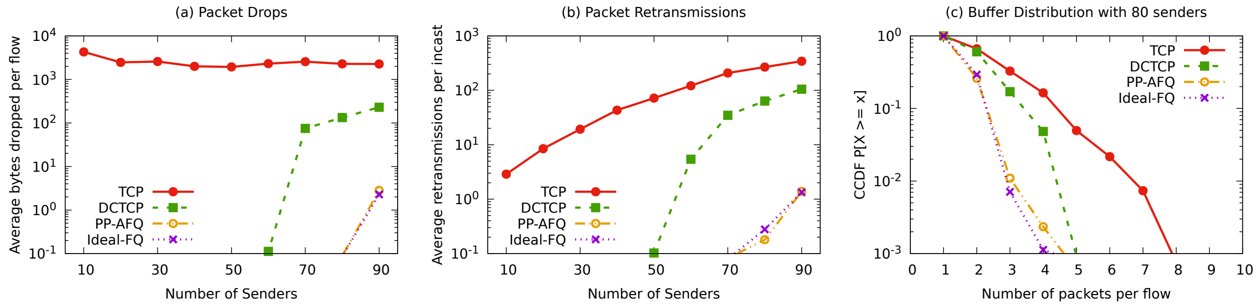


Figure 12: Packet drops, re-transmissions and buffer distribution across flows during incast traffic.

7 Related Work

Starting from Nagle's proposal [31] for providing fairness by using separate queues, several algorithms have been designed to implement a fair queueing mechanism inside the network. In [18], an efficient bit-by-bit round robin (BR) algorithm was developed to provide ideal fair queuing without per-flow queue support; [25] describes an efficient implementation of the BR algorithm. However, its inherent complexities make it hard to implement on today's high-speed routers.

Many algorithms were later proposed to reduce the complexity and cost of implementing fair queuing mechanisms. Most either use stochastic approaches or avoid complex per-flow management by using simpler heuristics. Stochastic Fair Queuing (SFQ) [29] hashes flows onto a reduced set of FIFO queues and perturbs the hashing function periodically to minimize unfairness. An efficient realization of SFQ using Deficit Round Robin (DRR) was proposed in [39]. However, its fairness guarantees are closely tied to the number of queues, and performance degrades significantly when the number of active flow exceeds the number of queues.

Several schemes [33, 28, 27] enforce fairness by dropping packets of flows sending faster than their fair share. They estimate flow rate by tracking recent history of packet arrivals or the current buffer occupancy. A variant, called Stochastic Fair Blue [19], uses an array of bloom filters to store packet counts and drop probabilities, which is similar to how AFQ stores round numbers. Core-Stateless Fair Queueing (CSFQ) [44] enforces fair allocation by splitting the mechanism between the edge and the core network. All complexity of rate estimation/labeling is at the edge, and the core performs simple packet forwarding based on labels. It achieves fairness by dropping packets with probability proportional to the rate above the estimated fair rate.

Other schemes – PIAS [5] and FDPA [11] also leverage multiple priority queues available in commodity switches to emulate shortest-job-next scheduling or achieve approximate fair bandwidth allocation using an array of rate estimators to assign flows to different priority queues. Although similar, AFQ uses multiple queues to emulate an ideal fair-queueing algorithm. A more recent approach PIFO [42] proposes a programmable scheduler that can implement variants of priority scheduling and ideal fair queuing at line rate by efficiently implementing $O(logN)$ sorted insertion complexity in hardware. However, like fixed-function schedulers, the number of distinct flows that can be scheduled is bound in hardware and can support up to 2048 flows. In addition, we discuss how to store approximate per-flow bid numbers in limited switch memory, and increment current round number efficiently, which has not been explored in prior work.

8 Conclusion

In this paper, we proposed a fair bandwidth allocation mechanism called Approximate Fair Queueing (AFQ), designed to run on emerging reconfigurable switches. We approximate the various mechanisms of a fair queueing scheduler using features available on reconfigurable switches. Specifically, we approximate the per-flow state regarding the number and timing of its previously transmitted packets using mutable switch state; we perform limited computation for each packet to compute its position in the output schedule; we dynamically determine which egress queue to use for a given packet; and we design a new dequeuing approach, called the Rotating Strict Priority scheduler, to transmit packets in approximate sorted order. Using a networking-processor-based prototype in a real hardware testbed and large scale simulations, we showed that AFQ approximates ideal queuing behavior accurately, improving performance significantly over existing schemes. We also showed that the overhead of implementing AFQ on top of programmable switches is fairly minimal.

Acknowledgments

We would like to thank the anonymous NSDI reviewers and our shepherd Mohammad Alizadeh for their valuable feedback. We also thank Antoine Kaufmann for many insightful discussions. This research was partially supported by the National Science Foundation under Grants CNS-1518702, CNS-1616774, and CNS-1714508.

References

[1] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., AND VARGHESE, G. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the ACM SIGCOMM Conference* (2014).

[2] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference* (2010).

[3] ALIZADEH, M., JAVANMARD, A., AND PRABHAKAR, B. Analysis of DCTCP: Stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS Conference* (2011).

[4] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM Conference* (2013).

[5] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Information-agnostic flow scheduling for commodity data centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA, 2015).

[6] BAREFOOT NETWORKS. Tofino Programmable Switch. https://www.barefootnetworks.com/technology/.

[7] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement* (2010).

[8] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (July 2014).

[9] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM Conference* (2013), pp. 99–110.

[10] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., AND JACOBSON, V. BBR: Congestion-based congestion control. *Queue 14*, 5 (Oct. 2016), 50:20–50:53.

[11] CASCONE, C., BONELLI, N., BIANCHI, L., CAPONE, A., AND SANSÒ, B. Towards approximate fair bandwidth sharing via dynamic priority queuing. In *Local and Metropolitan Area Networks (LANMAN)* (2017), IEEE.

[12] CAVIUM. XPliant Ethernet switch product family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[13] CAVIUM. OCTEON Development Kits, 2016. http://www.cavium.com/octeon_software_develop_kit.html.

[14] CAVIUM. Cavium OCTEON SoC Development Board, 2017. http://www.cavium.com/OCTEON_MIPS64.html.

[15] CHOLE, S., FINGERHUT, A., MA, S., SIVARAMAN, A., VARGAFTIK, S., BERGER, A., MENDELSON, G., ALIZADEH, M., CHUANG, S.-T., KESLASSY, I., ORDA, A., AND EDSALL, T. dRMT: Disaggregated programmable switching. In *Proceedings of the ACM SIGCOMM Conference* (2017).

[16] CHOUDHURY, A. K., AND HAHNE, E. L. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions on Networking 6*, 2 (1998), 130–140.

[17] CORMODE, G., AND MUTHUKRISHNAN, S. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[18] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *Proceedings on the ACM SIGCOMM Conference* (1989).

[19] FENG, W.-C., KANDLUR, D. D., SAHA, D., AND SHIN, K. G. Stochastic Fair Blue: A queue management algorithm for enforcing fairness. In *IEEE INFOCOM* (2001).

[20] JAFFE, J. Bottleneck flow control. *IEEE Transactions on Communications 29*, 7 (1981), 954–962.

[21] JAIN, R., CHIU, D., AND HAWE, W. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR cs.NI/9809099* (1998).

[22] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[23] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling packet programs to reconfigurable switches. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015).

[24] KESHAV, S. A control-theoretic approach to flow control. In *Proceedings of the ACM SIGCOMM Conference* (1991).

[25] KESHAV, S. On the efficient implementation of fair queueing. *Journal of Internetworking: Research and Experience 2* (1991), 157–173.

[26] KESHAV, S. The packet pair flow control protocol. Tech. Rep. 91-028, ICSI Berkeley, 1991.

[27] LIN, D., AND MORRIS, R. Dynamics of random early detection. In *ACM SIGCOMM Computer Communication Review* (1997).

[28] MAHAJAN, R., FLOYD, S., AND WETHERALL, D. Controlling high-bandwidth flows at the congested router. In *Network Protocols, 2001. Ninth International Conference on* (2001), IEEE, pp. 192–201.

[29] MCKENNEY, P. E. Stochastic fairness queueing. In *INFOCOM'90, IEEE Computer and Communication Societies. The Multiple Facets of Integration. Proceedings, IEEE* (1990).

[30] MITTAL, R., LAM, V. T., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., AND ZATS, D. TIMELY: RTT-based congestion control for the datacenter. In *Proceedings of the ACM SIGCOMM Conference* (2015).

[31] NAGLE, J. B. On packet switches with infinite storage. In *Innovations in Internetworking*. Artech House, Inc., 1988, pp. 136–139.

[32] OZDAG, R. Intel® Ethernet Switch FM6000 Series-Software Defined Networking. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[33] PAN, R., BRESLAU, L., PRABHAKAR, B., AND SHENKER, S. Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communication Review 33*, 2 (Apr. 2003).

[34] PAREKH, A. K., AND GALLAGER, R. G. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking 1*, 3 (1993), 344–357.

[35] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM Conference* (2012).

[36] RAICIU, C. MPTCP htsim simulator. http://nrg.cs.ucl.ac.uk/mptcp/implementation.html.

[37] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *Proceedings of the ACM SIGCOMM Conference* (2015).

[38] SHARMA, N. K., KAUFMANN, A., ANDERSON, T., KRISHNAMURTHY, A., NELSON, J., AND PETER, S. Evaluating the power of flexible packet processing for network resource allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, 2017), pp. 67–82.

[39] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queueing using deficit round robin. In *Proceedings on the ACM SIGCOMM Conference* (1995).

[40] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., MCKEOWN, N., AND LICKING, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[41] SIVARAMAN, A., KIM, C., KRISHNAMOORTHY, R., DIXIT, A., AND BUDIU, M. DC.P4: Programming the forwarding plane of a data-center switch. In *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research* (2015).

[42] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the ACM SIGCOMM Conference* (2016).

[43] SONG, H. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013).

[44] STOICA, I., SHENKER, S., AND ZHANG, H. Core-Stateless Fair Queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proceedings on the ACM SIGCOMM Conference* (1998).

[45] STRAUSS, J., KATABI, D., AND KAASHOEK, F. A measurement study of available bandwidth estimation tools. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement* (2003).

[46] VARGHESE, G., AND LAUCK, A. Hashed and hierarchical timing wheels: Efficient data structures for implementing a timer facility. *IEEE/ACM Transactions on Networking 5*, 6 (Dec. 1997), 824–834.

A End-host Pseudocode

Figure 13 shows our adapted packet-pair flow control at the end-host. Each flow begins by transmitting a pair of back-to-back packets and waits for the acks to return. The receiver measures the packet inter-arrival gap and returns it back to the sender piggybacked on the ack. After receiving the first ack, it starts normal transmission at the estimated rate in packet-pairs. For every ack received during normal transmission, the rate estimate is updated based on the packet gap and ECN marks.

SENDER PROTOCOL

```
Startup():
  state = STARTUP
  SendPacketPair()

On AckReceive(pktpair, rtt):
  newGap = pktpair.gap

  if (rtt < minRTT):
    minRTT = rtt

  if state == STARTUP:
    /* Start normal packet transmission. */
    state = NORMAL
    gap = newGap
    SendPacketPair()
  else:
    /* Update rate estimate. */
    gap = (1 - GAIN) * gap + GAIN * newGap
    linkRate = MSS / gap
    bdp = linkRate * minRTT

    /* Throttle rate based on ECN marks. */
    rate = linkRate * (1 - alpha / 2)

SendPacketPair():
  /* Bound inflight bytes to roughly bdp. */
  if (inflight > CWND_FACTOR * bdp):
    /* Wait for ack or retransmission timeout. */
    return

  packet1 = nextPacket()
  packet1.first = true
  send(packet1)

  /* Add delay if necessary. */

  packet2 = nextPacket()
  send(packet2)

  if (state == STARTUP):
    Wait for AckReceive()
  else:
    nextSendTime = now() + 2 * MSS / rate
    scheduleTimer(SendPacketPair, nextSendTime)
```

RECEIVER PROTOCOL

```
OnPacketReceive (packet):
  if (packet.first == true):
    first_pktpair_time = now()
    pktpair_ts = packet.sendTime
  else:
    gap = now - first_pktpair_time
    ack = nextAck()
    ack.sendTime = pktpair_ts
    ack.gap = gap
    send(ack)
```

Figure 13: Pseudocode for endhost flow control protocol

B Convergence and Fairness

To demonstrate that AFQ does indeed assign each flow its fair share rapidly, we connected two hosts via a 10Gbps, $10\mu s$ RTT link and sequentially started-stopped flows at 1-second intervals. We used standard TCP end-hosts, and change the queueing mechanism from droptail to AFQ. The time series in Figure 14 shows the throughput achieved by each flow as they enter and exit the link. AFQ assigns each flow its fair share immediately, while a droptail queue exhibits high variance in throughput.

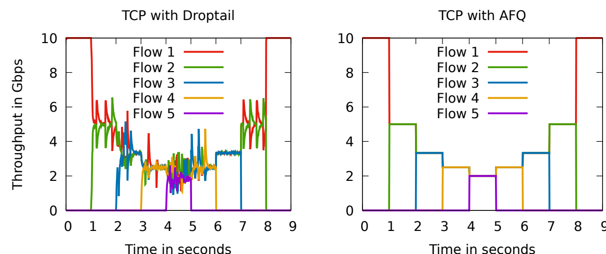


Figure 14: Convergence test

Next, we plot the FCT versus flow size from our cluster simulations in Figure 15 to demonstrate how fair each scheme is with respect to flow size. An ideal fair queuing scheme would be a straight line from the origin. All schemes achieve fairness over a period of time for long flows, but are significantly unfair to short flows either due to slow-start or queueing behind other flows in the network. AFQ lets all flows, regardless of size to achieve their fair share within an RTT, leading to better fairness.

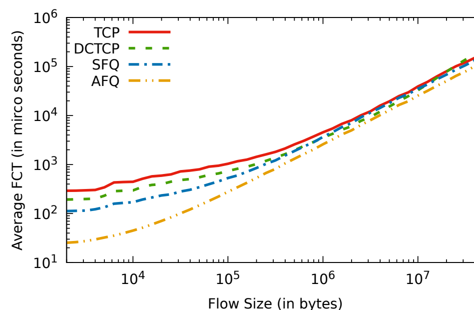


Figure 15: FCT vs flow size at 70% network load.

To further study AFQ's fairness guarantees, we simulated a 10Gbps, $25\mu s$ RTT link and increased the number of on-off senders transmitting concurrent flows on the link. We measured the Jain Unfairness index (1−Jain Fairness [21]) across all flows. Figure 16(a) shows the unfairness across different queueing schemes. AFQ has better fairness than other schemes, until the number of concurrent flows exceeds the sketch size. Figures 16(b) and (c), plot the same metric while varying the sketch-size and number of FIFO queues available to AFQ.

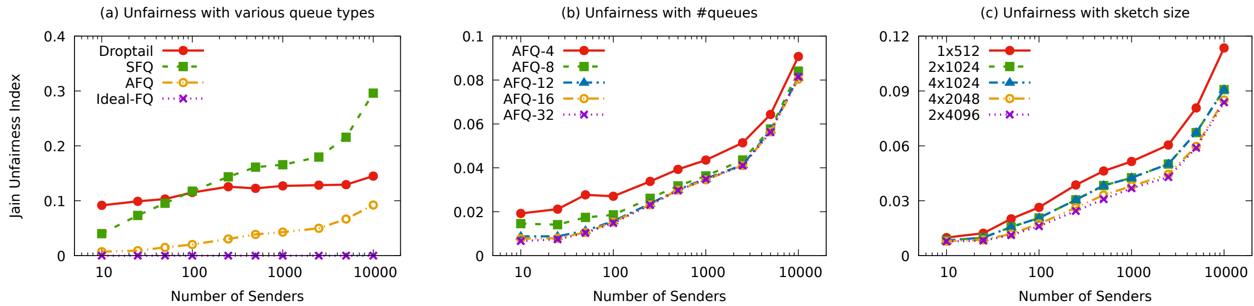
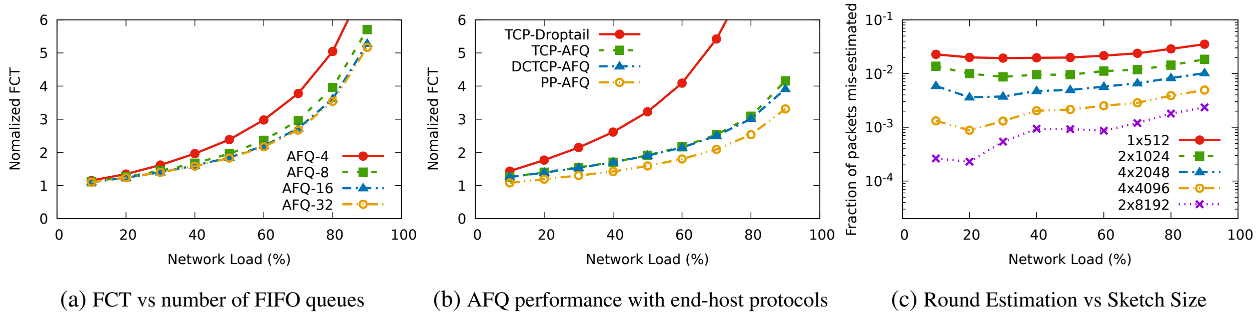


Figure 16: Micro-benchmarks showing deviation of various queuing mechanism compared to ideal fair queuing using a DCTCP end-host.



(a) FCT vs number of FIFO queues

(b) AFQ performance with end-host protocols

(c) Round Estimation vs Sketch Size

Figure 17: Benchmarks showing the impact of various AFQ parameters.

C Impact of Number of Queues on FCT

AFQ uses multiple FIFO queues to store packets in an approximate sorted order. To understand how many FIFO queues are required per-port to get accurate fair-queueing behavior, we ran the same enterprise workload while varying the number of FIFO queues available to the AFQ implementation and keeping `BpR` fixed at 1 MSS. Figure 17a shows the impact on average FCT of all flows as we varied the number of queues from 4 to 32. When fewer queues are available, AFQ buffers packets for very few rounds at any given time. This causes unnecessary packet drops during bursty arrivals, and also leads to poor bandwidth estimation at the endhost. Once there are sufficient queues to absorb packet bursts and accurately estimate bottleneck bandwidth, AFQ achieves near ideal fair queueing behavior, which occured around 16-20 queues. This is not surprising, given the analysis from [2], a queue of size roughly $1/6^{th}$ of the bandwidth delay product is required for efficient link utilization. For our testbed with 40 Gbps links and $20\mu s$ RTT, this value is ≈20KB, translating to about 15 queues.

D AFQ with Other End-host Protocols

As an in-network switch mechanism, AFQ can be deployed without modifying the end-host to achieve significant performance gains. To quantify the benefits, we simulate the same enterprise workload using TCP, DCTCP end-hosts with all switches implementing the AFQ mechanism. Figure 17b shows the significant improvement in average FCT when switching from droptail to AFQ behavior inside the network. Moving to DCTCP gives another small improvements due to shorter queues; finally, using packet-pair flow control eliminates slow-start behaviors, further reducing FCT. This matches our observations from the hardware prototype emulation.

E Impact of Sketch Size

AFQ stores bid numbers in a count-min sketch, trading off space for accuracy. To determine how large a sketch is required to achieve sufficient accuracy without affecting performance, we re-ran the enterprise workload in the leaf-spine topology while tracking exact bid numbers and those returned by the count-min sketch. During the 10 second simulation run, we count how many times a packet bid number was mis-estimated and enqueued in a later-than-expected queue. Figure 17c shows the mis-estimation rate as we change the sketch size. This is less than 1% using a relatively small sketch of 2x1024. This is not surprising since the collision probability is proportional to number of active flows that have packets enqueued at the switch, which is generally a few tens to hundreds. It is not affected by the total number of ongoing flows, which could be several thousands. Such a low rate of mis-estimation does not significantly impact the flow-level performance, because a bad estimate delays the packet by only a small amount of time. Further, increasing the number of cells in each row has a more significant impact on the accuracy than increasing the number of rows.

PASTE: A Network Programming Interface for Non-Volatile Main Memory

Michio Honda[†], Giuseppe Lettieri[‡], Lars Eggert[★] and Douglas Santry[★]
NEC Laboratories Europe[†], Università di Pisa[‡], NetApp[★]

Abstract

Non-Volatile Main Memory (NVMM) devices have been integrated into general-purpose operating systems through familiar file-based interfaces, providing efficient byte-granularity access by bypassing page caches. To leverage the unique advantages of these high-performance media, the storage stack is migrating from the kernel into user-space. However, application performance remains fundamentally limited unless network stacks explicitly integrate these new storage media and follow the migration of storage stacks into user-space. Moreover, we argue that the storage and the network stacks must be considered together when being designed for NVMM. This requires a thoroughly new network stack design, including low-level buffer management and APIs.

We propose PASTE, a new network programming interface for NVMM. It supports familiar abstractions—including busy-polling, blocking, protection, and run-to-completion—with standard network protocols such as TCP and UDP. By operating directly on NVMM, it can be closely integrated with the persistence layer of applications. Once data is DMA'ed from a network interface card to host memory (NVMM), it never needs to be copied again—even for persistence. We demonstrate the general applicability of PASTE by implementing two popular persistent data structures: a write-ahead log and a B+ tree. We further apply PASTE to three applications: Redis, a popular persistent key-value store, pKVS, our HTTP-based key value store and the logging component of a software switch, demonstrating that PASTE not only accelerates networked storage but also enables conventional networking functions to support new features.

1 Introduction

Non-volatile main memory (NVMMs) [49] has the potential to change the way modern systems are designed and implemented[1]. The memory hierarchy, with CPU registers at the top and persistent storage at the bottom, has changed little since the early 1970s. The media available at the bottom of the hierarchy, i.e., block-based persistent storage, has grown to offer a wider spectrum of choices, but ephemeral DRAM has ruled supreme as main memory.

Durable main memory will precipitate sweeping changes to how systems are designed end-to-end. The entire processing cycle of an application will change. Storage and networking, in the form of user-level libraries, will become inextricably intertwined with application logic, instead of maintaining the clean separation offered by the kernel APIs today (e.g., POSIX).

This paper addresses this space by examining the ramifications of NVMM from the perspective of an application—not the storage system—and offers a means of leveraging NVMM from the earliest stage of a server's request cycle. In particular, this paper addresses the following question: *What should the end-to-end data path—across a NIC, the network stack, an application and a persistent data store—look like?*

Consider a transactional data transfer. The NIC on the receiver writes an incoming packet to main memory via a DMA, then the kernel network stack processes the packet. The application then reads the packet data (if the socket API is used, this involves a data copy) and processes it. Processing a transaction can result in side-effects to persistent data structures (e.g., adding a row to a table). The semantics typically require the application to accept and persist a transaction prior to acknowledging it as successful. As persistence is required, and updating the primary data structure on disk (e.g., in a database table) is very slow, it is common practice to use a write-ahead log to speed up transaction processing. Using Write-ahead logs is much faster than updating a primary data structure, as it simply involves serially appending to a log of accepted transactions. The primary data structure is persisted periodically and corresponding log entries are discarded. Accepting a transaction thus involves updating the primary data structure in memory (but not pushing it to disk) and copying data to a write-ahead log.

Today, since the end-to-end latencies of transactional data transfers are dominated by slow block-device I/O (even for NVMe-attached SSDs), the impact of network stack performance is negligible. However, when applications store their data on NVMM, the time scales are such that they become sensitive to both networking and storage stack performance (see Section 2).

With NVMM, a transaction could in principle become durable when the NIC DMAs data to host memory, rather than after an explicit data copy to a write-ahead log by the application. The data copy is particularly problematic in systems with NVMM, because it introduces latency and pollutes the CPU caches. Further, because of low-latency random access on durably-stored data, NVMMs

[1]We define NVMM as byte-addressable memory that is persistent, connected to the memory bus and directly addressable by the CPU.

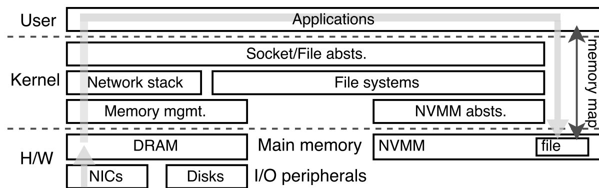


Figure 1: Today's OS organization: No integration between network stack and NVMM abstractions. A light gray arrow indicates path of data from NIC to NVMM.

could even obviate write-ahead logging depending on the primary database—creating a new opportunity of organizing primary data structures with packet buffers to which the NIC DMAs. However, since there is no integration between NVMM abstractions and the network stack, packet data and persistent data are treated separately. This leads to superfluous copying by applications (see Figure 1).

This paper proposes a fast new networking interface for persistent data on NVMMs, which we call PAcket STorE (PASTE). It allows applications to organize persistent data structures directly with network buffers, eliminating the superfluous data copies inherent in today's transactional systems. PASTE places static packet buffers into an NVMM region, statically named by a file so that they can be located across OS reboots. Therefore, applications can locate packet buffers across reboots using private metadata that *points* to arbitrary packet data. Network buffers are not recycled by the network stack until the owning application gives the stack permission. We implement PASTE as a Linux kernel module by extending the netmap [58] framework and exploiting the OS NVMM abstraction.

Our microbenchmarks that involve persistent data show PASTE outperforms a well-tuned Linux stack by up to 108 % in throughput and by up to 51 % in latency; It outperforms StackMap [68], the state-of-the art network stack by up to 43 % in throughput and 30 % in latency.

We apply PASTE to three applications: Redis, a popular persistent key-value store (up to 133 % improvement), pKVS, our custom key value store that runs over HTTP (up to 56 % improvement), and the logging component of software switch (up to 50 % improvement), in order to demonstrate that PASTE not only accelerates networked storage systems but also enables traditional networking functions to support new features.

The remainder of this paper is organized as follows: Section 2 describes background and analyzes the costs of durably storing data from an end-to-end perspective; Section 3 describes design and implementation of PASTE. Section 4 evaluates PASTE; Section 5 shows PASTE's use cases of key-value stores and software switch. Section 6 discusses PASTE's applicability and future work. Section 7 describes related work, and the paper concludes with Section 8. Appendix A provides supplemental information and advanced experiment results.

2 Motivation

To motivate the proposed reorganization of the network stack, this section briefly reviews literature around persistent data. We then perform case studies to see what happens in reality.

2.1 Background

A transactional data transfer is an essential operation in many networked storage systems, such as blob stores [7, 48, 51], key-value stores [13, 37] and databases [1, 8, 26]. A general transactional data transfer consists of following steps:

1. A client transmits data to a server.
2. The server receives packets at a NIC.
3. The NIC DMAs the packets to memory.
4. The packets are processed by the network stack.
5. A server application reads the data.
6. The server application durably stores a record of the transaction (e.g., on an SSD).
7. The server application replies to the client; the client now knows the transaction has been accepted and persisted.

Step 6 is where the largest contribution to end-to-end latency comes from (e.g., on the order of milliseconds). As discussed above, applications frequently use a write-ahead log to speed up transaction persistence instead of directly updating primary data structures, such as a B tree, which involves durably updating multiple blocks and hence many random seeks. The client-perceived transaction commit time is thus increased. Today, logs are implemented as files and are updated with the `write()` followed by `fsync()` or `fdatasync()` system calls (the latter differs only in that it does not update file metadata, so is faster).

As NVMM becomes available, applications will migrate away from using system calls and access persistent NVMM directly (a black arrow in Figure 1). It should be emphasized that NVMM DIMMs are expected within months. While they will be more expensive than NAND Flash, they are expected to be cheaper than DRAM; DRAM is what they will be replacing so adoption is expected to be rapid and wide-spread. File systems can be put on top of NVMM much as they are today for RAM disks—except the contents will survive reboots and power failures. Applications can `mmap()` files into their address space, without a buffer cache interposed, and access their data directly with unprivileged CPU load and store instructions. System calls will be far too slow in comparison, so applications will just flush data from CPU caches into NVMM, typically using the `clflush` instruction. Thus,

Memory	Measurement	Time [µs]
—	Network only (H/W, stack, HTTP)	23.32
NVMM	Network + memcpy()	25.57
	Network + memcpy()/clflush	27.17
	Network + read()/clflush	27.41
SSD	Network + memcpy()/msync()	1320
	Network + read()/msync()	1300
	Network + write()/fdatasync()	1370
	Network + write()/fsync()	3490

Table 1: End-to-end transaction latency with various persistence methods: NVMM dramatically reduces end-to-end latency and data copy comes at a significant cost.

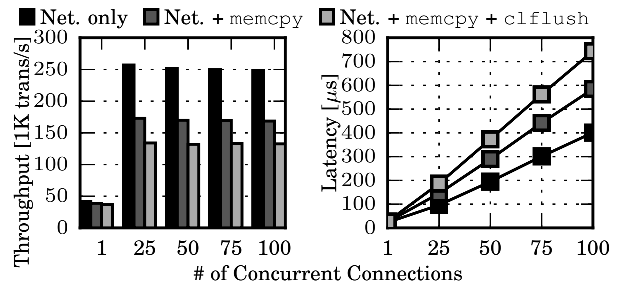


Figure 2: Throughput (left) and transaction latency (right) for concurrent requests and connections: Durably storing data still significantly reduces throughput and increases end-to-end transaction latency.

accessing storage in this new world will be two to three orders of magnitude faster than it is today.

2.2 End-to-End Transaction Latencies

To better understand the impact of logging on end-to-end latency, we wrote a simple HTTP server that implements three methods to durably log data. In all three cases, data arrives on a socket and is read by the server into a buffer. The methods of logging the data are:

(i) `write()` the buffer to a file, followed by either `fsync()` or `fdatasync()`.
(ii) `memcpy()` the buffer to a `mmap()`-ed file, followed by `msync()` for SSD or `clflush` for NVMM.
(iii) Pass the address of a `mmap()`-ed file to `read()` for use as the buffer, followed by `msync()` for SSD or `clflush` for NVMM.

The last method merges steps 5 and 6 of the general transactional data transfer (see Section 2.1), avoiding one of the two data copies that would occur otherwise. The data movement is depicted as a light gray arrow in Figure 1.

We examine two types of persistent media: a PCIe-attached SSD (Samsung 950 Pro, 256 GB) and an NVMM (HPE, 8 GB NVDIMM) attached to a DIMM slot. Both are formatted with the XFS file system that supports the Linux page-cache bypass mechanism, DAX [44] ("NVMM absts." in Figure 1). This NVMM has been available since early 2016, and costs approximately $900 for 8 GB [22].

On the client, we instrument `wrk`, a popular HTTP benchmark tool, to send 1412 B HTTP POSTs. The HTTP OK returned by the server is 127 B long. The server and client setup is described in Section 4.1.

Table 1 shows end-to-end transaction latencies that `wrk` reports. Storing data on NVMM is almost two orders of magnitude faster than on SSD. An interesting artifact is observed when comparing time to persist of the `read()/clflush` (4.09 µs) and `memcpy()/clflush` (3.85 µs) cases. Contrary to intuition, reading data directly into a `mmap()`'ed area is slower. This is because this case is more likely to require full virtual to physical address translation, so it is slower than reading into a temporary buffer; the same temporary buffer is used every time so the CPU cache has the physical addresses already. Further, `memcpy()` moves data into the log using SSE instructions. In the later discussion, we focus on the variant that uses `memcpy()`, also because it is more realistic, e.g., it applies to user-level NVMM management systems [10, 67].

The majority of costs to durably store data stem from the data copy. We ran the same measurements without flushing data after copying it (the `Network + memcpy()` row in Table 1), which exhibits 1.6 µs lower latency. This is because the access latency with our NVDIMM is almost the same as for DRAM, which is on the order of tens of nanoseconds. Since we flush 1412 B—or 23 cache lines—in a write-through manner, we expect a latency around 1 to 2 µs, which matches out measurement result rather well.

2.3 Implications

We claim that these costs of durably storing data should be regarded as high because of two reasons. First, we expect that network stacks will become faster, as demonstrated by mTCP [30], IX [4] and StackMap [68], which could further emphasize the costs of durably storing data. Second, increased latencies—which we have already observed in a single request-response transaction—amplify in realistic scenarios, because server applications typically serve a large number of clients.

Figure 2 plots throughputs and transaction latencies over concurrent requests over parallel TCP connections. We confirm these reduced throughputs and increased transaction latencies as the number of concurrent requests increases. Note that while our experiments are using a single CPU core, real deployments could serve similar or larger numbers of connections or requests on each core.

We think that these costs are unavoidable as long as we design storage and network stacks in isolation. For

example, Decibel [50] leverages DPDK for the network stack and SPDK for the storage stack, but it needs to move data between them, experiencing similar costs to those we identified above.

3 PASTE

In this section we describe PASTE, our integrated network and storage (in the form of NVMM) stack and API.

3.1 Design Principles

The persistence tier has been literally *secondary* storage, due to the costs of durably storing data on disks or SSDs, which we have quantified in the last section. NVMMs provide persistence primitives at the speed and with an interface comparable to main memory, and we envision ubiquitous deployment of them across many different applications. This includes not only storage systems, but also, for example, software switches and middleboxes for fault tolerance and fine-grained real-time monitoring, and different contexts, such as bare-metal servers in private data centres or virtual machines in the cloud [2]. For broadest deployability of PASTE, we do not rely on RDMA networking (we discuss it in Section 6.1).

PASTE is a new network programming interface for persistent data on NVMMs. There are a number of requirements for networking and persistent storage APIs, including blocking for efficiency and scalability, busy-polling for low latency and fault isolation between the stack and applications, which are benefits provided by the socket APIs today. In addition to these general requirements, PASTE achieves the following properties that concern applications:

Persisting data without a copy: This is essential, as in the previous subsection we identified that data copies to durably store data come at significant costs.

Crash recovery and consistency: Data must be randomly accessible from applications over reboots, otherwise persisting it is useless. Further, applications must be able to write and recover data *consistently*, so that they can reason about the validity of data and the metadata accompanying it (e.g., pointers and extent information) after system crash.

Avoiding unnecessary data persistence: Most network service also offer idempotent operations, some form of *read*. As seen above, persisting data is expensive so only mutable requests should be persisted.

Support for large data stores: Large capacity NVMMs are expected to store even a primary database [10, 40, 66] as opposed to fast, lower-capacity NVMMs that

[2]Virtualization and pass-through of NVMMs are active topics in both academia and industry [33, 63]

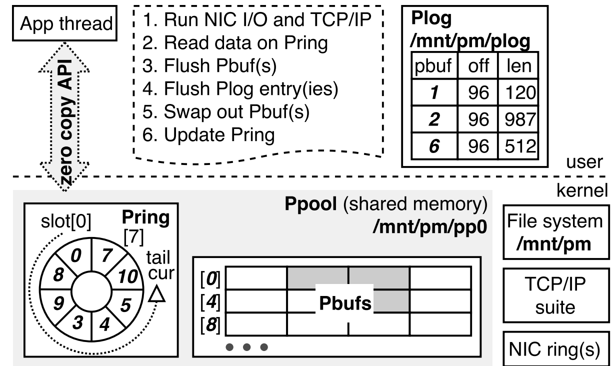


Figure 3: PASTE architecture: Packet buffers are named by a file backed by NVMM and pointed to by private application metadata.

are expected to store logs or journals [15, 36]. Therefore, we must design a networking framework that can manage large persistent data stores.

Support for network protocols: Applications must be able to use existing and new network protocols such as IP, UDP, TCP and NDP [20] for reliability, congestion control and/or compatibility with remote end systems.

Obviating serialization of application state: In general, applications have to maintain two forms of state: their in-memory state and their persistent state on disk/SSD. They are different, because DRAM is byte-addressable and exacts only minor performance penalties for random accesses whereas disks/SSDs offer awful random access performance. These conflicting characteristics lead to different interfaces that leads to different formats. The process of converting between the states is serialization [5, 6, 21, 59]. The DRAM image is the state that application actually wants, as that is the state that it actually computes with. Serialization leads to data corruptions bugs and performance problems. NVMM offers the opportunity to dispense with serialization where applications only need 1 state with NVMM, their in-memory state.

We describe details next, then show how useful these features are in building applications in the later section.

3.2 Architecture

PASTE is designed to execute NIC I/O, protocol stack processing and application logic synchronously in a batch, so called run-to-completion. This model is familiar and used by some recent systems such as Seastar [9], Sandstorm [42], IX [4] and StackMap [68], but PASTE extends it to accelerate persistence in cooperation with NVMM abstractions and provides suitable APIs for it. Figure 3 illustrates the PASTE architecture with its building blocks.

DMA into NVMM: First and foremost, we must avoid superfluous copying of data, as we have observed there are significant costs attached to such operations, which points

towards performing DMA directly into NVMM. However, this is just a starting point. We must leverage the fact that NVMM is persistent, and moving data from the NIC to NVMM introduces the opportunity to avoid later copies for persistence. Today's network stacks dynamically allocate packet buffers from a kernel pool of dynamic memory, which are thus *anonymous*. If the packet buffers themselves are to become the persistent version of data, then the first requirement is to ensure that such buffers can be found across system reboots and crashes.

Named packet buffers: To that end, packet buffers must become *named*; we use the well understood and supported *file* abstraction for this purpose. Using files is much more convenient compared to managing the physical addresses of NVMMs directly. This means that a file must be created and NVMM pages are allocated for its contents. The network stack must then statically allocate its packet buffers in the physical pages that the file contains.

Further, since the application needs to manage and access packet data with its private metadata, the network stack must provide a good representation of individual packet buffers. PASTE uses fixed-size packet buffers that are indexed by 32-bit integers, supporting several TB of data using 2 kB buffers. Hence, an application can represent data in its private data structures (`Plog` in Figure 3) by a simple tuple of a 32-bit buffer index, length and offset, which are 16-bit each in size (enough to accommodate an Ethernet jumbo frame).

PASTE initializes `Ppool` (see Figure 3) deterministically for a given memory region, which is "pinned" by a file. Hence, `Plog` and `Pbufs` are consistent over reboots.

Selective persistence: A quick digression into modern DMA is required to understand this issue. Modern NICs DMA packets into main memory *logically*, *physically* they are placed into the lowest-level CPU cache. Even if NVMM is backing the physical DMA target address, the contents of a packet are *not* persistent after a DMA. Thus, PASTE must *explicitly* push a packet to NVMM after a DMA to be certain that it is made persistent. Since this operation is costly, we do not want to perform it for every packet. Instead, the application examines a packet first while the packet is still in the CPU cache (applications are oblivious to this as the CPU manages its cache transparently). Only if the application decides that the packet needs to be persisted is the packet moved to NVMM. On the current generation of Intel CPUs, this can be done with the `clflush(opt)` machine instruction.

Applications must distinguish between request data that should be persisted and request data that may remain ephemeral. Requests that are idempotent, such as SQL `select` queries, do (by definition) not have side-effects and need not be persisted. Mutable transactions that must be logged must be persisted (e.g., inserting a row in a table). When an application identifies such a transaction, portions of the packets (bytes in the TCP stream) are pushed to NVMM and made persistent.

Lightweight ordered journaling: It is trivial to implement a *log* or *journal* [18] based on this primitive. A linked list with entries pointing to requests inside the packet buffer can be superimposed onto them. The result is a log that is temporally ordered and serves the same purpose as the journals stored on block devices today—but in a much faster fashion.

Applications can store their own log in their own NVMM-backed file (`/mnt/pm/plog` in Figure 3). In our example, the nodes of the linked list that comprise the log can be stored in said file, while the data they point to are in `/mnt/pm/pp0`.

Journaling with PASTE can be done as follows (see the pseudo-code in Figure 4, line 1–6): First, the application flushes buffer contents (lines 2–4), then durably writes a buffer extent that is a tuple of buffer index, offset and length (lines 5–6). The order ensures consistency against system failures. We analyse data integrity in detail in Appendix A.1.

Applications can perform each step over multiple buffers to journal long data. Since a tuple of buffer index, offset and length is 8 B in size, and Intel CPUs write an entire cache line of 64 B, it is possible to atomically commit up to eight entries. For longer data, the applications may put logs between additional "begin" and "end" entries like in conventional transaction logging.

Copy-on-write style free-space management: Committed buffers and logs comprise either write-ahead logs or primary data structures, such as a B+ tree. In either case, persisted buffers need to be moved out of the NIC ring (i.e., DMA target) so that buffers containing live data are not over-written. Since the `Pring` (in Figure 3) contains only *slots*—each of which includes a buffer index, length and offset (i.e., pointers to buffers)— this can be easily done by swapping buffers containing data with new empty ones outside the ring. The new empty buffers are thus attached to the slots of the ring and returned to the kernel to be eventually used as DMA targets.

The pseudo-code in lines 8–19 of Figure 4 shows the typical workflow. The application `poll()`s the receive ring for incoming requests (line 11); when it returns, it examines and generates a reply to each request (lines 12–19). Whenever it receives an update request (test at line 14) it also permanently stores the buffer containing data (lines 15–16) and then replaces it with a free one (lines 17–18) to preserve it. Buffers containing read requests are simply left in the receive ring to be reused. Responses are sent to the network in a batch at the next `poll()` (line 11).

Figure 3 illustrates an example. Initially, the `Pring` slots 0–7 pointed to buffers 0–7. Assume that the NIC has received packets on buffers 0–6 and the application has

```
1  flush_buf(buf, off, len, buf_idx, *log)
2    buf += off;
3    for (int i = 0; i < len; i += CACHE_LINE_SIZE)
4      clflush(buf + i);
5    *log = buf_idx << 32 | off << 16 | len;
6    clflush(log);
7
8  paste_eventloop(nmd, plog, plogsiz)
9    rx, tx = get_netmap_rings(nmd);
10   for (;;)
11     poll(/* on the rx ring */);
12     for each new slot s in rx
13       char *buf = get_netmap_buf(s, rx);
14       if (is_write_request(buf))
15         uint64_t *log = next_log(rx, plog, plogsiz);
16         flush_buf(buf, s->offset, s->len-s->offset,
                 s->buf_idx, log);
17         netmap_slot *extra = next_free_buf();
18         swap(s, extra); // swap buffer indices
19       write_response(tx);
20
21 main(pm_file, size, plog_file, plog_size, netmap_port)
22   fd = open(pm_file); // Ppool
23   p = mmap(fd, size);
24   netmap_pools_info *pi = p;
25   pi->memsize = size;
26   pi->buf_pool_objtotal = HOW_MANY_BUFFERS;
27   nmreq nmr = { .cmd = POOLS_CREATE, .extm = pi };
28   nm_desc *nmd = nm_open(netmap_port, &nmr);
29   plog_fd = open(plog_file);
30   plog_map = mmap(plog_fd, plog_size);
31   paste_event_loop(nmd, plog_map, plog_size);
```

Figure 4: Durably writing data and log in `flush_buf()`, event loop in `paste_eventloop()` and `Ppool` initialization in `main()`. Figure 12 in Appendix A.1 illustrates buffer state over time.

consumed them (indicated by advancing the "cur" ring pointer from slot 0 to 6) and persisted and logged buffers 1, 2 and 6 (the gray ones in the `Ppool`). The application thus has swapped the persisted buffers with free ones: in the example, these are the buffers 8, 9 and 10, respectively.

Although the `Plog` is depicted as an array of tuples for simplicity, it can be of an arbitrary form, such as a B+ tree accompanying more structured metadata (e.g., sorted by keys), as long as a single buffer extent can still be flushed atomically, thus ensuring consistency.

As it turns out, PASTE is suitable for copy-on-write operations, as opposed to in-place updates, because new data is always (DMA-)written to free space. Further, when data is stored in the primary database as with B+tree, since new data is written prior to logging, PASTE achieves *write-behind logging* [2].

If PASTE is used only for logging, primary data structures (not shown in Figure 3), such as a database table, may also be stored in NVMM, or stored on a block device (at much lower cost per byte) and updated at leisure (one of the purposes of a write-ahead log is to mask the cost of updating a primary data structure and permitting faster responses to waiting clients). Periodically, the primary data structure is updated to reflect the write-ahead log and the log contents can be safely discarded. The corresponding buffers can now be returned to the free pool.

Network protocols: A protocol suite operates directly on `Pbuf`s where the NIC or application reads or writes. On RX, the protocol suite sets only buffers whose data are *ready* (e.g., in-order TCP segments) to the application ring (`Pring`) with providing application data offset, so that the application can see useful data only. PASTE can hold non-ready data packets (e.g., out-of-order TCP segments) out of the NIC's DMA target, which are inserted to the `Pring` when the protocol suite indicates they are ready.

To exploit system call and I/O batching, a `Pring` multiplexes multiple streams (e.g., TCP connections); PASTE thus sets a file descriptor to each ring slot such that the application can distinguish them.

Protection: To be a generic programming interface, fault isolation is an essential property. Despite of the direct access to NVMM, PASTE only exposes data buffers to applications using the shared memory primitive in the kernel; it does not expose NIC registers or data structures managed by file systems or network protocols. When an application crashes, the rest of the system is unaffected.

3.3 API

In order to promote wide deployment, PASTE is designed to smoothly integrate with the netmap framework [58]; it runs in the kernel and mediates physical or virtual NIC ring(s) and applications via shared memory in which kernel- and user-owned regions are synchronized by `poll()` (blocking or non-blocking) or `ioctl()` (non-blocking) system calls. Therefore, PASTE inherits most parts of the netmap API.

Data semantics contained in ring slots depend on port types. When PASTE is used with the kernel TCP/IP implementation, each ring slot points to a buffer that contains an in-order TCP segment with offsets to payload data and a file descriptor as a ring may contain data from multiple TCP connections. On RX, buffers that belong to the same descriptor are grouped in the ring, so that the application needs to process each descriptor only once in an event loop. TX is opposite. The application puts data on `Pbuf`s pointed by available slots (or sets existing `Pbufs` to the slots, avoiding data copy) with providing a file descriptor and headroom for protocol headers to each of them. PASTE relies on regular socket APIs (e.g., `socket()`, `bind()`, `listen()`, `accept()`) for control operations. When PASTE is used for a user-level TCP/IP implementation or a middlebox that perform raw packet I/O, a ring is just a replica of the physical or virtual NIC ring where packets are placed in arrival order.

To (re)initialize the `Ppool` (which also includes all packet buffers), an application first `open()`s and `mmap()`s a file backed by NVMM (lines 22–23 in Figure 4). If this is the first time the file is opened, the application initializes a header that describes how the memory region should be

organized (lines 24–26). In any case, it prepares a netmap request pointing at the memory region (line 27) and then opens the netmap port, binding it to the region (line 28). The kernel validates the user-space virtual addresses and obtains the corresponding kernel-space virtual address, then initializes the `Ppool` using them.

Recovery: PASTE deterministically initializes `Ppool` for the given NVMM region. Therefore, after reboot, the application can restore previous buffers by simply re-initializing `Ppool` with `nm_open()`, and reason about application-specific organization of these buffers using `Plog` which can be a write-ahead log or a primary data structure like B+ tree. In Figure 4, lines 29–30, the `Plog` is also allocated on the NVMM, in a separate file.

3.4 Implementation

We implemented PASTE by heavily extending netmap, inserting approximately 4K lines of code and removing approximately 0.4K, which also contains the kernel TCP/IP support and software switch extension which we explain in Section 5.2. PASTE is a loadable kernel module and it supports Linux kernel versions of 4.6–4.12 (the latest version at the time of this writing). No modification to the main Linux kernel is needed.

We rely on the Linux NVMM kernel subsystem that provides standard NVMM abstractions [62], such as pages, namespaces [28] and DAX [44], a file system interface to access a physical NVMM device without buffer caches. Thus, applications can create their packet buffers, journals, data structures on their favorite file systems, including ones whose file operations (e.g., directory scan) are optimized for NVMMs [66].

PASTE is open source and under active development. It is available at https://github.com/luigirizzo/netmap/tree/paste with all the PASTE applications we use for experiments. We also provide some implementation details in Appendix A.4.

4 Evaluation

We begin with microbenchmarking PASTE in comparison to state-of-the art systems. We evaluate PASTE with more realistic applications in Section 5.

4.1 Hardware and Software Setup

We use two machines connected back-to-back with two Intel X540-T2 10 Gbit/s NICs and a direct attached cable. The server machine that runs PASTE has two Intel Xeon E5-2640v4 processors clocked at 2.4 GHz. For NVMM, we use an HPE 8 GB NVDIMM and format it with XFS with DAX enabled (See Figure 1 for an architecture diagram.) The client machine has an Intel Xeon E5-2690v4 processor. Both the server and the client disable "turbo boost", hyper-threading and all the C-states. They both run Linux kernel 4.11 and compiled with `gcc` version 6.3. Unless otherwise stated, we use a single CPU core at the server and the `wrk` HTTP benchmark tool with fourteen CPU cores at the client to saturate the server. Unless otherwise stated, we use busy-wait and TCP on all the systems except for Section 5.2.

	64 B	256 B	768 B	1280 B	2560 B
Net. only	22.2 $\sigma = 1.4$	22.9 $\sigma = 1.1$	23.9 $\sigma = 1.2$	24.7 $\sigma = 1.2$	28.0 $\sigma = 1.2$
Linux	21.5 $\sigma = 3.5$	22.8 $\sigma = 5.7$	25.0 $\sigma = 8.9$	27.2 $\sigma = 11.0$	33.1 $\sigma = 14.1$
StackMap	22.7 $\sigma = 3.6$	23.9 $\sigma = 5.7$	26.2 $\sigma = 8.8$	28.4 $\sigma = 10.9$	31.6 $\sigma = 10.7$
PASTE	22.6 $\sigma = 1.9$	23.2 $\sigma = 1.9$	24.7 $\sigma = 2.1$	26.4 $\sigma = 1.8$	29.4 $\sigma = 2.1$

Table 2: **Mean roundtrip latencies in µs with standard deviations σ for WAL without concurrent requests.**

4.2 Methodology

We compare PASTE against a well-tuned Linux stack and StackMap, which is the state-of-the art network stack that achieves comparable performance to user-space network stacks while using the feature-rich kernel TCP/IP implementation [68]. We refer to a version of PASTE that uses a DRAM region organized by the regular netmap as StackMap, because it resembles the architecture while details differ (e.g., PASTE does not modify the kernel, scales better to multiple cores and offers simpler API). PASTE's improvements over this StackMap thus indicates effect of reduction of data copy to persist data.

In the end, comparing Linux, StackMap and PASTE, which all use the same TCP/IP implementation, precisely exposes the stack architecture differences without any performance difference that could arise from different TCP/IP protocol implementations, which is (only) a subset of the network stack. This is important, because TCP/IP implementations have largely different features and supported protocol extensions, and adopt different software architectures to implement them.

4.3 Microbenchmarks

4.3.1 Write-Ahead Log

Write-ahead logs (WALs) are the simplest data structure to persist data in practice. We arrange the NVMM to accommodate as many WAL entries and packet buffers, which amount to roughly 3.5 million entries and buffers. The client continually generates a fixed-size HTTP POST on each experiment.

Table 2 and Figure 5 shows end-to-end throughput and mean latency of Linux, StackMap and PASTE. To see how each method compares to a networking-only performance

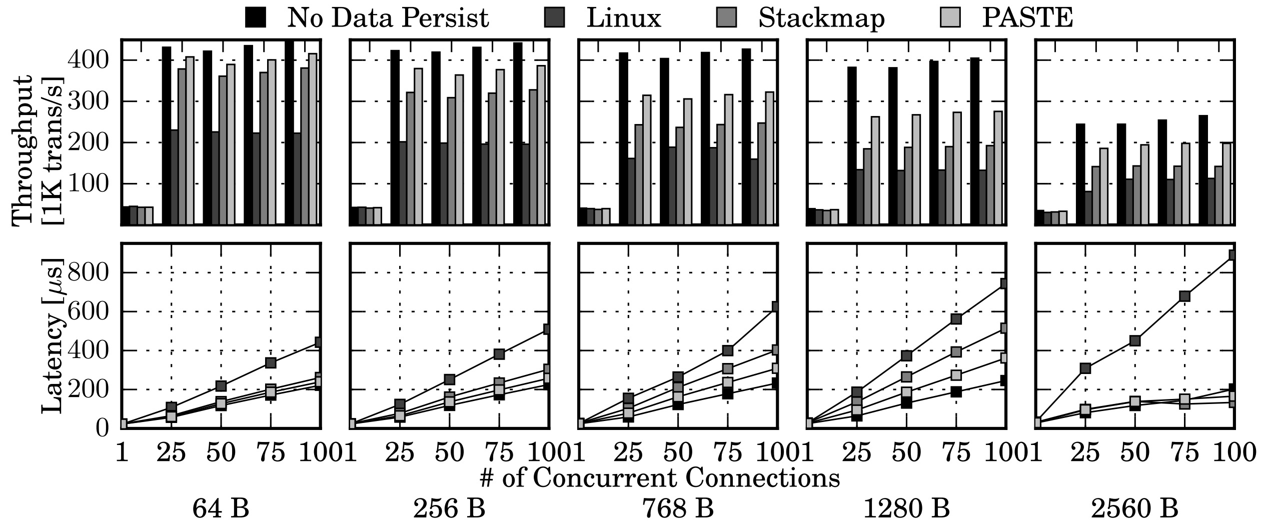


Figure 5: **Write throughput and mean latency with WAL**. Latency plots have standard deviations of at most 20 %, 43 %, 46 % and 9 % in No Data Persist, Linux, StackMap and PASTE, respectively.

baseline, we also plot PASTE without any persisting of data (it simply discards received messages and returns "HTTP OK" as if the transaction had been recorded).

For 64 to 1280 B message sizes, PASTE increases throughput by up to 108 % over Linux, and by up to 43 % over StackMap; it reduces latency by up to 51 % over Linux, and by up to 30 % over StackMap. In each method, throughput stays at almost flat on and after 25 parallel connections while latency keeps increasing. This is because the server (i.e. consumer) always has backlog requests to process. We observe improvements over StackMap by larger margins with increased message sizes. This is expected, because the cost of a data copy is small when messages are small. Latencies for the 2560 B case have different characteristics from the others, because each request now consists of two packets. In StackMap and PASTE, the lower latencies compared to that of smaller message cases is because queueing latency at the server becomes much lower due to decreased packet rates.

4.3.2 B+ Tree

Having identified that PASTE speeds up transactions to a write-ahead log, which is a temporary data structure, we now evaluate if PASTE can accelerate the case in which the data are directly stored in a primary database. We implement a B+ tree as a `Plog` on NVMM (Figure 3), a self-balanced, ordered tree which is widely used to organize primary data structures of file systems and databases. We instrument the B+ tree to store a `Plog` entry whose format is the same as in the WAL case (i.e., a tuple of buffer index, offset and length) as a value for a key. Recall from Section 3.2, the server flushes data for a network buffer prior to inserting the entry for this buffer to the B+tree.

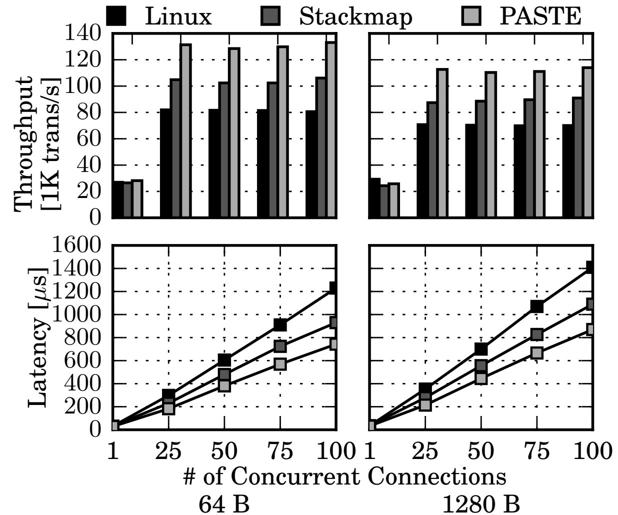


Figure 6: **Write throughput and mean latency with a B+ tree.** Latency plots have standard deviations of at most 39 %, 39 % and 7 % for in Linux, StackMap and PASTE respectively.

Figure 6 shows throughput and latency of Linux, StackMap and PASTE. The client continually transmits an "HTTP POST" message whose first 8 bytes indicate a "key" used by the B+ tree which contains 1 million random values. In the Linux and StackMap cases, the B+tree contains entire values copied from the network buffers. All the POST messages are served as insert or update. We test 64 B and 1280 B value cases.

While peak throughput is lower than in the WAL case because of tree traversal operations, we observe that PASTE improves throughput by up to 65 % over Linux

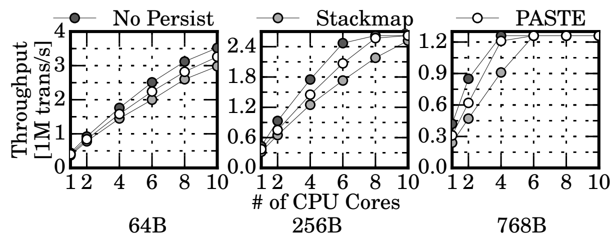


Figure 7: **WAL throughput over multiple CPU cores.**

and up to 28 % over StackMap, as well as reducing latency by up to 39 % over Linux and 23 % over StackMap.

For 64 B writes we see improvements by larger margins than WAL cases, because a B+tree is more memory intensive and the effect of reducing memory traffic is larger. We conclude that PASTE improves not only ephemeral persistent data structures, but also more complex primary data structures. In Section 5.1.1, we extend this PASTE B+ tree to a realistic key-value store that also serves read requests efficiently.

4.4 Multicore Scalability

Next, we evaluate PASTE's scalability to multiple CPU cores. We dedicate a single thread to each CPU core, and the NIC is configured to have one TX/RX pair of rings per core. Each thread independently processes a single pair of rings (`Prings` in Figure 3) with the `poll()` loop in Figure 4. It also persists data in its own `Plog` of WAL. The rings are mapped to the NIC rings, to which TCP connections are balanced by the NIC based on the connection hash value or the tuple of source-destination addresses and ports. All the rings share the same packet buffer pool or `Ppool` on NVMM. To saturate the server, we use an additional identical client machine. We use a ratio of 25 TCP connections to the number of cores.

PASTE reasonably increases throughout with additional cores. It reaches the 10 Gbit/s line rate at 8 and 6 cores with 256 and 768 B data, respectively.

5 Use Cases

In order to demonstrate how PASTE accelerates realistic applications and provides new opportunities, we have built three applications with it.

5.1 Key-Value Store

A popular use case is a key-value store (KVS) with durability support. The performance of a KVS is usually constrained by the network, because of lightweight `put/get` operations as opposed to relational databases, which require more computation to process client requests. While joint-optimization of the network stack and volatile main-memory management has been explored (see Section 7), efficiently supporting data durability requires PASTE.

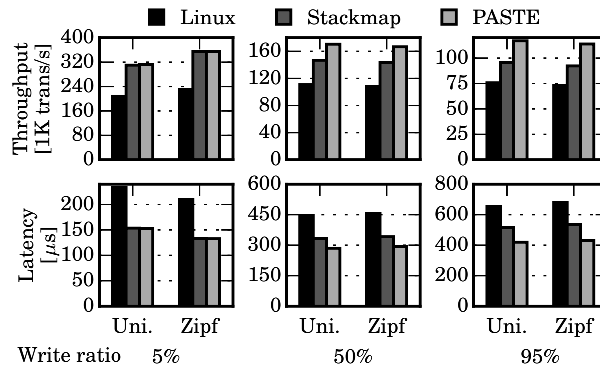


Figure 8: **Throughput and mean latency with pKVS.** Latency plots have standard deviations at most 30 % (Linux), 17 % (StackMap) and 11 % (PASTE).

5.1.1 pKVS

pKVS is our custom KVS, which builds on top of PASTE and organizes data in a B+ tree. It uses HTTP as a communication protocol, mapping "set" and "get" commands into HTTP "POST" and "GET" methods, respectively. In addition to durable zero-copy writes, which we share in Section 4.3.2, pKVS also performs opportunistic zero-copy reads. On the "set" command, the server records a pointer to a buffer slot, and on the "get" command, the server first searches for the key in the B+ tree to obtain the buffer index and extent, then further obtains the slot which contains this buffer. The result is a complete form of the previous POST message in a packet buffer. The server thus simply places this buffer into a TX ring. In order to enable zero-copy, we tailor the length of the HTTP POST and OK to be identical.

Figure 8 shows throughput and average latency on different write ratio and key skewness. 50% and 5% of write ratios with Zipfian 0.99 distribution correspond to YCSB [12] workload A (Update heavy) and B (Read mostly), respectively. We use the default YCSB parameters for the key space (1K) and size (1KB). We use 50 concurrent TCP connections.

Because of large benefit of zero-copy durable write, PASTE improves throughput and latency as the write ratio increases. PASTE increases throughput by up to 56 %, and reduces latency by up to 36 % in comparison to Linux. PASTE increases throughput by up to 23 %, and reduces latency by up to 19 % in comparison to StackMap.

5.1.2 Redis

Redis [57] is a popular named "data structure" on a network service. The server offers many services including

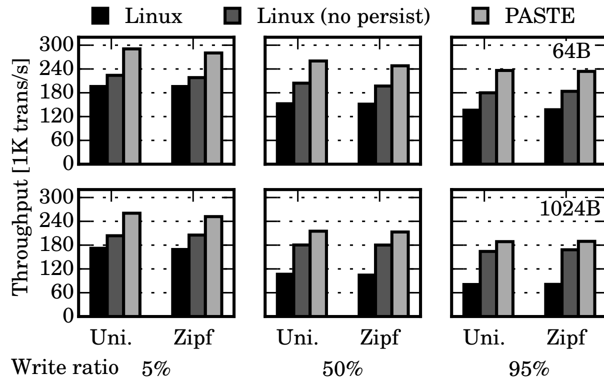


Figure 9: **Redis transaction rates.**

counters, hyperlog estimators and a key/value store to name but a few. To demonstrate both the advantages of using PASTE, and the feasibility, we extended Redis 3.2.8.

Redis uses the socket API to communicate with clients over TCP and the POSIX file I/O interface. The source code was modified to receive events from PASTE instead of `read()` and data was persisted in PASTE buffers. Around 200 lines of source code were added to the 65K line base system. We discuss porting effort in Appendix A.4.

Figure 9 plots throughput of the regular and PASTE-enabled Redis with a single CPU core over different write ratios and key distribution patterns, including two default YCSB's workloads: read-mostly (5 % writes with key skewness of Zipfian 0.99 for 1KB data) and update-heavy (50 % writes with the same distribution and data size). For comparison, we test the regular Redis with and without persisting write operations (HSETs). To be fair, PASTE does not use busy-polling in this test.

Since the data structure is a relatively lightweight hash table, peak throughputs with PASTE are similar to the WAL case in Figure 5. PASTE outperforms Redis by 43 to 133 %. Even in comparison to Redis without persistence, PASTE outperforms it by 12 to 31 %, indicating PASTE offers persistence for more than free.

5.2 Software Switch

There is a growing interest by operators in the reliability of network middleboxes whose failure impacts on many end systems [35, 56, 61]. Network Function Virtualization (NFV) has led to deploying and consolidating middleboxes on commodity servers, enabling better resource utilization and fine-grained isolation [41, 60, 69]. Fault-tolerant middlebox (FTMB) [61] allows them to recover with states after crash. It relies on input packets stored in stable storage at the virtualization backend, which are replayed after the middlebox fails [61] since the last VM snapshot.

Programmable traffic monitoring systems [14, 32, 47] could also benefit from real-time packet logging, which is now often performed by dynamically activating a mirroring port on a hardware or software switch [53, 64] to capture traffic. PASTE already maintains a file with captured packets that is a snapshot of the most recent packets decided to be recorded, which can be directly leveraged for this use-case.

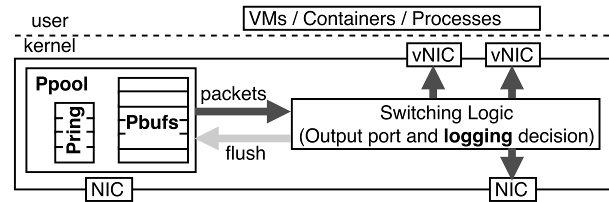


Figure 10: **mSwitch with packet logging support.**

In order to support FTMB and other applications that benefit from packet logging, we implement a logging feature in mSwitch, a fast, modular software switch that supports a large number of virtual and physical ports to serve an NFV backend [24]. Its switching logic is modular and can implement arbitrary packet processing, such as a learning bridge, L3 forwarding, the Open vSwitch datapath and a subset of P4 [54]. A module takes packets as input from the switching fabric, and returns packet action values indicating destination switch port, drop or broadcast.

mSwitch acts as an application of PASTE despite that it runs in the kernel (Figure 10). We implement a new packet action of "logging" which is used in conjunction with the existing actions by switching logic modules. When the module indicates a packet to be logged, mSwitch swaps out the buffer from the receive ring slots.

Figure 11 shows throughput with PASTE in comparison to a variant of mSwitch that implements packet logging without PASTE, by copying and flushing packets from the DRAM to the NVMM. We use the default learning bridge module that has moderate overhead consisting of two hash calculations for source and destination MAC addresses. Packets are forwarded between the two 10 Gbit/s NIC ports. For the latency measurement, we increase *burst* sizes, which indicate the number of packets arriving at the input NIC of the mSwitch at a line rate. This models a very common situation, for example, a TCP sender is allowed to send up to ten packets at once (i.e., at line rate) even at the beginning of a connection.

We confirm that PASTE improves throughput and latency by up to 50 % and 15 %, respectively. A higher latency with increasing burst sizes is due to batching in order to amortize device I/O cost (on the order of hundreds of ns) and improve packet processing locality [24]. Thus, reduction of per-packet logging costs with PASTE reduces latency by larger margins in the presence of larger numbers of packets processed within the same batch.

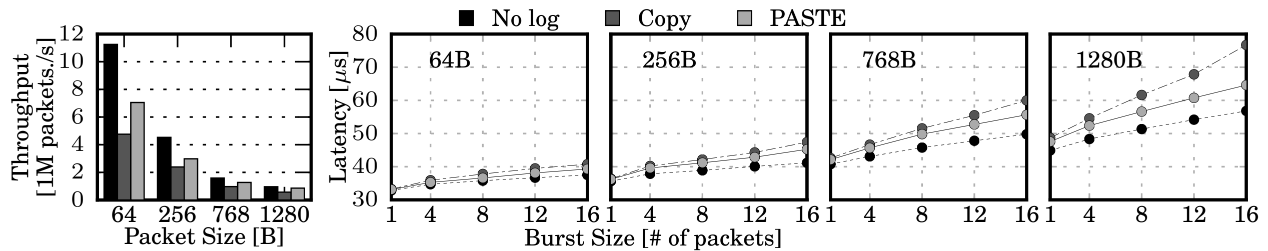


Figure 11: **mSwitch packet forwarding throughput and latency with packet logging.**

6 Discussion

In this section we discuss applicability of PASTE to various systems, and describe future work.

6.1 Kernel-Bypass Networking

PASTE relies on sharing network buffers between the network stack, the application and NVMM abstractions (e.g., files). Our implementation employs netmap [58], which executes in the kernel and thus allows PASTE to exploit a Linux file system with DAX [44] support. PASTE could also be implemented on systems such as Arrakis [55], IX [4] or Intel SPDK [27]. However, these systems have to implement NVMM abstractions by themselves.

Some user-space TCP/IP stacks, including Sandstorm [42], UTCP [25] and Warpcore [16], are easier to support with PASTE, because their buffers can be managed by netmap in the kernel. The only difference from our current design is that the TCP/IP implementation ("TCP/IP suite" in Figure 3) resides in user space.

As it turns out, our mSwitch extension in the last section follows this line and demonstrates flexibility of PASTE architecture, because it bypasses the vast majority of the kernel network stack to utilize a fast packet I/O framework.

We are starting to see RDMA deployments [19], but it requires loss-less network fabric and individual systems closely tied with low-level hardware details [15, 31, 40]. LITE [65] remedies the latter problem by kernel-level abstraction. This could help PASTE support RDMA, providing a higher-level interface to integrate with NVMMs and to be transparent to TCP/IP networking.

Our latency target range (e.g., 22.6–29.4 µs without parallel requests, see Section 4.3.1) is close to that of RDMA. [19] reports RTTs of several tens of µs over RDMA network fabric likely in the absence of queueing between network and application formed by parallel requests. In addition, recent work also reports comparable latency is achievable over lossy Ethernet fabric without RDMA [20].

6.2 NVMM Access Latency

Many different NVMM technologies are anticipated, with I/O latencies from tens to thousands of nanoseconds [29, 45]. As explained in Section 3.2, idempotent requests are only DMA'ed to the CPU cache, thus the performance of idempotent requests is decoupled from the characteristics of the underlying storage. PASTE would only be exposed to the underlying media for mutable transactions. This is unavoidable and inherent in storing data. We anticipate that PASTE would be suitable for many different NVMM types, but the performance of PASTE transactions would depend on the performance of any underlying media.

The latest generation of Intel CPUs have a faster cache-line flush instruction (`clflushopt`), which also works in a write-back fashion. Therefore, we will be able to *overlap* NVMM access latencies with subsequent processing; and this can be done across multiple requests processed in the same batch, i.e., in the same `poll()` loop (see Figure 4). We can guarantee that all the flushes are done at the time of triggering transmission (i.e., `poll()` using `mfence` instructions. In Appendix A.2 we describe details, and quantify effects with some experiments.

6.3 Generality

PASTE works as a fast, scalable network stack in the absence of NVMM, because it still exploits run-to-completion, system call and I/O batching, and zero copy between the NIC and application. The netmap API that PASTE is based on has been widely used in packet I/O applications and has proven its flexibility and ease of use. Further, PASTE can be used without modifying the kernel, and offers protection provided by the socket API. Therefore, we believe PASTE is a suitable basis which achieves high performance in general and makes applications ready to efficiently support NVMMs.

6.4 Limitations and Future Work

Space utilization: PASTE relies on fixed-size packet buffers for indexing. For better space utilization, we would combine copies for small data depending on workloads.

Multiple applications: Since the application needs to have direct access to NIC's DMA target, isolating multiple applications requires partitioning it. This could be done using Flow Director on multiple NIC queues and Smart NICs (based on more flexible policy).

NVMM wear: DMA-writes would increase wear on NVMMs, while it could be mitigated by DDIO. We leave analysing this effect in future work.

7 Related Work

Previous work discussed PASTE's concept and strategy, and made minimalistic implementation and experiments using an emulated NVMM device [23]; This paper completes our design and implementation, as well as extensive evaluation and case study of applying to applications.

Special-Purpose Network Stacks: Specializing a network stack by leveraging application knowledge has been proposed several times [17, 38, 42, 43]. PASTE takes a different approach with a network stack that is general enough to support different classes of applications.

Enhanced Network Stacks: IX [4], mTCP [30], Fastsocket [39] and StackMap [68] are fast network stacks. Since they do not assume DMA on NVMM, they do not address the overheads of durably storing data, as described in Section 2, and shown how PASTE improves these approaches in Section 4. We have discussed RDMA approaches in Section 6.1.

General-Purpose Networking API: On the transmit path, the `sendfile()` system call enables applications to directly transmit data from in-kernel buffer caches or NVMMs. However, doing the opposite (i.e., directly receiving data into the buffer cache or NVMM) is not trivial, because applications need to examine the data to make processing decisions. PASTE enables this by the persistent, named packet buffers and their abstraction.

New NIC Interfaces: FlexNIC [34] provides rich abstractions of NIC features, such as scheduling, offloading and classification. These works are complementary to PASTE. For example, they could support isolating multiple applications on the same NIC.

NVMM-Aware Persistent Data Store: There exists a large body of work on efficiently managing data in NVMM. They tend to examine the problem from the perspective of the storage system in isolation. There is little consideration of data arrival from a network or the requirements of application logic. The POSIX API is often their starting point. They can be generally classed into block-oriented storage systems, such as file systems and virtual disks [3, 26, 36, 46], or byte-oriented file systems [11, 66, 67], that is the latter's metadata is byte-oriented, but they still export a POSIX interface. Some NVMM programming systems [10, 59] are designed from the application's perspective. `malloc()` manages NVM and transactions on nodes in linked lists or binary trees are supported instead of file blocks. This approach fits the PASTE approach, the native representation of data is a first class citizen, not serialized data.

None offer a coherent and integrated life-cycle for work arriving from a network that needs to be persisted. For example, NVWAL [36] employs byte-granularity differential logging to reduce the amount of data to log, resulting in a reduced number of memory copies and cache-line flushes. There is no support efficiently storing data in its final resting place. However, PASTE allows applications to log only a packet buffer index, offset and length (8 B in total) per packet, which is much smaller than the differential data set.

DRAM-based Data Store: There is a large body of work which co-design in-memory data store with network stacks. For example, MICA [38] is an extremely fast, scalable key-value store that bypasses the most of the network stack and relies on UDP to tightly map key-value data structures and packets. RAMCloud [52] is a distributed key/value store that avoids the penalty of persisting to media by replicating to multiple physical machines. PASTE could help such systems to support persistence, because it creates and names packet buffers on NVMM and allows applications to organize them with zero-copy, protocol-independent networking API. On NVMM, applications can use the same cache invalidation mechanisms with this class of work.

8 Conclusion

NVMM is not just a faster, more exotic, storage medium. It is a fundamental change in the memory hierarchy. Its introduction and adoption will change the way we design and evaluate systems. The artificial sequestering of networking stacks, storage stacks and application logic will be infeasible with such hardware. The Network File System (NFS) was feasible because the network was not the bottleneck, the bottleneck was the disk. Commodity NVMM is pushing the stack out of the kernel and into user-land. Network stacks are following. As the application, network/storage stacks will be operating in the same address space they need to be co-designed for true efficiency.

In this paper we have quantified the cost of a network service offering reliable storage services under a variety of scenarios. We have shown that by tightly integrating the network stack, application logic and the storage stack large performance improvements can be realized. PASTE is a system that safely permits applications to be built, and back-ported, to gain these performance improvements. It does this while retaining the isolation, protection and software maintenance advantages of modern monolithic kernel stacks. We verified our system by implementing and evaluating PASTE then writing and back porting real applications to use it. We then showed PASTE-based applications' performance to be superior to the state of the art.

Acknowledgments

We thank our shepherd, Rachit Agarwal, and anonymous HotNets and NSDI reviewers for their feedback. We are thankful to Luigi Rizzo for his insightful comments. We would also like to thank PJ Waskievicz for his assistance during experiments. This paper has received funding from the European Union's Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866 ("SSICLOPS"). It reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] J. Arulraj, A. Pavlo, and S. R. Dulloor. "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems". *Proc. ACM SIGMOD/PODS*. 2015.

[2] J. Arulraj, M. Perron, and A. Pavlo. "Write-behind Logging". *Proc. VLDB Endow.* Nov. 2016.

[3] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. "Operating System Implications of Fast, Cheap, Non-Volatile Memory". *Proc. ACM HotOS*. 2011.

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

[5] A. Birrell, M. Jones, and E. Wobber. "A Simple and Efficient Implementation of a Small Database". *Proc. ACM SOSP*. 1987.

[6] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. "Orleans: cloud computing for everyone". *Proc. ACM SoCC*. ACM. 2011.

[7] B. Calder et al. "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency". *Proc. ACM SOSP*. 2011.

[8] A. Chatzistergiou, M. Cintra, and S. D. Viglas. "REWIND: Recovery Write-ahead system for In-memory Non-volatile Data-structures". *Proc. VLDB Endow.* Jan. 2015.

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

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. "NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories". *Proc. ACM ASPLOS*. 2011.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. "Better I/O Through Byte-addressable, Persistent Memory". *Proc. ACM SOSP*. 2009.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking cloud serving systems with YCSB". *Proc. ACM SoCC*. ACM. 2010.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store". *Proc. ACM SOSP*. 2007.

[14] S. Donovan and N. Feamster. "Intentional Network Monitoring: Finding the Needle Without Capturing the Haystack". *Proc. ACM HotNets*. 2014.

[15] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. "FaRM: Fast Remote Memory". *Proc. USENIX NSDI*. 2014.

[16] L. Eggert. *warpcore*. Jan. 2017.

[17] G. Ganger, D. Engler, M. Kaashoek, H. Briceno, R. Hunt, and T. Pinckney. "Fast and flexible application-level networking on exokernel systems". *ACM ToCS*, 2002.

[18] G. R. Ganger and Y. N. Patt. "Metadata Update Performance in File Systems". *Proc. USENIX OSDI*. 1994.

[19] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. "RDMA over Commodity Ethernet at Scale". *Proc. ACM SIGCOMM*. 2016.

[20] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. "Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance". *Proc. ACM SIGCOMM*. 2017.

[21] M. P. Herlihy and B. Liskov. "A Value Transmission Method for Abstract Data Types". *ACM Trans. Program. Lang. Syst.* Oct. 1982.

[22] Hewlett Packard Enterprise. *Turbo-charge performance with HPE Persistent Memory*. https://www.hpe.com/h20195/v2/GetDocument.aspx?docname=4AA6-4771ENW&doctype=data%20sheet&doclang=EN_US. Mar. 2016.

[23] M. Honda, L. Eggert, and D. Santry. "Paste: Network stacks must integrate with nvmm abstractions". *Proc. ACM HotNets*. ACM. 2016.

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

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

[26] J. Huang, K. Schwan, and M. K. Qureshi. “NVRAM-aware Logging in Transaction Systems”. *Proc. VLDB Endow.* Dec. 2014.

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

[28] Intel Corporation. *NVDIMM Namespace Specification*. http://pmem.io/documents/NVDIMM_Namespace_Spec.pdf.

[29] Jeff Chang. *NVDIMM-N Cookbook: A Soup-to-Nuts Primer on Using NVDIMM-Ns to Improve Your Storage Performance*. http://www.snia.org/sites/default/files/SDC15_presentations/persistant_mem/JeffChang-ArthurSainio_NVDIMM_Cookbook.pdf. Sep. 2015.

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

[31] A. Kalia, M. Kaminsky, and D. G. Andersen. “Design Guidelines for High Performance RDMA Systems”. *Proc. USENIX ATC*. 2016.

[32] A. Kangarlou, S. Shete, and J. D. Strunk. “Chronicle: Capture and Analysis of NFS Workloads at Line Rate”. *Proc. USENIX FAST*. 2015.

[33] S. Kannan, A. Gavrilovska, and K. Schwan. “pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage”. *Proc. ACM EuroSys*. 2016.

[34] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. “High Performance Packet Processing with FlexNIC”. *Proc. ACM ASPLOS*. 2016.

[35] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. “Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr.” *Proc. USENIX NSDI*. 2016.

[36] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. “NVWAL: Exploiting NVRAM in Write-Ahead Logging”. *Proc. ACM ASPLOS*. 2016.

[37] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. “SILT: A Memory-efficient, High-performance Key-value Store”. *Proc. ACM SOSP*. 2011.

[38] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. “MICA: A Holistic Approach to Fast In-Memory Key-Value Storage”. *Proc. USENIX NSDI*. 2014.

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

[40] Y. Lu, J. Shu, Y. Chen, and T. Li. “Octopus: an RDMA-enabled Distributed Persistent Memory File System”. *Proc. USENIX ATC*. 2017.

[41] V. Maffione, L. Rizzo, and G. Lettieri. “Flexible virtual machine networking using netmap passthrough”. *Proc. IEEE LANMAN*. IEEE. 2016.

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

[43] I. Marinos, R. N. Watson, M. Handley, and R. R. Stewart. “Disk|Crypt|Net: Rethinking the Stack for High-performance Video Streaming”. *Proc. ACM SIGCOMM*. 2017.

[44] Matthew Wilcox. *DAX: Page cache bypass for filesystems on memory storage*. https://lwn.net/Articles/618064/. Oct. 2014.

[45] Micron. *Breakthrough Nonvolatile Memory Technology*. https://www.micron.com/about/emerging-technologies/3d-xpoint-technology.

[46] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. “Operating System Support for NVM+DRAM Hybrid Main Memory”. *Proc. ACM HotOS*. 2009.

[47] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. “Trumpet: Timely and Precise Triggers in Data Centers”. *Proc. ACM SIGCOMM*. 2016.

[48] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. “f4: Facebook’s Warm BLOB Storage System”. *Proc. USENIX OSDI*. 2014.

[49] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield. “Non-volatile Storage”. *Commun. ACM*, Dec. 2015.

[50] M. Nanavati, J. Wires, and A. Warfield. “Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage”. *Proc. USENIX NSDI*. 2017.

[51] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. “Flat Datacenter Storage”. *Proc. USENIX OSDI*. 2012.

[52] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. “Fast Crash Recovery in RAMCloud”. *Proc. ACM SOSP*. 2011.

[53] Open vSwitch. *Basic Configuration*. http://docs.openvswitch.org/en/latest/faq/configuration/.

[54] P4 Consortium. *P4 Language Specification*. https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

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

[56] R. Potharaju and N. Jain. "Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters". *Proc. ACM IMC*. 2013.

[57] Redis. *Official Redis Website*. https://redis.io/.

[58] L. Rizzo. "netmap: A Novel Framework for Fast Packet I/O". *Proc. USENIX ATC*. 2012.

[59] D. Santry and K. Voruganti. "Violet: A Storage Stack for IOPS/Capacity Bifurcated Storage Environments". *Proc. USENIX ATC*. 2014.

[60] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. "Design and Implementation of a Consolidated Middlebox Architecture". *Proc. USENIX NSDI*. 2012.

[61] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. "Rollback-Recovery for Middleboxes". *Proc. ACM SIGCOMM*. 2015.

[62] SNIA Technical Position. *NVM Programming Model Version 1.2*. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf. 2017.

[63] Stefan Hajnoczi. *Using NVDIMM under KVM*. https://vmsplice.net/~stefan/stefanha-fosdem-2017.pdf.

[64] O. Tilmans, T. Bühler, S. Vissicchio, and L. Vanbever. "Mille-Feuille: Putting ISP Traffic Under the Scalpel". *Proc. ACM HotNets*. 2016.

[65] S.-Y. Tsai and Y. Zhang. "LITE Kernel RDMA Support for Datacenter Applications". *Proc. ACM SOSP*. 2017.

[66] J. Xu and S. Swanson. "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories". *Proc. USENIX FAST*. 2016.

[67] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. "NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems". *Proc. USENIX FAST*. 2015.

[68] K. Yasukata, M. Honda, D. Santry, and L. Eggert. "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs". *Proc. USENIX ATC*. 2016.

[69] W. Zhang, J. Hwang, S. Rajagopalan, K. Ramakrishnan, and T. Wood. "Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization". *Proc. ACM CoNEXT*. 2016.

A Appendix

A.1 Consistency Analysis

Figure 12 illustrates data states over a single network event loop cycle. If the system crashes before metadata (`Plog` entries) are flushed, extant packet buffers are simply overwritten by the next packets following reboot. If it crashes after the metadata has been written but before the corresponding buffers are swapped out of `Pring`, the application must do so right after re-initializing `Ppool`, before starting network I/O. Note that the application should not have updated the ring's pointer (`cur` in Figure 3) before swapping out the buffers.

The application can identify the buffers to be swapped out by reading its `Plog`. There is no atomicity semantics on buffer swapping, so the application should read `Plog` and ensure that the necessary buffers are in the intended place in either on or outside the `Pring`. The application may also leverage the ring pointer to identify buffers that have been swapped out, because the ring pointer can be updated atomically.

If the system crashes after buffers have been swapped out, buffers are consistent. However, some data sent after that, such as response messages might have been lost before being dispatched to the network. It is the responsibility of the application-level protocol to address or tolerate duplicate responses.

A.2 Overlap Flushes for an Event Loop

In Section 6.2, we have introduced a technique that overlaps flushes and other processing in a network event loop that processes multiple requests, by leveraging `clflushopt` and `mfence`. We set out to test this method using a server equipped with an Intel Xeon Silver 4110 CPU clocked at 2.1 Ghz that supports this instruction. Unfortunately, since this machine does not support our NVMM device, we emulate NVMM using a reserved region of DRAM as prior work does [26, 40].

Figure 13a shows WAL throughput and mean latency. The overlap improves throughput by up to 47 % and latency by up to 32 % in StackMap that copies data. It improves throughput by up to 72 % and latency by up to 42 % in PASTE. PASTE with the overlap improves throughput by up to 54 % and latency by up to 35 % in comparison to StackMap.

Figure 13b shows the B+tree case. The overlap improves throughput by up to 93 % in StackMap, and up to 133 % in PASTE; PASTE with the overlap improves throughput by up to 59 % in comparison to StackMap.

We observe higher throughputs in comparison to equivalent results in Section 4, although the CPU clock is lower in this server and the real NVMM used in the other server achieves the same speed with DRAM "in theory". This is perhaps because of higher memory clock frequency of this server (2600 Mhz, as opposed to 2133 Mhz in that section), and the newer CPU generation.

A.3 Effect of High NVMM Access Latency

Using the aforementioned overlap technique, we examine the effect of NVMMs with higher access latency. Since `clflushopts` are asynchronous, we expect that higher NVMM access latency delays `mfence` to return. We thus insert artificial `sleep()` before `mfence`, and measure impact on overall throughput.

Figure 14 plots results, and they match our expectation. Emulated latency decreases throughput by larger margin as the number of parallel connections or requests decreases, because the NVMM access latency is amortised over the number of requests processed in the same network event loop.

In summary, also including the previous subsection, we conclude that the overlap technique significantly improves performance, and could mask high NVMM access latency. However, there is also a caveat. This method could increase the complexity of consistency guarantees. It certainly avoids compromising data after acknowledging to the client. However, when the system crashes before doing so, the system does not have any guarantee of the correctness of receiving data to be written. We can mitigate this risk by either flushing metadata, or designing the application-level protocol to tolerate duplicate writes where the server thinks the data is written but the client does not so thus precipitating resends of the previous write. We leave the analysis of these approaches for future work.

A.4 Implementation Note

In the OS kernel, network protocols are usually implemented using OS-specific packet representation structures (`sk_buff` in Linux, `mbuf` in *BSD). They typically contain metadata and one or more *pointers* to buffers that contain actual packet data, allowing them to point `Pbufs`. Once an RX buffer is passed to the TCP/IP implementation (`netif_receive_skb()` in Linux, `ifp->if_input()` in FreeBSD), in order to identify whether it is ready to be set to a `Pring` (e.g., in-order TCP segment), we exploit a callback that is invoked on data enqueued to a socket buffer (`sk_data_ready()` in Linux and `sb_upcall()` in FreeBSD). The socket structure also has interfaces for kernel subsystems (e.g., iSCSI) similar to user-space socket APIs. But in the kernel they also provide zero-copy APIs (`kernel_sendpage()` in Linux and `sosend()` in FreeBSD), which allow PASTE to pass data that reside in `Pbufs` to the TCP/IP implementation on the TX path.

Further, the OS kernels provide an interface (`get_user_pages()` in Linux and vm_map_*() family

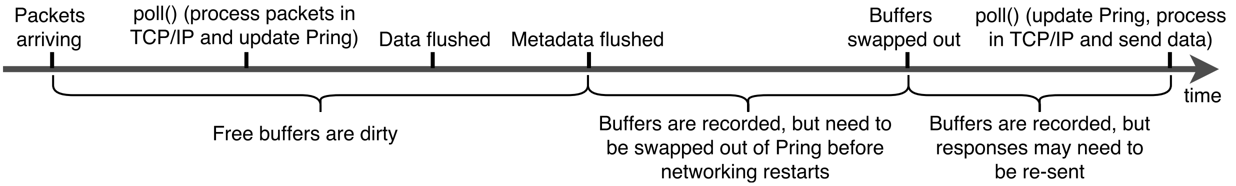


Figure 12: **Buffer state over a networking event cycle.**

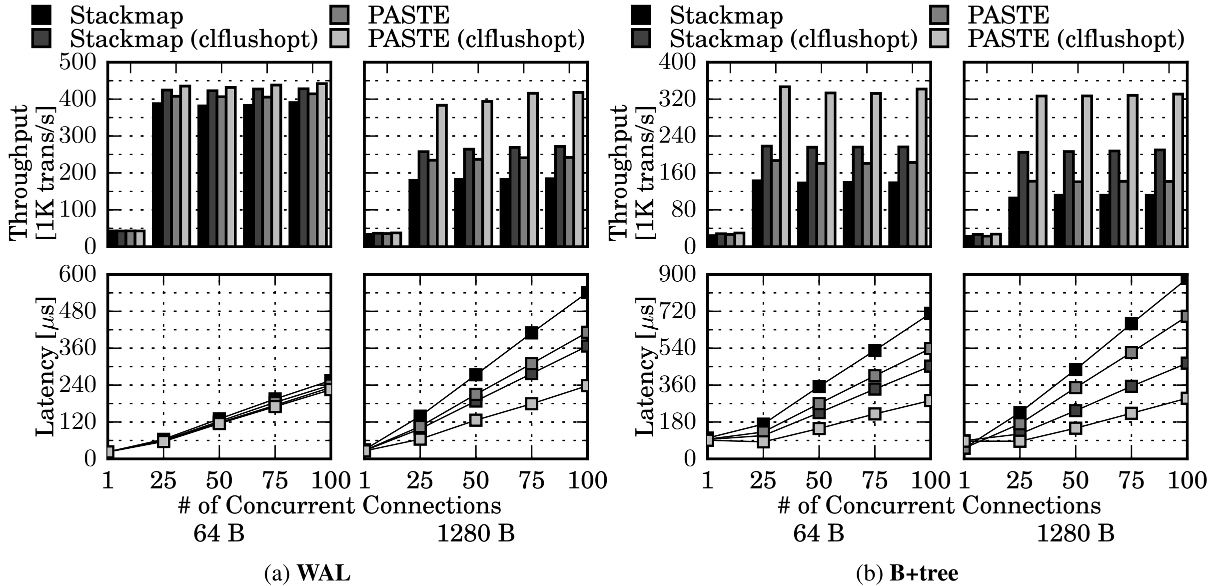


Figure 13: **Throughput and mean latency with `clflushopt`.**

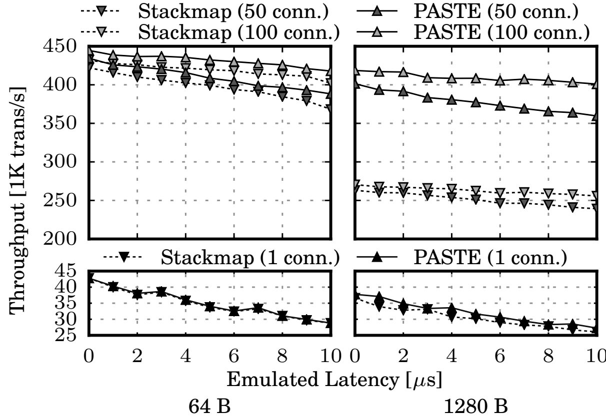


Figure 14: **WAL throughput over emulated NVMM access latency.**

in FreeBSD) to obtain kernel-space virtual addresses from the user-space ones `mmap()`ed to the file (e.g., `Ppool`). Therefore, PASTE can be implemented without modifying the OS kernel, using its *good* parts, such as protection mechanisms inherited from the netmap framework, NVMM abstractions, file systems and extensive network protocol implementations.

FreeBSD support is our ongoing effort. It appears possible once the basic NVMM programming model [62] is supported, because the netmap framework is already there.

The porting effort of existing applications to use PASTE is medium, according to our experience with Redis (where the majority of the effort was to understand how Redis works, a burden that the maintainers would not have to bear). We have a library implemented as a header file to initialize and run an event loop in Figure 4. This library implements two callbacks to be registered by an application: one invoked at `accept()` and the other invoked on every RX packet buffer when traversing the ring (line 12 in the figure). In addition to rearranging Redis to use these features, we extended a function that parses and identifies a write request to flush and swap out the buffer, using the same procedure with `flush_buf()` in the figure.

In order to ease porting existing applications and writing new ones, we plan to extend `libuv`, a popular event-driven networking library, to support PASTE.

NetChain: Scale-Free Sub-RTT Coordination

Xin Jin[1], Xiaozhou Li[2], Haoyu Zhang[3], Nate Foster[2,4],
Jeongkeun Lee[2], Robert Soulé[2,5], Changhoon Kim[2], Ion Stoica[6]

[1]*Johns Hopkins University,* [2]*Barefoot Networks,* [3]*Princeton University,*
[4]*Cornell University,* [5]*Università della Svizzera italiana,* [6] *UC Berkeley*

Abstract

Coordination services are a fundamental building block of modern cloud systems, providing critical functionalities like configuration management and distributed locking. The major challenge is to achieve low latency and high throughput while providing strong consistency and fault-tolerance. Traditional server-based solutions require multiple round-trip times (RTTs) to process a query. This paper presents NetChain, a new approach that provides scale-free sub-RTT coordination in datacenters. NetChain exploits recent advances in programmable switches to store data and process queries entirely in the network data plane. This eliminates the query processing at coordination servers and cuts the end-to-end latency to as little as half of an RTT—clients only experience processing delay from their own software stack plus network delay, which in a datacenter setting is typically much smaller. We design new protocols and algorithms based on chain replication to guarantee strong consistency and to efficiently handle switch failures. We implement a prototype with four Barefoot Tofino switches and four commodity servers. Evaluation results show that compared to traditional server-based solutions like ZooKeeper, our prototype provides orders of magnitude higher throughput and lower latency, and handles failures gracefully.

1 Introduction

Coordination services (e.g., Chubby [1], ZooKeeper [2] and etcd [3]) are a fundamental building block of modern cloud systems. They are used to synchronize access to shared resources in a distributed system, providing critical functionalities such as configuration management, group membership, distributed locking, and barriers. These various forms of coordination are typically implemented on top of a *key-value store* that is replicated with a consensus protocol such as Paxos [4] for *strong consistency* and *fault-tolerance*.

High-throughput and low-latency coordination is essential to support interactive and real-time distributed applications, such as fast distributed transactions and sub-second data analytics tasks. State-of-the-art in-memory transaction processing systems such as FaRM [5] and DrTM [6], which can process hundreds of millions of transactions per second with a latency of tens of microseconds, crucially depend on fast distributed locking to mediate concurrent access to data partitioned in multiple servers. Unfortunately, acquiring locks becomes a significant bottleneck which severely limits the transaction throughput [7]. This is because servers have to spend their resources on (*i*) processing locking requests and (*ii*) aborting transactions that cannot acquire all locks under high-contention workloads, which can be otherwise used to execute and commit transactions. This is one of the main factors that led to relaxing consistency semantics in many recent large-scale distributed systems [8, 9], and the recent efforts to avoid coordination by leveraging application semantics [10, 11]. While these systems are successful in achieving high throughput, unfortunately, they restrict the programming model and complicate the application development. A fast coordination service would enable high transaction throughput without any of these compromises.

Today's server-based solutions require multiple end-to-end round-trip times (RTTs) to process a query [1, 2, 3]: a client sends a request to coordination servers; the coordination servers execute a consensus protocol, which can take several RTTs; the coordination servers send a reply back to the client. Because datacenter switches provide sub-microsecond per-packet processing delay, the query latency is dominated by host delay which is tens to hundreds of microseconds for highly-optimized implementations [12]. Furthermore, as consensus protocols do not involve sophisticated computations, the workload is communication-heavy and the throughput is bottlenecked by the server IO. While state-of-the-art solutions such as NetBricks [12] can boost a server to process tens of millions of packets per second, it is still orders of magnitude slower than a switch.

We present NetChain, a new approach that leverages the power and flexibility of new-generation programmable switches to provide scale-free sub-RTT coordination. In contrast to server-based solutions, NetChain is an in-network solution that stores data and processes queries entirely within the network data plane. We stress that NetChain is not intended to provide a new theoretical

answer to the consensus problem, but rather to provide a systems solution to the problem. Sub-RTT implies that NetChain is able to provide coordination within the network, and thus reduces the query latency to as little as half of an RTT. Clients only experience processing delays caused by their own software stack plus a relatively small network delay. Additionally, as merchant switch ASICs [13, 14] can process several billion packets per second (bpps), NetChain achieves orders of magnitude higher throughput, and scales out by partitioning data across multiple switches, which we refer to as *scale-free*.

The major challenge we address in this paper is building *a strongly-consistent, fault-tolerant, in-network key-value store* within the functionality and resource limit of the switch data plane. There are three aspects of our approach to address this challenge. (*i*) We leverage the switch on-chip memory to store key-value items, and process both *read* and *write* queries directly in the data plane. (*ii*) We design a variant protocol of *chain replication* [15] to ensure strong consistency of the key-value store. The protocol includes a routing protocol inspired by segment routing to correctly route queries to switches according to the chain structure, and an ordering protocol based on sequence numbers to handle out-of-order packet delivery. (*iii*) We design an algorithm for fast failover that leverages network topologies to quickly resume a chain's operation with remaining nodes, and an algorithm for failure recovery that restores the partial chains to the original fault-tolerance level and leverages virtual groups to minimize disruptions.

NetChain is incrementally deployable. The NetChain protocol is compatible with existing routing protocols and services already in the network. NetChain only needs to be deployed on a few switches to be effective, and its throughput and storage capacity can be expanded by adding more switches. In summary, we make the following contributions.

- We design NetChain, a new strongly-consistent, fault-tolerant, in-network key-value store that exploits new-generation programmable switches to provide scale-free sub-RTT coordination.
- We design protocols and algorithms tailored for programmable switches to ensure strong consistency (§4) and fault-tolerance (§5) of the key-value store.
- We implement a NetChain prototype with Barefoot Tofino switches and commodity servers (§7). Evaluation results show that compared to traditional server-based solutions like ZooKeeper, NetChain provides orders of magnitude higher throughput and lower latency, and handles failures gracefully (§8).

Recently there has been an uptake in leveraging programmable switches to improve distributed systems. NetChain builds on ideas from two pioneering works in particular: NetPaxos [16, 17] and NetCache [18]. NetPaxos uses programmable switches to accelerate consensus protocols, but it does not offer a replicated key-value service, and the performance is bounded by the overhead of application-level replication on servers. NetCache uses programmable switches to build a load-balancing cache for key-value stores, but the cache is not replicated and involves servers for processing write queries. In comparison, NetChain builds a strongly-consistent, fault-tolerant key-value store in the network data plane. We discuss NetChain's limitations (e.g., storage size) and future work in §6, and related work in detail in §9.

	Server	**Switch**
Example	NetBricks [12]	Tofino [13]
Packets per sec.	30 million	a few billion
Bandwidth	10-100 Gbps	6.5 Tbps
Processing delay	10-100 μs	$< 1\mu s$

Table 1: Comparison of packet processing capabilities.

2 Background and Motivation

2.1 Why a Network-Based Approach?

A network data-plane-based approach offers significant advantages on latency and throughput over traditional server-based solutions. Moreover, such an approach is made possible by the emerging programmable switches such as Barefoot Tofino [13] and Cavium XPliant [19].

Eliminating coordination latency overhead. The message flow for a coordination query is:

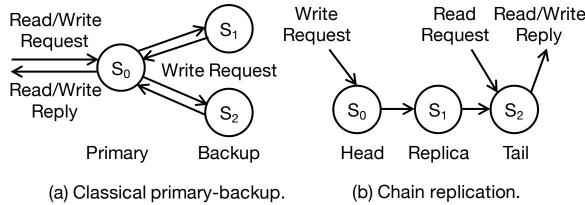
client → coordination servers → client.

Coordination servers execute a consensus protocol such as Paxos [4] to ensure consistency, which itself can take multiple RTTs. NOPaxos [20] uses the network to order queries and eliminates the message exchanges between coordination servers. It reduces the latency to two message delays or one RTT, which is the *lower bound* for server-based solutions. As shown in Table 1, since datacenter switches only incur sub-microsecond delay, the host delay dominates the query latency. By moving coordination to the network, the message flow becomes:

client→ network switches → client.

Because the network time is negligible compared to host delay, a network-based solution is able to cut the query latency to one message delay or sub-RTT, which is better than the lower-bound of server-based solutions. Note that the sub-RTT latency is not a new theoretical answer to the consensus problem, but rather a systems solution that eliminates the overhead on coordination servers.

Improving throughput. The workload of coordination systems is communication-heavy, rather than computation-heavy. While varying in their details, consensus protocols typically involve multiple rounds of message exchanges, and in each round the nodes examine their messages and perform simple data comparisons



(a) Classical primary-backup. (b) Chain replication.

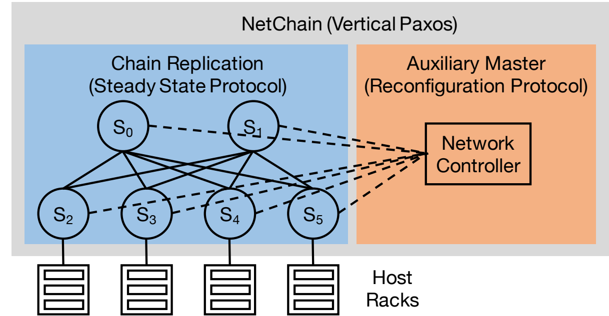
Figure 1: Primary-backup and chain replication.

and updates. The throughput is determined by how fast the nodes can process messages. Switches are specifically designed and deeply optimized for packet processing and switching. They provide orders of magnitude higher throughput than highly-optimized servers (Table 1). Alternative designs like offloading to NICs and leveraging specialized chips (FPGAs, NPUs or ASICs) either do not provide comparable performance to switch ASICs or are not immediately deployable due to cost and deployment complexities.

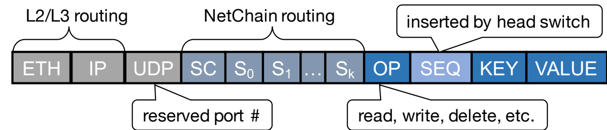
2.2 Why Chain Replication?

Given the benefits, the next question is how to build a replicated key-value store with programmable switches. NetCache [18] has shown how to leverage the switch on-chip memory to build a key-value store on *one* switch. Conceivably, we can use the key-value component of NetCache and *replicate* the key-value store on *multiple* switches. But the challenge in doing so would be how to ensure strong consistency and fault-tolerance.

Vertical Paxos. We choose to realize Vertical Paxos [21] in the network to address this challenge. Vertical Paxos is a variant of the Paxos algorithm family. It divides a consensus protocol into two parts, i.e., *a steady state protocol* and *a reconfiguration protocol*. The division of labor makes it a perfect fit for a network implementation, because *the two parts can be naturally mapped to the network data and control planes*. (*i*) The steady state protocol is typically a primary-backup (PB) protocol, which handles read and write queries and ensures strong consistency. It is simple enough to be implemented in the network data plane. In addition, it only requires $f+1$ nodes to tolerate f node failures, which is lower than $2f+1$ nodes required by the ordinary Paxos, due to the existence of the reconfiguration protocol. This is important as switches have limited on-chip memory for key-value storage. Hence, given the same number of switches, the system can store more items with Vertical Paxos. (*ii*) The heavy lifting for fault-tolerance is offloaded to the reconfiguration protocol, which uses an auxiliary master to handle reconfiguration operations like joining (for new nodes) and leaving (for failed nodes). The auxiliary master can be mapped to the network control plane, as modern datacenter networks already have a logically centralized controller replicated on multiple servers.



(a) NetChain architecture.



(b) NetChain packet format.

Figure 2: NetChain overview.

While it seems to move the fault-tolerance problem from the consensus protocol to the auxiliary master, Vertical Paxos is well-suited to NetChain because reconfigurations such as failures (on the order of minutes) are orders of magnitude less frequent than queries (on the order of microseconds). So handling queries and reconfigurations are mapped to data and control planes, respectively.

Chain Replication. We design a variant of chain replication (CR) [15] to implement the steady state protocol of Vertical Paxos. CR is a form of PB protocols. In the classical PB protocol (Figure 1(a)), all queries are sent to a primary node. The primary node needs to keep some state to track each write query to each backup node, and to retry or abort a query if it does not receive acknowledgments from all backup nodes. Keeping the state and confirming with all backup nodes are costly to implement with the limited resources and operations provided by switch ASICs. In CR (Figure 1(b)), nodes are organized in a chain structure. Read queries are handled by the tail; write queries are sent to the head, processed by each node along the chain, and replied by the tail. Write queries in CR use fewer messages than PB (n+1 instead of 2n where n is the number of nodes). CR only requires each node to apply a write query locally and then forward the query. Receiving a reply from the tail is a direct indication of query completion. Thus CR is simpler than PB to be implemented in switches.

3 NetChain Overview

We design NetChain, an in-network coordination service that provides sub-RTT latency and high throughput. It provides a strongly-consistent, fault-tolerant key-value store abstraction to applications (Figure 2(a)).

NetChain data plane (§4)**.** We design a replicated key-value store with programmable switches. Both read and

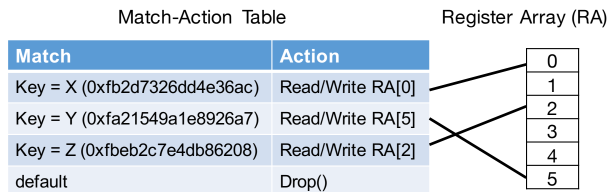


Figure 3: Key-value index and storage.

write queries are *directly processed in the switch data plane* without controller involvement. We design a variant of CR to ensure strong consistency of the key-value store. Coordination queries use a custom network format based on UDP (Figure 2(b)) and the processing logic of NetChain is invoked by a reserved UDP port.

NetChain control plane (§5). The controller handles system reconfigurations such as switch failures, which are orders of magnitude less frequent than read and write queries. The controller is assumed to be reliable by running a consensus protocol on multiple servers, as required by Vertical Paxos. Note that NetChain can also work with traditional distributed network control planes. In such scenarios, we need to implement an auxiliary master for handling system reconfigurations, and the network control plane should expose an interface for NetChain to manage NetChain's tables and registers in switches. NetChain does not need access to any other switch state, such as forwarding tables.

NetChain client. NetChain exposes a key-value store API to applications. A NetChain agent runs in each host, and translates API calls from applications to queries using our custom packet format. The agent gathers returned packets from NetChain and generates API responses. The complexity of the in-network key-value store is hidden from applications by the agent.

4 NetChain Data Plane

Problem statement. The data plane provides a replicated, in-network key-value store, and handles read and write queries directly. We implement CR to guarantee strong consistency, which involves three specific problems: ($R1$) how to store and serve key-value items in each switch; ($R2$) how to route queries through the switches according to the chain structure; ($R3$) how to cope with best-effort network transport (i.e., packet reordering and loss) between chain switches.

Properties. NetChain provides strong consistency: (i) reads and writes on individual keys are executed in some sequential order, and (ii) the effects of successful writes are reflected in subsequent reads. NetChain assumes trustworthy components; otherwise, malicious components can destroy these consistency guarantees.

Algorithm 1 ProcessQuery(pkt)

- *sequence*: the register array that stores sequence numbers
- *value*: the register array that stores values
- *index*: the match table that stores array locations of keys

1: $loc \leftarrow index[pkt.key]$
2: **if** $pkt.op == read$ **then**
3: Insert value header field $pkt.val$
4: $pkt.val \leftarrow value[loc]$
5: **else if** $pkt.op == write$ **then**
6: **if** $isChainHead(pkt)$ **then**
7: $sequence[loc] \leftarrow sequence[loc] + 1$
8: $pkt.seq \leftarrow sequence[loc]$
9: $value[loc] \leftarrow pkt.val$
10: **else if** $pkt.seq > sequence[loc]$ **then**
11: $sequence[loc] \leftarrow pkt.seq$
12: $value[loc] \leftarrow pkt.val$
13: **else** Drop()
14: Update packet header and forward

4.1 Data Plane Key-Value Storage

On-chip key-value storage. Modern programmable switch ASICs (e.g., Barefoot Tofino [13]) provide on-chip register arrays to store user-defined data that can be *read* and *modified* for each packet at line rate. NetChain separates the storage of *key* and *value* in the on-chip memory. Each *key* is stored as an entry in a match table, and each *value* is stored in a slot of a register array. The match table's output is the index (location) of the matched key. Figure 3 shows an example of the indexing. Key `X` is stored in slot 0 of the array, and key `Z` is stored in slot 2. NetChain uses the same mechanism as NetCache [18] to support variable-length values with multiple stages.

UDP-based key-value query. We leverage the capability of programmable switches to define a custom packet header format (Figure 2(b)) and build a UDP-based query mechanism (Algorithm 1). The core header fields for key-value operations are `OP` (which stands for *operator*), `KEY` and `VALUE`. Other fields are used for routing (§4.2) and ordered delivery (§4.3). NetChain supports four operations on the key-value store: `Read` and `Write` the value of a given key; `Insert` and `Delete` key-value items. `Read` and `Write` queries are entirely handled in the data plane at line rate. `Delete` queries invalidate key-value items in the data plane and requires the control plane for garbage collection. `Insert` queries require the control plane to set up entries in switch tables, and thus are slower than other operations. This is acceptable because `Insert` is a less frequent operation for the use cases of coordination services. Most queries are reads and writes on configuration parameters and locks that already exist. Nevertheless, if necessary in particular use cases, data plane insertions are feasible using data structures like d-left hashing, but at the cost of low utilization of the on-chip memory.

Data partitioning with consistent hashing. NetChain

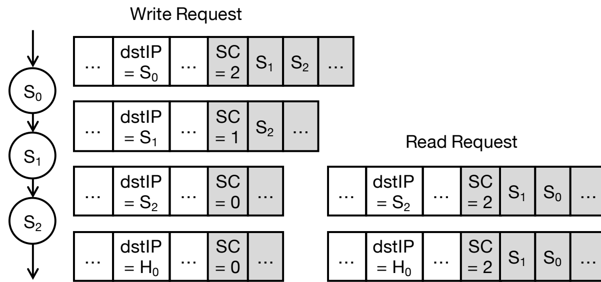
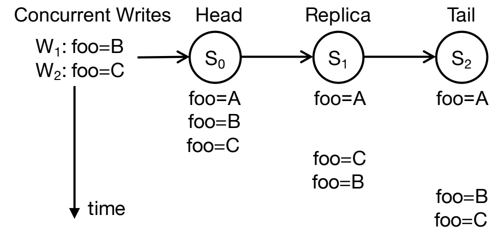
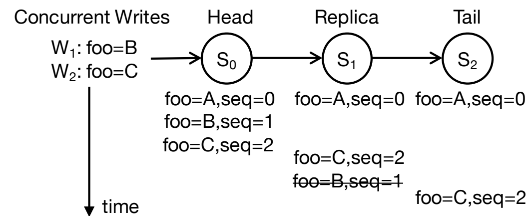


Figure 4: NetChain routing.



(a) Problem of out-of-order delivery.



(b) Serialization with sequence numbers.

Figure 5: Serializing out-of-order delivery.

uses consistent hashing [22] to partition the key-value store over multiple switches. Keys are mapped to a hash ring, and each switch is responsible for several continuous segments on the ring. Virtual nodes [23] are used to help evenly spread the load. Given n switches, NetChain maps m virtual nodes to the ring and assign m/n virtual nodes to each switch. Keys of each segment on the ring are assigned to $f+1$ subsequent virtual nodes. In cases where a segment is assigned to two virtual nodes that are physically located on the same switch, NetChain searches for the following virtual nodes along the ring until we find $f+1$ virtual nodes that all belong to different switches.

4.2 NetChain Routing

NetChain routing protocol. Our goal is to route queries for switch processing according to the chain structure. This is different from *overlay routing* that uses servers as intermediate hops and *underlay routing* that specifies hop-by-hop routing. We require that certain hops must be visited in the chain order, but do not care how queries are routed from one chain node to the next. While this is similar to *segment routing* and *loose source routing*, the write and read queries visit the chain switches in different orders, and read queries only visit the tail of the chain. We build the NetChain routing protocol on top of existing underlay routing protocols. This allows us to partially deploy NetChain with only a few switches being NetChain nodes, and take advantage of many properties provided by existing routing protocols, e.g., fast rerouting upon failures.

Specifically, we assign an IP address for each switch, and store an IP list of the chain nodes in the packet header (Figure 2(b)). *SC*, which is short for switch count, denotes the number of remaining switches in the chain. The destination IP in the IP header indicates the next chain node for the query. When a switch receives a packet and the destination IP matches its own address, the switch decodes the query and performs the read or write operation. After this, the switch updates the destination IP to the next chain node, or to the client IP if it is the tail.

Write queries store chain IP lists as the chain order from head to tail; read queries use the reverse order (switch IPs other than the tail are used for failure handling with details in §5). The chain IP lists are encoded to UDP payloads by NetChain agents. As we use consistent hashing, a NetChain agent only needs to store a small amount of data to maintain the mapping from keys to switch chains. The mapping only needs to be updated when the chain is reconfigured, which does not happen frequently. As will be described in §5, we only need to make local changes on a few switches to quickly reflect chain reconfigurations in the network, followed by a slower operation to propagate the changes to all agents.

Example. The example in Figure 4 illustrates NetChain routing for chain $[S_0, S_1, S_2]$. A write query is first sent to S_0, and contains S_1 and S_2 in its chain IP list. After processing the query, S_0 copies S_1 to the destination IP, and removes S_1 from the list. Then the network uses its underlay routing protocol to route the query based on the destination IP to S_1, unaware of the chain. After S_1 processes the packet, it updates the destination IP to S_2 and forwards the query to S_2. Finally, S_2 receives the query, modifies the query to a reply packet, and returns the reply to the client H_0. A read query is simpler: it is directly sent to S_2, which copies the value from its local store to the packet and returns the reply to the client.

4.3 In-Order Key-Value Update

Problem of out-of-order delivery. CR is originally implemented on servers and uses TCP to serialize messages. However, when we move CR to switches and use UDP, the network data plane only provides best-effort packet delivery. Packets can arrive out of order from one switch to another, which introduces consistency problems that do not exist in the server-based CR. Figure 5(a) shows the problem of out-of-order delivery. We have a key `foo` that is replicated on S_0, S_1 and S_2 with initial

value `A`. We have two concurrent write queries, which modify the value to `B` and `C` respectively. W_1 and W_2 are reordered when they arrive at S_1, and are reordered again when they arrive at S_2. `foo` has inconsistent values on the three switch replicas. Clients would see `foo` changing from `C` to `B` if S_2 fails. To make things worse, if S_1 also fails, clients would see the value of the item reverting to `C` again. This violates the consistency guarantees provided by CR.

Serialization with sequence numbers. We use sequence numbers to serialize write queries, as shown in Algorithm 1. Each item is associated with a sequence number, which is stored in a dedicated register array that shares the same key indexes with the value array. The head switch assigns a *monotonically increasing* sequence number to each write query, and updates its local value. Other switches only perform write queries with higher sequence numbers. Figure 5(b) shows how the out-of-order delivery problem is fixed by sequence numbers. S_1 drops W_1 as W_1 carries a lower sequence number. The value is consistent on the three replicas.

4.4 Comparison with Original CR

NetChain replaces the servers used in the original CR with programmable switches, which requires new solutions for $R1$, $R2$ and $R3$ (see the preamble of §4). These solutions are limited by switch capabilities and resources. Here we describe the differences and the intuitions behind the correctness of the new protocol.

The solution to $R1$ (§4.1) implements *per-key* read and *write* queries, as opposed to *per-object* read and *update* queries. NetChain does not support multi-key transactions. The motivation for this change is due to characteristics of modern switches, which have limited computation and storage. Note that per-object operations are more general than per-key operations, as objects can include multiple keys or even the entire database. In principle it is possible to pack multiple application values into a single database value for atomic and consistent updates, although there are practical limitations on value sizes, as discussed in §6. Likewise, updates are more general than writes, as they need not be idempotent and may require multiple operations to implement. However, writes are still sufficient to implement state updates and so are often provided as primitives in the APIs of modern coordination services [2, 3].

The solution to $R2$ (§4.2) routes queries through switches, which does not change the CR protocol.

The solution to $R3$ (§4.3) delivers messages between chain switches over UDP. The original CR assumes ordered, reliable message delivery (implemented with TCP), which is not available on switches. NetChain uses *sequence numbers* to order messages, and relies on *client-side retries* (e.g., based on a timeout) when messages are lost. Because datacenter networks are typically well managed, the already-small packet loss rate can be further reduced by prioritizing NetChain packets, since coordination services are critical and do not have large bandwidth requirements. In addition, because writes are idempotent, retrying is benign. By considering every (repeated) write as a new operation, the system is able to ensure strong consistency. Note that client-side retries are also used by the original CR to handle message losses between the clients and the system, because it is difficult to track the liveness of the clients which are peripheral to the system.

CR and its extensions have been used by many key-value stores to achieve strong consistency and fault-tolerance, such as FAWN-KV [24], Flex-KV [25] and HyperDex [26]. The design of NetChain is inspired by these server-based solutions, especially FAWN-KV [24] and its Ouroboros protocol [27]. The unique contribution of NetChain is that it builds a strongly-consistent, fault-tolerant key-value store into the *network*, with the switch on-chip key-value storage, a routing protocol for chain-based query routing and an ordering protocol for query serialization, which can all be realized in the *switch data plane*. In addition, as we will show in §5, compared to the failure handling in the Ouroboros protocol [27], NetChain leverages the network topology to reduce the number of updated nodes for failover, and relies on the controller to synchronize state in failure recovery instead of directly through the storage nodes themselves.

4.5 Protocol Correctness

To prove that our protocol guarantees consistency in unreliable networks with switch failures, we show that the following invariant always holds, which is a relaxation of the invariant used to establish the correctness of the original CR [15].

Invariant 1 *For any key k that is assigned to a chain of nodes $[S_1, S_2, ..., S_n]$, if $1 \leq i < j \leq n$ (i.e., S_i is a predecessor of S_j), then $State^{S_i}[k].seq \geq State^{S_j}[k].seq$.*

Formally, we restrict the history to one with a single value for each key and show that state transitions of NetChain can be mapped to those of the original CR, proving that NetChain provides per-key consistency. The proof shows that when a write query between switches is lost, it is equivalent to a state where the write query is processed by the tail switch, but the item is overwritten by a later write query, before the first query is exposed by any read queries. Moreover, as explained in §5, under various failure conditions, the protocol ensures that for each key, the versions exposed to client read queries are monotonically increasing. We provide a TLA+ verification of our protocol in the extended version [28].

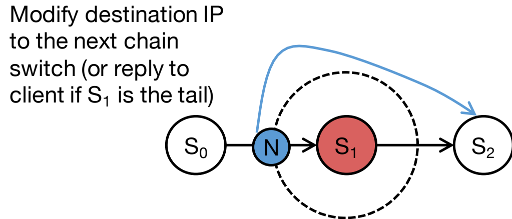


Figure 6: Fast failover.

Algorithm 2 Failover(fail_sw)

```
1: for sw ∈ fail_sw.neighbors() do
2:     rule.match ← dst_ip = fail_sw
3:     if fail_sw is not tail then
4:         rule.action ← (dst_ip = chain_ip[1], pop two chain IPs)
5:     else
6:         rule.action ← (swap src_ip & dst_ip, pop chain IP)
7:     sw.insert(rule)
```

5 NetChain Control Plane

Overview. The NetChain controller runs as a component in the network controller and only manages switch tables and registers related to NetChain. As discussed in §3, we assume the network controller is reliable. We mainly consider system reconfigurations caused by switch failures, which are detected by the network controller using existing techniques. We assume the failure model is fail-stop, and the switch failures can be correctly detected by the controller. To gracefully handle switch failures, we divide the process into two steps: *fast failover* and *failure recovery*. (i) In fast failover, the controller quickly reconfigures the network to resume serving queries with the remaining f nodes in each affected chain. This degrades an affected chain to tolerate $f-1$ node failures. (ii) In failure recovery, the controller adds other switches as new replication nodes to the affected chains, which restores these chains to f+1 nodes. Since failure recovery needs to copy state to the new replicas, it takes longer than fast failover.

Other types of chain reconfigurations that (temporarily) remove switches from the network (e.g., switch firmware upgrade) are handled similarly to fast failover; those that add switches to the network (e.g., new switch onboarding) are handled similarly to failure recovery.

5.1 Fast Failover

Fast failover quickly removes failed switches and minimizes the durations of service disruptions caused by switch failures. Given a chain $[S_1, S_2, ..., S_k]$, when S_i fails, the controller removes S_i and the chain becomes $[S_1, ..., S_{i-1}, S_{i+1} ..., S_k]$. A strawman approach is to notify the agents on all servers so that when the agents send out requests, they would put IPs of the updated chains to the packet headers. The drawback is the huge cost of disseminating chain updates to hundreds of thousands of servers in a datacenter. Another solution is to update S_i's previous hop S_{i-1}. Specifically, after S_{i-1} finishes processing the requests, it pops up one more IP address from the chain IP list and uses S_{i+1} as the next hop. Given n switches and m virtual nodes, each switch is mapped to m/n virtual nodes. Since each virtual node is in f+1 chains, a switch failure affects $m(f+1)/n$ chains in total. This implies that we need to update $m(f+1)/n$ switches for one switch failure in fast failover. While $m(f+1)/n$ is fewer than the number of servers, it still incurs considerable reconfiguration cost as m/n can be a few tens or hundreds. Furthermore, if S_i is the head (tail) of a chain, the *previous* hop for write (read) queries would be *all the servers*, which need to be updated. In order to minimize service disruptions, we want to *reduce the number of nodes that must be updated* during failover.

Reducing number of updated nodes. Our key idea is that the controller only needs to update the *neighbor switches* of a failed switch to remove it from *all* its chains (Algorithm 2). Specifically, the controller inserts a new rule to each neighbor switch which examines the destination IP. If the destination IP is that of the failed switch (line 2), the neighbor switches copy the IP of the next chain hop of the failed switch (i.e., S_{i+1}'s IP) to the destination IP (line 3-4), or reply to the client if the failed switch is the tail (line 5-6). The condition of whether the failed switch is the tail (line 3) is implemented by checking the size of the chain IP list. Write and read queries are handled in the same way as they store the original and reverse orders of the chain IP list respectively. Figure 6 illustrates this idea. All requests from S_0 to S_1 have to go through S_1's neighbor switches, which are represented by the dashed circle. N is one of the neighbor switches. After S_1 fails, the controller updates N to use S_2's IP as the new destination IP. Note that if N overlaps with S_0 (S_2), it updates the destination IP after (before) it processes the query. If S_1 is the end of a chain, N would send the packet back to the client.

With this idea, even if a failed switch is the head or tail of a chain, the controller only needs to update its neighbor switches, instead of updating all the clients. Multiple switch failures are handled in a similar manner, i.e., updating the neighbor switches of the failed switches, and NetChain can only handle up to f node failures for a chain of f+1 nodes.

5.2 Failure Recovery

Failure recovery restores all chains to f+1 switches. Suppose a failed switch S_i is mapped to virtual nodes $V_1, V_2, ..., V_k$. These virtual nodes are removed from their chains in fast failover. To restore them, we first randomly assign them to k live switches. This helps spread the load of failure recovery to multiple switches rather than a sin-

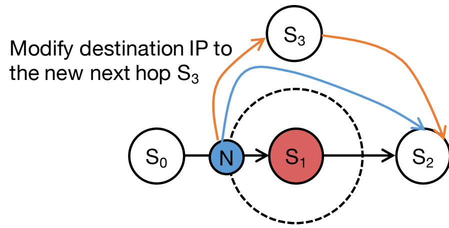


Figure 7: Failure recovery.

Algorithm 3 FailureRecovery(fail_sw, new_sw)

```
1: ref_sw ← getLiveReferenceSwitch(fail_sw)
2: preSyncState(new_sw, ref_sw)
3: for sw ∈ fail_sw.neighbors() do
4:     sw.stopForward(fail_sw)
5: syncState(new_sw, ref_sw)
6: new_sw.activateProcess()
7: for sw ∈ fail_sw.neighbors() do
8:     sw.activateForward(fail_sw, new_sw)
```

gle switch. Let V_x be reassigned to switch S_y. Since V_x belongs to $f+1$ chains, we need to add S_y to each of them. We use Figure 7 to illustrate how we add a switch to a chain. Fast failover updates S_1's neighbor switches to forward all requests to S_2 (the blue line from N to S_2). Failure recovery adds S_3 to the chain to restore the chain to three nodes (the orange lines from N to S_3 and from S_3 to S_2). The process involves two steps, described below. Note that in essence the process is similar to live virtual machine migrations [29] and reconfigurations for server-based chain replication [24]. The difference is that we have the network controller to copy state and to modify switch rules to perform the reconfigurations. Algorithm 3 gives the pseudo code. In the example, $fail_sw$, new_sw and ref_sw are S_1, S_3 and S_2, respectively.

Step 1: Pre-synchronization. The controller copies state from S_2 to S_3 (line 2). This includes (i) copying values from S_2's register array to S_3's register array, and (ii) inserting rules to S_3's index table (Figure 3). While this step is time-consuming, the availability is not affected as the chain continues its operation with S_0 and S_2. We cover the case where there is no S_2 in the paragraph below on handling special cases.

Step 2: Two-phase atomic switching. The controller switches the chain from $[S_0, S_2]$ to $[S_0, S_3, S_2]$. This step has to be done carefully to ensure consistency. We keep the following invariant: *any node in the chain has newer values than its next hop.* The switching has two phases.

- **Phase 1: Stop and synchronization.** The controller inserts rules to *all* neighbors of S_1 (N in Figure 7) to stop forwarding queries to S_2 (line 3-4). At the same time, it continues synchronizing the state between S_2 and S_3 (line 5). Since *no* new queries are forwarded to S_2, the state on S_2 and S_3 would eventually become the same. In this phase, the chain stops serving write queries, but still serves read queries with S_2.
- **Phase 2: Activation.** The controller activates the new chain switch S_3 to start processing queries (line 6), and activates *all* neighbor switches of S_1 (N in the figure) to forward queries to S_3 (line 7-8). These rules modify the destination IP to S_3's IP. They override the rules of fast failover (by using higher rule priorities), so that instead of being forwarded to S_2, all queries would be forwarded to S_3. The chain is restored after this phase.

Note that Step 1 is actually an optimization to shorten the temporary downtime caused by Phase 1 in Step 2, as most data is synchronized between the switches after Step 1. The invariant is kept throughout Phase 1 and Phase 2: in Phase 1, S_3 is synchronized to the same state as S_2 before it is added to the chain; in Phase 2, the neighbor switches of S_1 gradually restart to forward write queries again, which always go to S_3 first. The read queries in this process are continuously handled by S_2 and do not have temporary disruptions as write queries. The failure recovery process is performed for each failed switch under multiple failures.

Handling special cases. The above example shows the normal case when S_1 is in the middle of a chain. Now we discuss the special cases when S_1 is the head or tail of a chain. (i) When S_1 is the head, the process is the same as S_1 is in the middle. In addition, since the head switch assigns sequence numbers, the new head must assign sequence numbers bigger than those assigned by the failed head. To do this, we adopt a mechanism similar to NOPaxos [20], which uses an additional session number for message ordering. The session number is increased for every new head of a chain, and the messages are ordered by the lexicographical order of the (session number, sequence number) tuple. (ii) When S_1 is the tail, in Step 1, we copy state from its previous hop to it as we do not have S_2 (line 1 would assign S_0 to ref_sw). In Phase 1 of Step 2, both read and write queries are dropped by S_1's neighbors to ensure consistency. Although S_0 is still processing write queries during synchronization, we only need to copy the state from S_0 that are updated before we finish dropping all queries on S_1's neighbor switches, as no new read queries are served. Then in Phase 2, we activate the chain by forwarding the queries to S_3.

Minimizing disruptions with virtual groups. While we guarantee the consistency for failure recovery with two-phase atomic switching, write queries need to be stopped to recover head and middle nodes, and both read and write queries need to be stopped to recover tail nodes. We use virtual groups to minimize the service disruptions. Specifically, we only recover one virtual group each time. Let a switch be mapped to 100 virtual groups. Each group is available by 99% of the recovery time and only queries to one group are affected at each time.

6 Discussion

Switch on-chip storage size. Coordination services are not intended to provide generic storage services. We argue that the switch on-chip memory is sufficient from two aspects. (*i*) *How much can the network provide?* Commodity switches today have tens of megabytes of on-chip SRAM. Because datacenter networks do not use many protocols as WANs and use shallow buffers for low latency, a large portion of the on-chip SRAM can be allocated to NetChain. Assuming 10MB is allocated in each switch, a datacenter with 100 switches can provide 1GB total storage, or 333MB effective storage with a replication factor of three. The number can be further increased with better semiconductor technology and allocating more on-chip SRAM to NetChain. (*ii*) *How much does NetChain need?* As shown in [1], a Chubby instance at Google that serves tens of thousands of clients store 22k files in total, among which ∼90% are 0-1 KB and only ∼0.2% are bigger than 10KB. This means 42MB is enough to store ∼99.8% of the files. Furthermore, consider the use case for locks. Assuming each lock requires 30B, then 333MB storage can provide 10 million concurrent locks. For in-memory distributed transactions that take $100\mu s$, NetChain would be able to provide 100 billion locks per second, which is enough for the most demanding systems today. Even with a small deployment using three switches (providing 10MB storage), NetChain would be able to provide 0.3 million concurrent locks or 3 billion locks per second.

Value size. The value size is limited by the packet size (9KB for Ethernet jumbo frames). Conceivably, a big value can be stored with multiple keys, but strong consistency is not provided for a multi-key query spanning several packets. The value size is also limited by the switch chip. Typically, a switch pipeline contains multiple stages (k) and each stage can read or write a few bytes (n). Assuming k=12 and n=16, switches can handle values up to kn=192 bytes at line rate. Values bigger than kn can be supported using packet mirroring/recirculation which sends packets to go through the pipeline for another round of processing, but at the cost of lower effective throughput [18]. We suggest that NetChain is best suitable for small values that need frequent access, such as configuration parameters, barriers and locks.

Data placement. NetChain uses consistent hashing and virtual nodes to partition the key-value store between multiple switches. The data placement strategy can be optimized for throughput and latency by taking into account the network topology and the query characteristics.

Full interface. The current NetChain prototype provides a basic key-value interface for fixed-length keys and limited-size variable-length values. Many commercial and open-source systems like ZooKeeper provide

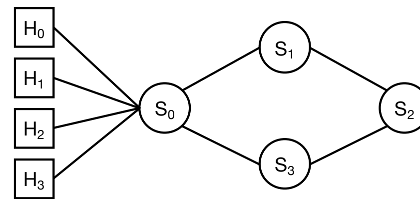


Figure 8: NetChain testbed with four 6.5Tbps Barefoot Tofino switches (S_0-S_3) and four servers (H_0-H_3).

additional features, e.g., hierarchical name space (as a file system), watches (which notify clients when watched values are updated), access control list (ACL) and data encryption. We leave these features as future work.

Accelerator for server-based solutions. NetChain can be used as an accelerator to server-based solutions such as Chubby [1], ZooKeeper [2] and etcd [3]). The key space is partitioned to store data in the network and the servers separately. NetChain can be used to store *hot* data with *small* value size, and servers store *big* and *less popular* data. Such a hybrid approach provides the advantages of both solutions.

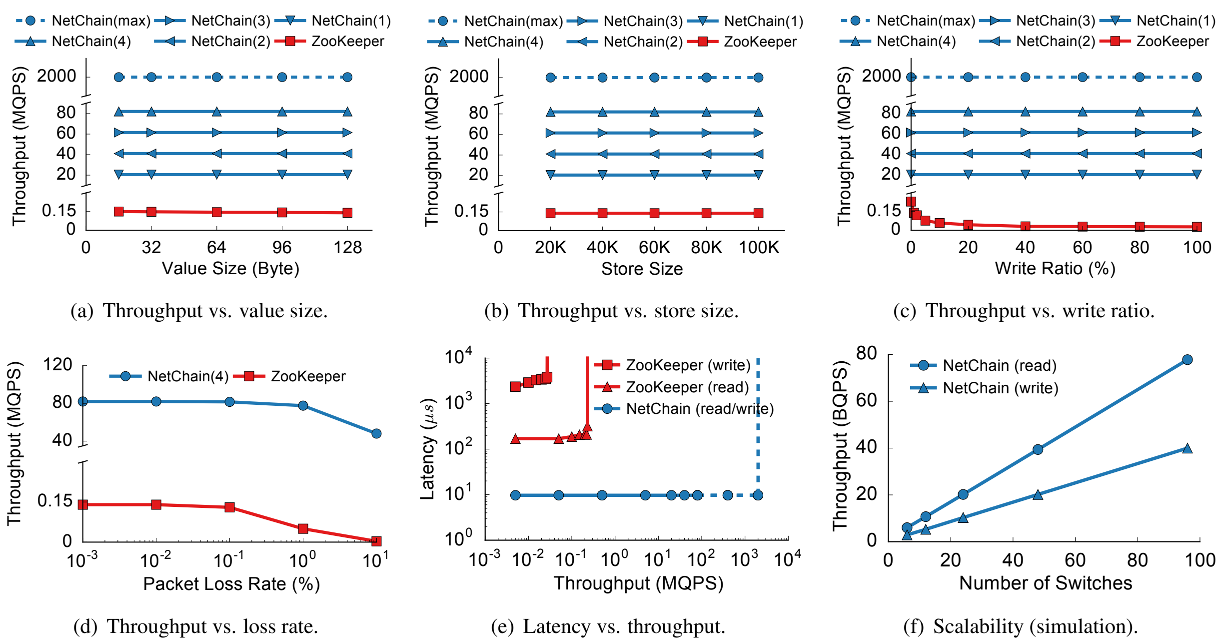
7 Implementation

We have implemented a prototype of NetChain, including a switch data plane, a controller and a client agent. (*i*) The switch data plane is written in P4 [30] and is compiled to Barefoot Tofino ASIC [13] with Barefoot Capilano software suite [31]. We use 16-byte keys and use 8 stages to store values. We allocate 64K 16-byte slots in each stage. This in total provides 8 MB storage. We use standard L3 routing that forwards packets based on destination IP. In total, the NetChain data plane implementation uses much less than 50% of the on-chip memory in the Tofino ASIC. (*ii*) The controller is written in Python. It runs as a process on a server and communicates with each switch agent through the standard Python RPC library xmlrpclib. Each switch agent is a Python process running in the switch OS. It uses a Thrift API generated by the P4 compiler to manage the switch resources and update the key-value items in the data plane (e.g., for failure handling) through the switch driver. (*iii*) The client agent is implemented in C with Intel DPDK [32] for optimized IO performance. It provides a key-value interface for applications, and achieves up to 20.5 MQPS with the 40G NICs on our servers.

8 Evaluation

In this section, we provide evaluation results to demonstrate NetChain provides orders of magnitude improvements on throughput (§8.1) and latency (§8.2), is scalable (§8.3), handles failures gracefully (§8.4), and significantly benefits applications (§8.5).

Testbed. Our evaluation is conducted on a testbed consisting of four 6.5 Tbps Barefoot Tofino switches and



(a) Throughput vs. value size.
(b) Throughput vs. store size.
(c) Throughput vs. write ratio.
(d) Throughput vs. loss rate.
(e) Latency vs. throughput.
(f) Scalability (simulation).

Figure 9: Performance results. (a-e) shows the experimental results of a three-switch NetChain prototype. Netchain(1), Netchain(2), Netchain(3) and Netchain(4) correspond to measuring the prototype performance with one, two, three and four servers respectively. NetChain(max) is the theoretical maximum throughput achievable by a three-switch chain; it is not a measured throughput. (f) shows the simulation results of spine-leaf networks of various sizes.

four server machines. Each server machine is equipped with a 16-core CPU (Intel Xeon E5-2630) and 128 GB total memory (four Samsung 32GB DDR4-2133 memory). Three server machines are equipped with 40G NICs (Intel XL710) and the other one is equipped with a 25G NIC (Intel XXV710). The testbed is organized in a topology as shown in Figure 8.

Comparison. We compare NetChain to Apache ZooKeeper-3.5.2 [33]. We implement a client to measure ZooKeeper's performance with Apache Curator-4.0.0 [34], which is a popular client library for ZooKeeper. The comparison is slightly unfair: NetChain does not provide all features of ZooKeeper (§6), and ZooKeeper is a production-quality system that compromises its performance for many software-engineering objectives. But at a high level, the comparison uses ZooKeeper as a reference for server-based solutions to demonstrate the performance advantages of NetChain.

8.1 Throughput

We first evaluate the throughput of NetChain. We use three switches to form a chain $[S_0, S_1, S_2]$, where S_0 is the head and S_2 is the tail. Each server can send and receive queries at up to 20.5 MQPS. We use NetChain(1), NetChain(2), NetChain(3), NetChain(4) to denote the measured throughput by using one, two, three and four servers, respectively. We use Tofino switches in a mode that guarantees up to 4 BQPS throughput and each query packet is processed twice by a switch (e.g., a query from H_0 follows path H_0-S_0-S_1-S_2-S_1-S_0-H_0). Therefore, the maximum throughput of the chain is 2 BQPS in this setup. As the four servers cannot saturate the chain, we use NetChain(max) to denote the maximum throughput of the chain (shown as dotted lines in figures). For comparison, we run ZooKeeper on three servers, and a separate 100 client processes on the other server to generate queries. This experiment aims to thoroughly evaluate the throughput of one switch chain under various setups with real hardware switches. For large-scale deployments, a packet may traverse multiple hops to get from one chain switch to the next, and we evaluate the throughput with simulations in §8.3. Figure 9(a-d) shows the throughputs of the two systems. The default setting uses 64-byte value size, 20K store size (i.e., the number of key-value items), 1% write ratio, and 0% link loss rate. We change one parameter in each experiment to show how the throughputs are affected by these parameters.

Figure 9(a) shows the impact of value size. NetChain provides orders of magnitude higher throughput than ZooKeeper and both systems are not affected by the value size in the evaluated range. NetChain(4) keeps at 82 MQPS, meaning that NetChain can fully serve all the queries generated by the four servers. This is due to the nature of a switch ASIC: as long as the P4 program is compiled to fit the switch resource requirements, the switch is able to run NetChain at line rate. In fact,

a three-switch chain is able to provide up to 2 BQPS, as denoted by NetChain(max). Our current prototype support value size up to 128 bytes. Larger values can be supported using more stages and using packet mirroring/recirculation as discussed in §6.

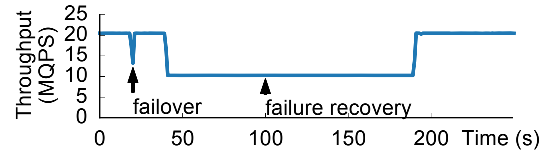
Figure 9(b) shows the impact of store size. Similarly, both systems are not affected by the store size in the evaluated range, and NetChain provides orders of magnitude higher throughput. The store size is restricted by the allocated total size (8MB in our prototype) and the value size. The store size is large enough to be useful for coordination services as discussed in §6.

Figure 9(c) shows the impact of write ratio. With read-only workloads, ZooKeeper achieves 230 KQPS. But even with a write ratio of 1%, its throughput drops to 140 KQPS. And when the write ratio is 100%, its throughput drops to 27 KQPS. As for comparison, NetChain(4) consistently achieves 82 MQPS. This is because NetChain uses chain replication and each switch is able to process both read and write queries at line rate. As the switches in the evaluated chain $[S_0, S_1, S_2]$ process the same number of packets for both read and write queries, the total throughput is not affected by the write ratio, which would be different in more complex topologies. As we will show in §8.3, NetChain has lower throughput for write queries for large deployments, as write queries require more hops than read queries.

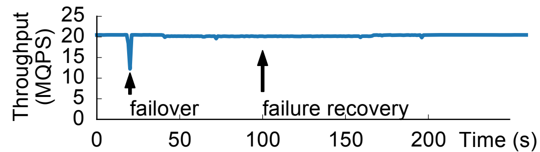
Figure 9(d) shows the impact of packet loss rate. We inject random packet loss rate to each switch, ranging from 0.001% to 10%. The throughput of ZooKeeper drops to 50 KQPS (3 KQPS) when the loss rate is 1% (10%). As for comparison, NetChain(4) keeps around 82 MQPS for packet loss rate between 0.001% and 1%, and only drops to 48 MPQS when the loss rate is 10%. The reason is because ZooKeeper uses TCP for reliable transmission which has a lot of overhead under high loss rate, whereas NetChain simply uses UDP and lets the clients retry a query upon packet loss. Although high packet loss rate is unlikely to happen frequently in datacenters, this experiment demonstrates that NetChain can provide high throughput even under extreme scenarios.

8.2 Latency

We now evaluate the latency of NetChain. We separate the read and write queries, and measure their latencies under different throughputs. For NetChain, since the switch-side processing delay is sub-microsecond, the client-side delay dominates the query latency. In addition, as both read and write queries traverse the same number of switches in the evaluated chain, NetChain has the same latency for both reads and writes, as shown in Figure 9(e). Because we implement NetChain clients with DPDK to bypass the TCP/IP stack and the OS kernel, NetChain incurs only 9.7 μs query latency. The latency keeps at 9.7 μs even when all four severs are generating queries to the system at 82 MQPS (the solid line of NetChain in the figure), and is expected to be not affected by throughput until the system is saturated at 2 BQPS (the dotted line of NetChain in the figure).



(a) 1 Virtual Group.



(b) 100 Virtual Groups.

Figure 10: Failure handling results. It shows the throughput time series of one client server when one switch fails in a four-switch testbed. NetChain has fast failover. By using more virtual groups, NetChain provides smaller throughput drops for failure recovery.

As for comparison, ZooKeeper has a latency of 170 μs for read queries and 2350 μs for write queries at low throughput. The latencies slightly go up before the system is saturated (27 KQPS for writes and 230 KQPS for reads), because servers do not have deterministic per-query processing time as switch ASICs and the latency is affected by the system load. Overall, NetChain provides orders of magnitude lower latency than ZooKeeper at orders of magnitude higher throughput.

8.3 Scalability

We use simulations to evaluate the performance of NetChain in large-scale deployments. We use standard spine-leaf topologies. We assume each switch has 64 ports and has a throughput of 4 BQPS. Each leaf switch is connected to 32 servers in its rack, and uses the other 32 ports to connect to spine switches. We assume the network is non-blocking, i.e., the number of spine switches is a half of that of leaf switches. We vary the network from 6 switches (2 spines and 4 leafs) to 96 switches (32 spines and 64 leafs). Figure 9(f) shows the maximum throughputs for read-only and write-only workloads. Both throughputs grow linearly, because in the two-layer network, the average number of hops for a query does not change under different network sizes. The write throughput is lower than the read throughput because a write query traverses more hops than a read query. When the queries have mixed read and write operations, the throughput curve will be between NetChain(read) and NetChain(write).

8.4 Handling Failures

The four-switch testbed allows us to evaluate NetChain under different failure conditions. For example, to evaluate the failure of a middle (tail) node, we can use an initial chain $[S_0, S_1, S_2]$, fail S_1 (S_2), and let S_3 replace S_1 (S_2). Since failing S_0 would disconnect the servers from the network, to evaluate a head failure and show the system throughput changes during the transition, we can use an initial chain $[S_1, S_2, S_0]$, fail S_1, and let S_3 replace S_1. Due to the limited space, we show one such experiment where we fail S_1 in the chain $[S_0, S_1, S_2]$ and use S_3 to replace S_1 for failure recovery. We use a write ratio of 50%, and let the write queries use path S_0-S_1-S_2 and the read queries use path S_0-S_3-S_2. In this way, we can demonstrate that when the middle node fails, the system is still able to serve read queries. We show the throughput time series of one client with different numbers of virtual groups.

Figure 10(a) shows the throughput time series with only one virtual group. Initially, the client has a throughput of 20.5 MQPS. Then at 20s, we inject the failure of S_1 by letting S_0 drop all packets to S_1. NetChain quickly fails over to a two-switch chain $[S_0, S_2]$, and the client throughput is fully restored. Note that to make the throughput drop visible, we manually inject a one-second delay to the controller before it starts the failover routine. In practice, the duration of failover depends on how fast the control plane can detect the failure, which is beyond the scope of this paper. We also add 20 seconds to separate the failure recovery process from the failover process. The failure recovery starts around 40s and lasts about 150s. The throughput drops to half because NetChain needs to synchronize the key-value store between S_2 and S_3, during which write queries cannot be served. Note that if it was a tail failure, the throughput would drop to 0, because both read and write queries cannot be served during the recovery of the tail (§5).

Figure 10(b) shows the time series of NetChain throughput with 100 virtual groups. The failover part is similar, but the failure recovery part only experiences a 0.5% throughput drop. This is because by using 100 virtual groups, NetChain only needs to stop serving write queries for one virtual group each time, or 1% of queries. Since we use a write ratio of 50%, only $1\% \times 50\% = 0.5\%$ of queries are affect during failure recovery. Therefore, using more virtual groups provides both slight throughput drops and better availability.

8.5 Application Performance

Finally, we use distributed transactions as an application to further demonstrate the benefits of NetChain. We use a benchmark workload from previous work [35, 36], which is a generalization of the new-order transaction in TPC-C benchmark [37], because the workload allows us

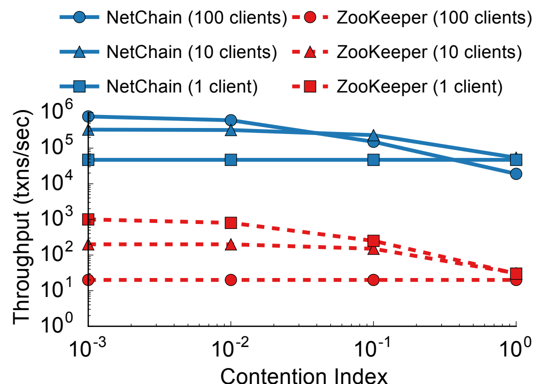


Figure 11: Application results. NetChain provides orders of magnitude higher transaction throughput.

to test transactions with different contention levels. In this workload, for each transaction, a client needs to acquire ten locks, where one lock is chosen from a small set of hot items and the other nine locks are from a very large set of items. The workload uses *contention index*, which is the inverse of the number of hot items, to control the contentions. For example, a contention index of 0.001 (or 1) means all clients compete for one item in 1000 (or 1) items. We use the classic two-phase locking (2PL) protocol: each client first acquire all the locks from NetChain or ZooKeeper, and then releases all the locks to complete one transaction. For NetChain, we use the compare-and-swap (CAS) primitive in Tofino switches to implement exclusive locks. Specifically, in addition to checking sequence numbers, a lock can only be released by the client that owns the lock by comparing the client ID in the value field. For ZooKeeper, exclusive locks can be implemented by ephemeral znodes and are directly provided by Apache Curator client library.

Figure 11 shows the transaction throughputs. By using NetChain as a locking server, the system can achieve orders of magnitude higher transaction throughput than ZooKeeper. We observe that the line with one client is flat because there are no contentions with one client. With 100 clients, the system has higher throughput at small contention index, because more clients can do more transactions. But the throughput decreases when the contention index increases, and is even slightly lower than that with one client, due to more contentions. We expect a lightning fast coordination system like NetChain can open the door for designing a new generation of distributed systems beyond distributed transactions.

9 Related Work

Consensus protocols. Various protocols have been proposed to solve the distributed consensus problem, such as Paxos [4], ZAB [38], Raft [39], Viewstamped Replication [40], Virtual Synchrony [41], etc. By leveraging that in some scenarios messages arrive in order, some

protocols are designed to reduce the overhead, such as Fast Paxos [42] and Optimistic Atomic Broadcast [43]. Recent work goes one step further by ordering messages with the network, including SpecPaxos [44] and NOPaxos [20]. NetChain directly processes coordination queries in the network, providing higher performance.

Coordination services. As distributed coordination is widely used, some systems are built to provide coordination as a service, e.g., Chubby [1], ZooKeeper [2] and etcd [3]. These services have highly-optimized implementations of consensus protocols in their core and provide simple APIs. NetChain provides a similar key-value API and implements the system in the network.

Hardware accelerations. NetPaxos [16, 17] implements Paxos on switches. It still requires to build a replicated key-value store on servers, which may use NetPaxos for consensus to improve performance. As such, the key-value store is still bounded by server IO, and thus is much slower than NetChain. Besides, NetChain uses Vertical Paxos for consensus which is more suitable for in-network key-value stores, provides protocols and algorithms for routing and failure handling, and has an implementation with multiple switches and an evaluation. Compared to NetCache [18], NetChain uses NetCache's on-chip key-value store design, and designs a replicated, in-network key-value store that handles both read and write queries and provides strong consistency and fault-tolerance. Some other work has also used hardware to speed up distributed systems. SwitchKV [45] uses switches to enable content-based routing for load balancing of key-value stores; the switches do not cache key-value items or serve queries. Marple [46] designs a new hardware primitive to support key-value stores for network measurements. Schiff *et al.* [47] designs a synchronization framework to resolve conflicts for distributed network controllers, which has to go through the switch control plane. Li *et al.* [48] proposes a new hardware design to achieve a throughput of 1 BQPS with a single server platform. IncBricks [49] caches key-value items with NPUs. Consensus in a box [50] uses FPGAs to speed up ZooKeeper's atomic broadcast protocol. NetChain does not require specialized chips, and switch ASICs have higher performance than NPUs and FPGAs.

10 Conclusion

We present NetChain, an in-network coordination service that provides billions of coordination operations per second with sub-RTT latencies. NetChain leverages programmable switches to build a strongly-consistent, fault-tolerant, in-network key-value store. This powerful capability dramatically reduces the coordination latency to as little as half of an RTT. We believe NetChain exemplifies a new generation of ultra-low latency systems enabled by programmable networks.

Acknowledgments We thank our shepherd Amar Phanishayee and the anonymous reviewers for their valuable feedback. Robert Soulé is supported in part by SNF 167173. Nate Foster is supported in part by NSF CNS-1413972. Ion Stoica is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, and gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

[1] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *USENIX OSDI*, November 2006.

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *USENIX ATC*, June 2010.

[3] "etcd key-value store." `https://github.com/coreos/etcd`.

[4] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, May 1998.

[5] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: Distributed transactions with consistency, availability, and performance," in *ACM SOSP*, October 2015.

[6] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *ACM SOSP*, October 2015.

[7] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," in *VLDB*, September 2014.

[8] "MongoDB." `https://www.mongodb.com/`.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SOSP*, October 2007.

[10] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," in *VLDB*, August 2015.

[11] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke, "The homeostasis protocol: Avoiding transaction coordination through program analysis," in *ACM SIGMOD*, May 2015.

[12] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *USENIX OSDI*, November 2016.

[13] "Barefoot Tofino." https://www.barefootnetworks.com/technology/#tofino.

[14] "Broadcom Tomahawk II." https://www.broadcom.com/.

[15] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *USENIX OSDI*, December 2004.

[16] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "NetPaxos: Consensus at network speed," in *ACM SOSR*, June 2015.

[17] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *SIGCOMM CCR*, April 2016.

[18] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing key-value stores with fast in-network caching," in *ACM SOSP*, October 2017.

[19] "Cavium XPliant." https://www.cavium.com/.

[20] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say NO to Paxos overhead: Replacing consensus with network ordering," in *USENIX OSDI*, November 2016.

[21] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *ACM PODC*, August 2009.

[22] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM Symposium on Theory of Computing*, May 1997.

[23] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *ACM SOSP*, October 2001.

[24] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A fast array of wimpy nodes," in *ACM SOSP*, October 2009.

[25] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini, "Flex-KV: Enabling high-performance and flexible KV systems," in *Workshop on Management of Big Data Systems (MBDS)*, September 2012.

[26] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A distributed, searchable key-value store," in *ACM SIGCOMM*, August 2012.

[27] A. Phanishayee, *Chaining for Flexible and High-Performance Key-Value Systems*. PhD thesis, Carnegie Mellon University, September 2012.

[28] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-free sub-RTT coordination (extended version)," in *arXiv*, February 2018.

[29] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *USENIX NSDI*, May 2005.

[30] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM CCR*, July 2014.

[31] "Barefoot Capilano." https://www.barefootnetworks.com/technology/#capilano.

[32] Intel, "Intel data plane development kit (dpdk)," 2017. http://dpdk.org/.

[33] apache, "Apache zookeeper," 2017. http://zookeeper.apache.org/.

[34] curator, "Apache curator," 2017. http://curator.apache.org/.

[35] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *ACM SIGMOD*, May 2012.

[36] K. Ren, A. Thomson, and D. J. Abadi, "Lightweight locking for main memory database systems," *VLDB*, December 2012.

[37] "TPC-C." http://www.tpc.org/tpcc/.

[38] B. Reed and F. P. Junqueira, "A simple totally ordered broadcast protocol," in *ACM Large-Scale Distributed Systems and Middleware*, September 2008.

[39] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC*, June 2014.

[40] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *ACM PODC*, August 1988.

[41] K. Birman and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems," *SIGOPS Operating Systems Review*, November 1987.

[42] L. Lamport, "Fast Paxos," *Distributed Computing*, October 2006.

[43] F. Pedone and A. Schiper, "Optimistic atomic broadcast," in *International Symposium on Distributed Computing*, September 1998.

[44] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing distributed systems using approximate synchrony in data center networks," in *USENIX NSDI*, May 2015.

[45] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with SwitchKV," in *USENIX NSDI*, March 2016.

[46] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *ACM SIGCOMM*, August 2017.

[47] L. Schiff, S. Schmid, and P. Kuznetsov, "In-band synchronization for distributed SDN control planes," *SIGCOMM CCR*, January 2016.

[48] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *ACM/IEEE ISCA*, June 2015.

[49] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "IncBricks: Toward in-network computation with an in-network cache," in *ACM ASPLOS*, April 2017.

[50] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *USENIX NSDI*, March 2016.

Azure Accelerated Networking: SmartNICs in the Public Cloud

Daniel Firestone Andrew Putnam Sambhrama Mundkur Derek Chiou Alireza Dabagh
Mike Andrewartha Hari Angepat Vivek Bhanu Adrian Caulfield Eric Chung
Harish Kumar Chandrappa Somesh Chaturmohta Matt Humphrey Jack Lavier Norman Lam
Fengfen Liu Kalin Ovtcharov Jitu Padhye Gautham Popuri Shachar Raindel Tejas Sapre
Mark Shaw Gabriel Silva Madhan Sivakumar Nisheeth Srivastava Anshuman Verma Qasim Zuhair
Deepak Bansal Doug Burger Kushagra Vaid David A. Maltz Albert Greenberg
Microsoft

Abstract

Modern cloud architectures rely on each server running its own networking stack to implement policies such as tunneling for virtual networks, security, and load balancing. However, these networking stacks are becoming increasingly complex as features are added and as network speeds increase. Running these stacks on CPU cores takes away processing power from VMs, increasing the cost of running cloud services, and adding latency and variability to network performance.

We present Azure Accelerated Networking (AccelNet), our solution for offloading host networking to hardware, using custom Azure SmartNICs based on FPGAs. We define the goals of AccelNet, including programmability comparable to software, and performance and efficiency comparable to hardware. We show that FPGAs are the best current platform for offloading our networking stack as ASICs do not provide sufficient programmability, and embedded CPU cores do not provide scalable performance, especially on single network flows.

Azure SmartNICs implementing AccelNet have been deployed on all new Azure servers since late 2015 in a fleet of $>$1M hosts. The AccelNet service has been available for Azure customers since 2016, providing consistent $<$15μs VM-VM TCP latencies and 32Gbps throughput, which we believe represents the fastest network available to customers in the public cloud. We present the design of AccelNet, including our hardware/software co-design model, performance results on key workloads, and experiences and lessons learned from developing and deploying AccelNet on FPGA-based Azure SmartNICs.

1 Introduction

The public cloud is the backbone behind a massive and rapidly growing percentage of online software services [1, 2, 3]. In the Microsoft Azure cloud alone, these services consume millions of processor cores, exabytes of storage, and petabytes of network bandwidth. Network performance, both bandwidth and latency, is critical to most cloud workloads, especially interactive customer-facing workloads.

As a large public cloud provider, Azure has built its cloud network on host-based software-defined networking (SDN) technologies, using them to implement almost all virtual networking features, such as private virtual networks with customer supplied address spaces, scalable L4 load balancers, security groups and access control lists (ACLs), virtual routing tables, bandwidth metering, QoS, and more. These features are the responsibility of the host platform, which typically means software running in the hypervisor.

The cost of providing these services continues to increase. In the span of only a few years, we increased networking speeds by 40x and more, from 1GbE to 40GbE+, and added countless new features. And while we built increasingly well-tuned and efficient host SDN packet processing capabilities, running this stack in software on the host requires additional CPU cycles. Burning CPUs for these services takes away from the processing power available to customer VMs, and increases the overall cost of providing cloud services.

Single Root I/O Virtualization (SR-IOV) [4, 5] has been proposed to reduce CPU utilization by allowing direct access to NIC hardware from the VM. However, this direct access would bypass the host SDN stack, making the NIC responsible for implementing all SDN policies. Since these policies change rapidly (weeks to months), we required a solution that could provide software-like programmability while providing hardware-like performance.

In this paper we present Azure Accelerated Networking (AccelNet), our host SDN stack implemented on the FPGA-based Azure SmartNIC. AccelNet provides near-native network performance in a virtualized environment, offloading packet processing from the host CPU to the Azure SmartNIC. Building upon the software-based VFP host SDN platform [6], and the hardware and software infrastructure of the Catapult program [7, 8], AccelNet provides the performance of dedicated hardware, with the programmability of software running in the hypervisor. Our goal is to present both our design and our experiences running AccelNet in production at scale, and lessons we learned.

2 Background

2.1 Traditional Host Network Processing

In the traditional device sharing model of a virtualized environment such as the public cloud, all network I/O to and from a physical device is exclusively performed in the host software partition of the hypervisor. Every packet

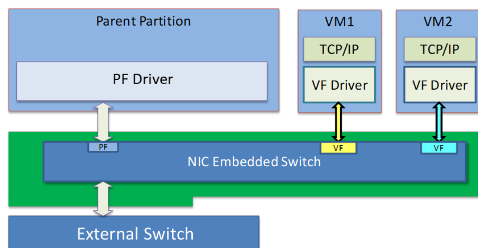


Figure 1: An SR-IOV NIC with a PF and VFs.

sent and received by a VM is processed by the Virtual Switch (vSwitch) in the host networking stack. Receiving packets typically involves the hypervisor copying each packet into a VM-visible buffer, simulating a soft interrupt to the VM, and then allowing the VM's OS stack to continue network processing. Sending packets is similar, but in the opposite order. Compared to a non-virtualized environment, this additional host processing: reduces performance, requires additional changes in privilege level, lowers throughput, increases latency and latency variability, and increases host CPU utilization.

2.2 Host SDN

In addition to selling VMs, cloud vendors selling Infrastructure-as-a-Service (IaaS) have to provide rich network semantics, such as private virtual networks with customer supplied address spaces, scalable L4 load balancers, security groups and ACLs, virtual routing tables, bandwidth metering, QoS, and more. These semantics are sufficiently complex and change too frequently that it isn't feasible to implement them at scale in traditional switch hardware. Instead, these are implemented on each host in the vSwitch. This scales well with the number of servers, and allows the physical network to be simple, scalable and very fast.

The Virtual Filtering Platform (VFP) is our cloud-scale programmable vSwitch, providing scalable SDN policy for Azure. It is designed to handle the programmability needs of Azure's many SDN applications, providing a platform for multiple SDN controllers to plumb complex, stateful policy via match-action tables. Details about VFP and how it implements virtual networks in software in Azure can be found in [6].

2.3 SR-IOV

Many performance bottlenecks caused by doing packet processing in the hypervisor can be overcome by using hardware that supports SR-IOV. SR-IOV-compliant hardware provides a standards-based foundation for efficiently and securely sharing PCI Express (PCIe) device hardware among multiple VMs. The host connects to a privileged physical function (PF), while each virtual machine connects to its own virtual function (VF). A VF is exposed as a unique hardware device to each VM, allowing the VM direct access to the actual hardware, yet still isolating VM data from other VMs. As illustrated in Figure 1 , an SR-IOV NIC contains an embedded switch to forward packets to the right VF based on the MAC address. All data packets flow directly between the VM operating system and the VF, bypassing the host networking stack entirely. This provides improved throughput, reduced CPU utilization, lower latency, and improved scalability.

However, bypassing the hypervisor brings on a new set of challenges since it also bypasses all host SDN policy such as that implemented in VFP. Without additional mechanisms, these important functions cannot be performed as the packets are not processed by the SDN stack in the host.

2.4 Generic Flow Table Offload

One of AccelNet's goals was to find a way to make VFP's complex policy compatible with SR-IOV. The mechanism we use in VFP to enforce policy and filtering in an SR-IOV environment is called Generic Flow Tables (GFT). GFT is a match-action language that defines transformation and control operations on packets for one specific network flow. Conceptually, GFT is comprised of a single large table that has an entry for every active network flow on a host. GFT flows are defined based on the VFP unified flows (UF) definition, matching a unique source and destination L2/L3/L4 tuple, potentially across multiple layers of encapsulation, along with a header transposition (HT) action specifying how header fields are to be added/removed/changed.

Whenever the GFT table does not contain an entry for a network flow (such as when a new network flow is started), the flow can be vectored to the VFP software running on the host. VFP then processes all SDN rules for the first packet of a flow, using a just-in-time flow action compiler to create stateful exact-match rules for each UF (e.g. each TCP/UDP flow), and creating a composite action encompassing all of the programmed policies for that flow. VFP then populates the new entry in the GFT table and delivers the packet for processing.

Once the actions for a flow have been populated in the GFT table, every subsequent packet will be processed by the GFT hardware, providing the performance benefits of SR-IOV, but with full policy and filtering enforcement of VFP's software SDN stack.

3 Design Goals and Rationale

We defined the GFT model in 2013-2014, but there are numerous options for building a complete solution across hardware and software. We began with the following goals and constraints as we set out to build hardware offloads for host SDN:

1. Don't burn host CPU cores

Azure, like its competitors, sells VMs directly to customers as an IaaS offering, and competes on the price of those VMs. Our profitability in IaaS is the difference between the price a customer pays for a VM and what it costs us to host one. Since we have fixed costs per server, the best way to lower the cost of a VM is to pack more VMs

onto each host server. Thus, most clouds typically deploy the largest number of CPU cores reasonably possible at a given generation of 2-socket (an economical and performant standard) blades. At the time of writing this paper, a physical core (2 hyperthreads) sells for $0.10-0.11/hr[1], or a maximum potential revenue of around $900/yr, and $4500 over the lifetime of a server (servers typically last 3 to 5 years in our datacenters). Even considering that some fraction of cores are unsold at any time and that clouds typically offer customers a discount for committed capacity purchases, using even one physical core for host networking is quite expensive compared to dedicated hardware. Our business fundamentally relies on selling as many cores per host as possible to customer VMs, and so we will go to great lengths to minimize host overheads. Thus, running a high-speed SDN datapath using host CPU cores should be avoided.

2. Maintain host SDN programmability of VFP

VFP is highly programmable, including a multi-controller model, stateful flow processing, complex matching capabilities for large numbers of rules, complex rule-processing and match actions, and the ability to easily add new rules. This level of programmability was a key factor in Azure's ability to give customers highly configurable and feature-rich virtual networks, and enabling innovation with new virtual networking features over time. We did not want to sacrifice this programmability and flexibility for the performance of SR-IOV — in fact we wanted SDN controllers to continue targeting VFP without any knowledge that the policy was being offloaded. This would also maintain compatibility on host servers that do not have the necessary hardware for AccelNet.

Offloading every rule to hardware is neither feasible nor desirable, as it would either constrain SDN policy or require hardware logic to be updated every time a new rule was created. However, we concluded that offloading all rules is unnecessary. Most SDN policies do not change during the duration of the flow. So all policies can be enforced in VFP software on the first packet of a new TCP/UDP flow, after which the actions for that flow can be cached as an exact-match lookup. Even for short flows, we typically observe at least 7-10 packets including handshakes, so processing only the first packet in software still allows the majority to be offloaded (if the offload action is fast and efficient).

3. Achieve the latency, throughput, and utilization of SR-IOV hardware

Basic SR-IOV NICs set an initial bar for what is possible with hardware-virtualized networking — bypassing the host SDN stack and schedulers entirely to achieve low (and consistent) latency, high throughput, and no host CPU utilization. Offloading only exact match flows with associated actions allows for a tractable hardware design with the full performance of a native SR-IOV hardware solution on all but the first packet of each flow.

4. Support new SDN workloads and primitives over time

VFP continues to evolve, supporting new requirements and new policies, and AccelNet must be able to evolve along with VFP. We were, and continue to be, very wary of designs that locked us into a fixed set of flow actions. Not only does AccelNet need to support adding/changing actions, but the underlying platform should allow for new workloads that don't map neatly to a single exact-match table.

5. Rollout new functionality to the entire fleet

A corollary to the previous requirement, the AccelNet platform needed to enable frequent deployment of new functionality in the existing hardware fleet, not just on new servers. Customers should not have to move their existing deployments to new VM types to enable new features. Similarly, maintaining the same SDN functionality across hardware generations makes development, qualification, and deployment easier for us.

6. Provide high single-connection performance

From our experience with software-based SDN, we knew that network processing on a single CPU core generally cannot achieve peak bandwidth at 40Gb and higher. A good way to scale throughput past the limit of what one core can process is to break single connections into multiple parallel connections, utilizing multiple threads to spread load to multiple cores. However, spreading traffic across multiple connections requires substantial changes to customer applications. And even for the apps that implement multiple connections, we saw that many do not scale well over many flows because flows are often bursty — apps will dump large messages onto one flow while others remain idle.

An explicit goal of AccelNet is to allow applications to achieve near-peak bandwidths without parallelizing the network processing in their application.

7. Have a path to scale to 100GbE+

We designed AccelNet for a 2015 server generation that was going to deploy 40GbE widely. But we knew that the number of cores per server and the networking bandwidths would continue to increase in future generations, with speeds of 100GbE and above likely in the near future. We wanted a SmartNIC design that would continue to scale efficiently as network speeds and the number of VMs increase.

8. Retain Serviceability

VFP was designed to be completely serviceable in the background without losing any flow state, and supports live migration of all flow state with a VM being migrated. We wanted our SmartNIC software and hardware stack to have the same level of serviceability.

[1] Azure D v3 series or AWS EC2 m4 series instances, with price varying slightly by region

4 SmartNIC Hardware Design

4.1 Hardware Options

Based on the above goals, we proceeded to evaluate different hardware designs for our SmartNIC architecture.

Traditionally Microsoft worked with network ASIC vendors, such as Intel, Mellanox, Broadcom, and others, to implement offloads for host networking in Windows — for example TCP checksum and segmentation offloads in the 1990s [9], Receive-Side Scaling (RSS) [10] and Virtual Machine Queues (VMQ) [11] for multicore scalability in the 2000s, and more recently stateless offloads for NVGRE and VxLAN encapsulation for virtual networking scenarios for Azure in the 2010s [12]. In fact, GFT was originally designed to be implemented by ASIC vendors as an exact match-action table in conjunction with SR-IOV, and we shared early design ideas widely in the industry to see if vendors could meet our requirements. After time, our enthusiasm for this approach waned as no designs were emerging that could meet all of the design goals and constraints laid out in Section 3.

One major problem for SmartNIC vendors is that SR-IOV is an example of an all-or-nothing offload. If any needed SDN feature cannot be handled successfully in the SmartNIC, the SDN stack must revert to sending flows back through the software-based SDN stack, losing nearly all of the performance benefit of SR-IOV offload.

We saw four different possible directions emerge: ASICs, SoCs, FPGAs, and sticking with existing CPUs.

4.1.1 ASIC-based NICs

Custom ASIC designs for SDN processing provide the highest performance potential. However, they suffer from a lack of programmability and adaptability over time. In particular, the long time span between requirement specifications and the arrival of silicon was on the order of 1-2 years, and in that span requirements continued to change, making the new silicon already behind the software requirements. ASIC designs must continue to provide all functionality for the 5 year lifespan of a server (it's not feasible to retrofit most servers at our scale). All-or-nothing offloading means that the specifications for an ASIC design laid out today must meet all the SDN requirements for 7 years into the future.

ASIC vendors often add embedded CPU cores to handle new functionality. These cores can quickly become a performance bottleneck compared to rest of the NIC processing hardware. In addition, these cores can be expected to take an increasing burden of the processing over time as new functionality is added, exacerbating the performance bottleneck. These cores are also generally programmed via firmware updates to the NIC, which is handled by the ASIC vendors and slows the deployment of new features.

4.1.2 Multicore SoC-based NICs

Multicore SoC-based NICs use a sea of embedded CPU cores to process packets, trading some performance to provide substantially better programmability than ASIC designs. These designs became widely available in the 10GbE NIC generation. Some, like Cavium [13], used general purpose CPU cores (MIPS, later ARM64), while others, like Netronome [14] and Tilera, had specific cores for network processing. Within this space, we much preferred the general purpose SoCs — based on our evaluation that they were easier to program (you could take standard DPDK-style code and run it in a familiar Linux environment). To our surprise, these didn't have much of a drawback in performance compared to similar-generation ASIC designs.

However, at higher network speeds of 40GbE and above, the number of cores increases significantly. The on-chip network and schedulers to scatter and gather packets becomes increasingly complex and inefficient. We saw often 10 μs or more delays associated with getting packets into a core, processing the packet, and back out to the network — significantly higher latency than ASICs, and with significantly more variability. And stateful flows tend to be mapped to only one core/thread to prevent state sharing and out-of-order processing within a single flow. Thus individual network flow performance does not improve much because embedded CPUs are not increasing performance at the same pace as network bandwidths. This leads to the problem of developers having to spread their traffic across multiple flows, as discussed in Section 3, limiting the performance advantage of faster networks to only the most parallel workloads.

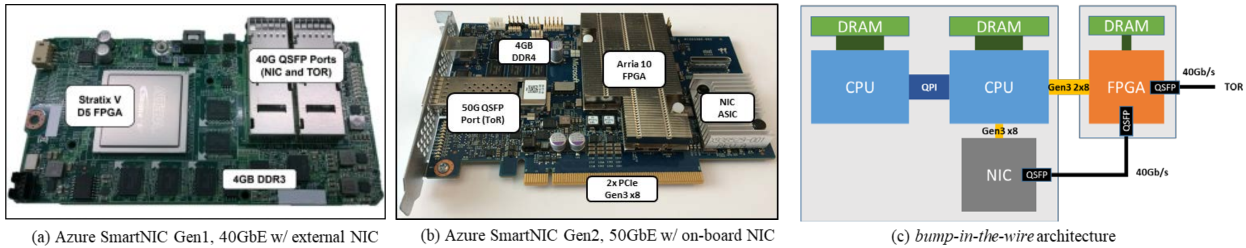
The future of SoC-style network offload is also questionable. At 10GbE, the total package was tolerable, with a few general purpose SoC cores being sufficient. 40GbE required nearly 4x the cores, though several vendors still created viable solutions. Still, 40GbE parts with software-based datapaths are already surprisingly large, power hungry, and expensive, and their scalability for 100GbE, 200GbE, and 400GbE looks bleak.

So while we found that the SoC approach has the advantage of a familiar programming model, the single-flow performance, higher latency, and poor scalability at higher network speeds left us looking for another solution.

4.1.3 FPGAs

Field programmable gate arrays (FPGAs) are reconfigurable hardware devices composed of small generic logic blocks and memories, all connected by a statically configured network. Programmers write code to assemble the generic logic and memory into "soft logic" circuits, forming custom application-specific processing engines — balancing the performance of ASICs with the programmability of SoC NICs.

In contrast to CPUs like those on SoC-based NICs, an



(a) Azure SmartNIC Gen1, 40GbE w/ external NIC (b) Azure SmartNIC Gen2, 50GbE w/ on-board NIC (c) *bump-in-the-wire* architecture

Figure 2: Azure SmartNIC boards with Bump-in-the-Wire Architecture

FPGA is programmed only with the essential elements to complete the application, and can even take advantage of application characteristics such as the maximum size of the data to reduce bus widths and storage requirements. There are many studies that demonstrate FPGA can accelerate applications several orders of magnitude over pure software implementations for a wide range of application spaces from microprocessor simulation [15], genomics [16], machine learning [17], networking, pattern matching, graph processing, and so on.

The key characteristics of FPGAs that made it attractive for AccelNet were the programmability to adapt to new features, the performance and efficiency of customized hardware, and the ability to create deep processing pipelines, which improve single-flow performance.

When we evaluated SmartNIC options, Microsoft had already done the work to deploy FPGAs as datacenter accelerators for project Catapult [7] — we had a successful multi-thousand node cluster of networked FPGAs doing search ranking for Bing, with greatly-improved performance and lowered costs, and with a network transport layer running between the FPGAs within a rack. This led us to believe that FPGAs could be a viable option at scale for SmartNIC, as they had the potential to solve our dilemma of wanting the performance characteristics of an ASIC, but the programmability and reconfigurability inherent in a software solution like an SoC.

4.1.4 Burn host cores

We still evaluated all options against our original strategy of just using host cores to run our SDN stack, especially as technologies such as DPDK [18] showed that we could lower the cost of packet processing significantly by bypassing the OS networking stack and running cores in poll-mode. This option beat out ASICs given we couldn't get ASICs to meet our programmability requirements, but the cost and performance overhead of burning cores to our VM hosting costs was sufficiently high as outlined in Section 3 that even the inefficient multicore SoCs were a better approach.

4.2 Evaluating FPGAs as SmartNICs

FPGAs seemed like a great option from our initial analysis, but our host networking group, who had until then operated entirely as a software group, was initially skeptical — even though FPGAs are widely used in networking in routers, cellular applications, and appliances, they were not commonly used as NICs or in datacenter servers, and the team didn't have significant experience programming or using FPGAs in production settings. A number of questions below had to be answered before we decided to go down this path:

1. Aren't FPGAs much bigger than ASICs?

The generic logic portions of FPGAs are roughly 10x-20x bigger than identical logic in ASICs, since programmable memories (look up tables, or LUTs) are used instead of gates, and a programmable network of wires and muxes are used instead of dedicated wires to connect components together. If the FPGA design were simply generic logic, we should expect to need 10-20x more silicon area than an ASIC. However, FPGAs have numerous hardened blocks, such as embedded SRAMs, transceivers, and I/O protocol blocks, all of which are custom components nearly identical to those found in custom ASICs.

Looking at a modern NIC, the packet processing logic is not generally the largest part. Instead, size is usually dominated by SRAM memory (e.g. to hold flow contexts and packet buffers), transceivers to support I/O (40GbE, 50GbE, PCIe Gen3), and logic to drive these interfaces (MAC+PCS for Ethernet, PCIe controllers, DRAM controllers), all of which can be hard logic on an FPGA as well. Furthermore, modern ASIC designs often include significant extra logic and configurability (and even embedded CPU cores) to accommodate different requirements from different customers. This extra logic is needed to maximize volumes, handle changing requirements, and address inevitable bugs. Prior work demonstrates that such configurability can add an order of magnitude of area to ASIC logic [19]. So the trend has been for FPGAs to include more and more custom logic, while ASICs include more and more programmable logic, closing the efficiency gap between the two alternatives.

In practice, we believe for these applications FPGAs should be around 2-3x larger than similarly functioned ASICs, which we feel is reasonable for the massively increased programmability and configurability.

2. Aren't FPGAs very expensive?

While we cannot disclose vendor pricing publicly, the FPGA market is competitive (with 2 strong vendors), and we're able to purchase at significant volumes at our scale.

In our experience, our scale allows non-recoverable engineering costs to be amortized, and the cost of the silicon becomes dominated by the silicon area and yield. Total silicon area in a server tends to be dominated by CPUs, flash, and DRAM, and yields are typically good for FPGAs due to their regular structure.

3. Aren't FPGAs hard to program?

This question was the source of the most skepticism from the host networking team, who were not at the time in the business of digital logic design or Verilog programming. FPGAs can provide incredible performance compared to a CPU for programmable logic, but only if hardware designers really think through efficient pipeline designs for an application and lay them out as such. The project was originally assisted by the Catapult team in Microsoft Research, but eventually we built our own FPGA team in Azure Networking for SmartNIC. The team is much smaller than a typical ASIC design team, with the AccelNet team averaging fewer than 5 FPGA developers on the project at any given time.

Our experience on AccelNet as well as other projects within Microsoft, such as Bing Ranking [7, 8] for web search, LZ77 for data compression [20], and BrainWave [17] for machine learning, demonstrate that programming FPGAs is very tractable for production-scale cloud workloads. The exact same hardware was used in all four of these applications, showing the programmability and flexibility of the Azure SmartNIC expands well beyond SDN and network processing capabilities. This bodes well as we seek to add new functionality in years to come. Investment in strong development teams, infrastructure, simulation capabilities, and tools is essential, but much of this can be shared across different teams.

We have found the most important element to successfully programming FPGAs has been to have the hardware and software teams work together in one group, and use software development methodologies (e.g. Agile development) rather than hardware (e.g. Waterfall) models. The flexibility of the FPGA allows us to code, deploy, learn, and revise at a much faster interval than is possible for any other type of hardware desig. This hardware/software co-design model is what enables hardware performance with software-like flexibility.

4. Can FPGAs be deployed at hyperscale?

Getting FPGAs into our data centers was not an easy effort — when project Catapult started this was just not a common use case for FPGAs, and the team had to work through numerous technical, logistical, and team structure issues. However by the time we began SmartNIC, Catapult had worked out many of the common infrastructure details that were needed for a hyperscale deployment. The Catapult shell and associated software libraries abstracted away underlying hardware-specific details and allowed both hardware and software development for SmartNIC to focus largely on application functionality. Though much of this functionality is now common for FPGA vendors to support, at the time it wasn't. This project would not have been feasible without the prior Catapult work.

5. Isn't my code locked in to a single FPGA vendor?

Today's FPGA development is done almost entirely in hardware description languages like SystemVerilog (which we use), which are portable if the original development was done with the intention to facilitate porting. There are vendor-specific details, for example Intel FPGAs have 40b wide SRAMs versus Xilinx's 36b SRAMs, but once such details are accounted for, compiling code for a different FPGA is not that difficult. As a proof point of portability, project Catapult was first developed on Xilinx FPGAs, but was ported over to Altera FPGAs before our original pilot.

4.3 SmartNIC System Architecture

Even after selecting FPGAs as the path forward, there were still major questions about how to integrate it — where should the FPGA fit in our system architecture for our first SmartNIC, and which functions should it include? The original Catapult FPGA accelerator [7] was deliberately not attached the data center network to avoid being a component that could take down a server, and instead was connected over an in-rack backend torus network. This was not ideal for use in SDN offload, since the FPGA needed to be on the network path to implement VFP functionality.

Another option was to build a full NIC, including SR-IOV, inside the FPGA — but this would have been a significant undertaking (including getting drivers into all our VM SKUs), and would require us to implement unrelated functionality that our currently deployed NICs handle, such as RDMA[14]. Instead we decided to augment the current NIC functionality with an FPGA, and initially focus FPGA development only on the features needed for offload of SDN.

The converged architecture places the FPGA as a bump-in-the-wire between the NIC and the Top of Rack (TOR) switch, making the FPGA a filter on the network. A cable connects the NIC to the FPGA and another cable connects the FPGA to the TOR. The FPGA is also connected by 2 Gen3x8 PCIe connections to the CPUs, useful for accelerator workloads like AI and web search. When used as an accelerator, the network connection (along with an RDMA-like lossless transport layer using DCQCN [21]) allows scaling to workloads such as large DNN models that don't fit on one chip. The resulting first generation Azure SmartNIC, deployed in all Azure compute servers beginning in 2015, is shown in Figure 2(a).

The second generation of SmartNIC, running at 50GbE, Figure 2(b), is designed for the Azure Project Olympus OCP servers [22]. We integrated a standard NIC with SR-IOV on the same board as the FPGA, keeping the same bump-in-the-wire architecture, but eliminating the separate NIC board and the cable between the NIC and the

FPGA, reducing cost, and upgrading to a newer Intel Arria 10 FPGA.

5 AccelNet System Design

The control plane for AccelNet is largely unchanged from the original VFP design in [6], and remains almost entirely in the hypervisor. It remains responsible for the creation and deletion of flows from the flow table, along with determining the associated policy for each flow. The data plane for AccelNet is offloaded to the FPGA SmartNIC. The driver for the NIC is augmented with a filter driver called the GFT Lightweight Filter (LWF), which abstracts the details of the split NIC/FPGA hardware from VFP to make the SmartNIC appear as a single NIC with both full SR-IOV and GFT support, and to help in serviceability, as discussed in detail in Section 7.1.

5.1 Software Design

While the vast majority of the packet processing workload for GFT falls onto the FPGA hardware, software remains responsible for control operations, including the setup/teardown of flows, health monitoring, and enabling serviceability so that flows can continue during updates to VMs and the FPGA hardware. A high-level view of our architecture is shown in Figure 3. The flow table may not contain a matching rule for a given packet. In these cases, the offload hardware will send the packet to the software layer as an Exception Packet. Exception packets are most common on the first packet of a network flow, when the flow is just getting established.

A special virtual port (vPort) dedicated to the hypervisor is established for exception packets. When the FPGA receives an exception packet, it overloads the 802.1Q VLAN ID tag in the packet to specify that it is an exception path packet, and forwards the packet to the hypervisor vPort. VFP monitors this port and performs the necessary flow creation tasks after determining the appropriate policy for the packet's flow. If the exception packet was destined for a VM on the same host, the VFP software can deliver the packet directly to the VM. If the exception packet was outbound (sent by a VM for a remote destination), then the VFP software must resend the packet to the SmartNIC, which it can do using the same dedicated hypervisor vPort.

VFP also needs to be aware of terminated connections so that stale connection rules do not match to new network flows. When the FPGA detects termination packets such as TCP packets with SYN, RST or FIN flag set, it duplicates the packet — sending the original packet to its specified destination, and an identical copy of the packet to the dedicated hypervisor vPort. VFP uses this packet to track TCP state and delete rules from the flow table.

5.2 FPGA Pipeline Design

The GFT datapath design was implemented on the Azure SmartNIC hardware described in 4.3. For the remainder of this section we focus on the implementation

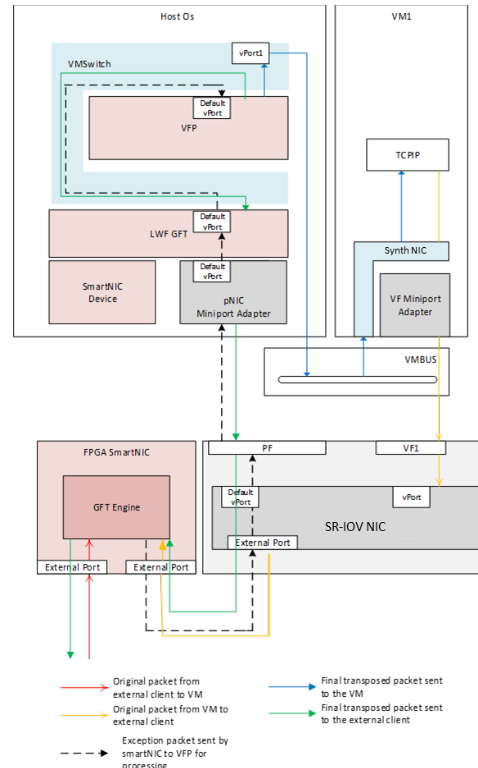


Figure 3: The SmartNIC GFT architecture, showing the flow of exception packets from the FPGA to software to establish a flow offloaded in hardware

on SmartNIC Gen1, though the same structure (with different values) applies to Gen2.

The design of the GFT implementation on FPGA can be divided into 2 deeply pipelined packet processing units, each comprised of four major pipeline stages: (1) a store and forward packet buffer, (2) a parser, (3) a flow lookup and match, and (4) a flow action. A high-level view of our system architecture is shown in Figure 4.

The Parser stage parses the aggregated header information from each packet to determine its encapsulation type. We currently support parsing and acting on up to 3 groups of L2/L3/L4 headers (9 headers total, and 310 possible combinatons). The output of the parser is a key that is unique for each network flow.

The third processing stage is Match, which looks up the rules for the packet based on the unique key from the Parser stage. Matching computes the Toeplitz hash [23] of the key, and uses that as the cache index. We use a 2-level caching system with an L1 cache stored on-chip, and an L2 cache stored in FPGA-attached private DRAM.

The L1 cache is structured as a direct-mapped cache supporting 2,048 flows. We experimented with 2-way set associative caches and simpler hash algorithms than the Toeplitz hash [24], but found that the more computationally-intensive but less collision-prone Toeplitz hash coupled with the simpler direct-mapped

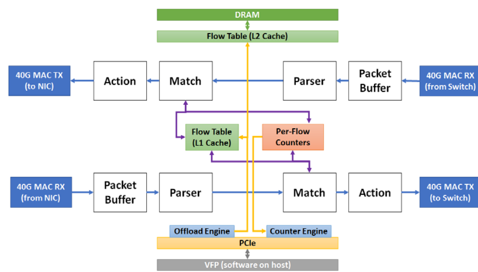


Figure 4: Block diagram of the GFT processing pipeline

cache resulted in better overall performance.

The L2 cache is structured as an 8-way set-associative cache, with support for O(1M) flows. Total number of supported flows is limited only by the DRAM capacity.

The final component is the Action stage, which takes the parameters looked up from the flow table, and then performs the specified transformations on the packet header. The Action block uses microcode to specify the exact behaviors of actions, enabling easy updates to actions without recompilation of the FPGA image. Only when entirely new actions are added will the FPGA need to be recompiled and a new bitstream loaded.

Non-trivial software-programmable Quality-of-Service guarantees can be implemented as optional components of the processing pipeline. For example, rate limiting can be done on a per-flow basis using a packet buffer in DRAM. Full description of our QoS frameworks and actions is beyond the scope of this paper. In total, the GFT role uses about 1/3 of the logic of the Intel Stratix V D5 chip that we used in the Gen1 SmartNICs.

5.3 Flow tracking and reconciliation

VFP is used by overlying controllers and monitoring layers to track per-flow connection state and data. GFT keeps track of all per-connection byte/flow counters, such as TCP sequence/ack numbers and connection state, and timestamps of the last time a flow got a packet. It periodically transmits all flow state to VFP via DMA transfers over PCIe, allowing VFP to ensure proper flow configurations, and to perform actions such as the cleanup of inactive flows.

GFT must also perform reconciliation so that flow actions get updated when VFP policy changes. Like VFP's unified flow table, GFT maintains a generation ID of the policy state on a system, and tracks what the generation ID when the rules for each flow were created. When a controller plumbs new policy to VFP, the generation ID on SmartNIC is incremented. Flows are then updated lazily by marking the first packet of each flow as an exception packet, and having VFP update the flow with the new policy actions.

6 Performance Results

Azure Accelerated Networking has been available since 2016. Performance results are measured on normal Azure AccelNet VMs in an Azure datacenter, running on Intel Xeon E5-2673 v4 (Broadwell at 2.3 Ghz) CPUs with 40Gbps Gen1 SmartNICs. Sender and receiver VMs are in the same datacenter and cross a Clos network of 5 standard switching ASICs between each other. We created no special configuration and the results, in Figure 5, should be reproduceable by any Azure customer using large Dv2 or Dv3 series Azure VMs. VFP policy applied to these VMs includes network virtualization, stateful NAT and stateful ACLs, metering, QoS, and more.

We measure one-way latency between two Windows Server 2016 VMs using registered I/O sockets [25] by sending 1 million 4-byte pings sequentially over active TCP connections and measuring response time. With our tuned software stack without AccelNet, we see an average of 50μs, with a P99 around 100μs and P99.9 around 300μs. With AccelNet, our average is 17μs with P99 of 25μs and P99.9 of 80μs — both latency and variance are much lower.

Azure offers VM sizes with up to 32Gbps of network capacity. With pairs of both Ubuntu 16.04 VMs and Windows 10 VMs with TCP congestion control set to CUBIC [26] and a 1500 Byte MTU, we consistently measure 31Gbps on a single connection between VMs with 0% associated CPU utilization in the host. Without AccelNet we see around 5Gbps on a single connection and multiple cores utilized in the host with enough connections running ($\sim$8) to achieve line rate. Because we don't have to scale across multiple cores, we believe we can scale single connection performance to 50/100Gb line rate using our current architecture as we continue to increase network speeds with ever-larger VMs.

For an example of a real world application, we deployed AccelNet to our Azure SQL DB fleet, which runs in VM instances, and ran SQL queries from an AccelNet VM in the same DC against an in-memory DB replicated across multiple nodes for high availability (both reads and writes to the service traverse multiple network hops). Average end-to-end read times went from $\sim$1ms to $\sim$300μs, and P99 read and write times dropped by over half as a result of reduced jitter in the network. Replication and seeding jobs that were often bound by the performance of a burst on a single connection ran over 2x faster.

Figure 6 shows comparative performance of AccelNet compared to other public cloud offerings that we measured on latest generation Intel Skylake-based instances (Fs72 v2 on Azure, c5.18xlarge on AWS, n1-highcpu-64 on GCP, measured in November 2017). All tests used unmodified Ubuntu 16.04 images provided by the platform with busy_poll enabled. We used the open source tools `sockperf` and `iperf` to measure latency and throughput, respectively. In our measurements, AccelNet had the lowest latencies, highest throughput, and lowest tail latencies (measured as percentiles of all pings over multiple 10-second runs of continuous ping-pong on established

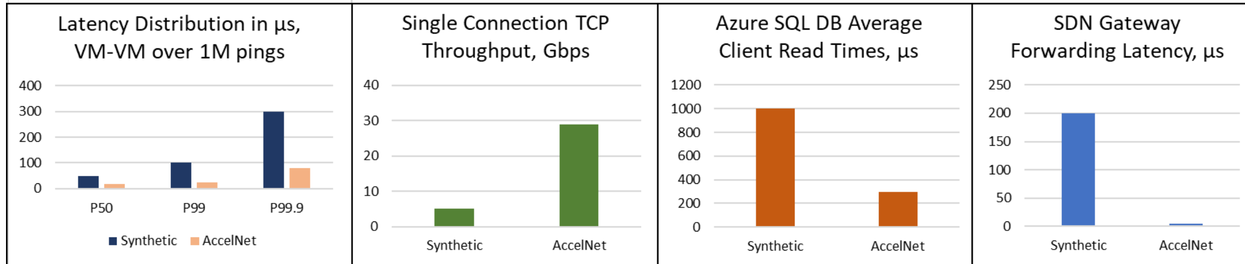


Figure 5: Selected performance data from Azure AccelNet VMs

TCP connections) of the instances we measured, including a consistent 10μs average latency between our Linux VMs. To test the performance of userspace I/O such as DPDK, we used a userspace TCP stack based on the open source VMA library [27], which achieved about 5μs latency pinging standard TCP sockets between our production VMs.

VFP is widely used in our fleet to run software gateways bridging between our SDN and external networks, such as ExpressRoute [28] circuits to customer datacenters. We used the programmable forwarding and QoS in our FPGA GFT implementation to offload all forwarding (after first packet on a flow) to the SmartNIC. Including encap/decap, stateful ACLs/NAT and metering, QoS, and forwarding, we saw a gateway forwarding line-rate 32Gbps traffic (even with just one flow), and consistent $<$5s latency with 0% host CPU utilization. Our prior gateway required multiple connections to hit line rate, burned CPU cores, and could spike to 100-200μs latency (including going to and from a VM) depending on system scheduling behavior. We believe this platform will let us create many more accelerated programmable appliances.

Using configurable power regulators on our SmartNIC, we've measured the power draw of our Gen1 board including all components in operational servers at 17-19W depending on traffic load. This is well below the 25W power allowed for a typical PCIe expansion slot, and on par with or less than other SmartNIC designs we've seen.

7 Operationalization

7.1 Serviceability

As with any other feature that is being built for the public cloud, serviceability, diagnostics and monitoring are key aspects of accelerated networking. The fact that both software and hardware are serviceable makes this particular scenario deployable for us. As discussed in [6], VFP is already fully serviceable while keeping existing TCP connections alive, and supports VM live migration with existing connections. With AccelNet, we needed to extend this serviceability as well — TCP flows and vNICs should survive FPGA reconfiguration, FPGA driver updates, NIC PF driver updates (which bring down VFs), and GFT driver updates.

We accomplished online serviceability by turning off hardware acceleration and switching back to synthetic vNICs to maintain connectivity when we want to service the SmartNICs or the software components that drive them, or when live migrating a VM. However, since AccelNet is exposed directly into the VM in the form of VFs, we must ensure that none of the applications break when the VF is revoked and the datapath is switched to synthetic mode. To satisfy this requirement, we do not expose the VF to the upper protocol stack in the VM directly. Instead, when the VF comes up, our synthetic NIC driver, the Hyper-V Network Virtual Service Consumer (NetVSC), marks the VF as its slave, either by using the slave mode present in the kernel for Linux VMs, or by binding to the NetVSC's upper NDIS edge in Windows VMs. We call this transparent bonding — the TCP/IP stack is bound only to the synthetic NIC. When VF is active, the synthetic adapter automatically issues sends over the VF adapter rather than sending it through the synthetic path to the host. For receives, the synthetic adapter forwards all receives from both the VF and synthetic path up the stack. When the VF is revoked for servicing, all transmit traffic switches to the synthetic path automatically, and the upper stack is not even aware of the VF's revocation or later reassignment. Figure 7 shows the accelerated data path and synthetic data path. The network stack is completely transparent to the current data path since NetVSC provides transparent bonding.

One of the benefits of SR-IOV is that VMs can use kernel bypass techniques like DPDK (Data Plane Development Kit) [18] or RDMA (Remote Direct Memory Access) to directly interface with hardware via the VF, but we needed to consider serviceability for them too when the VF is revoked, and the VM is potentially live-migrated elsewhere. We needed these applications to transparently fall back to a non-accelerated code path for that brief time period.

We found that there is no built-in fallback mechanism for many DPDK applications. So, we use a failsafe PMD (Poll Mode Driver) which acts as a bond between the VF PMD and a PMD on the synthetic interface. When the VF is active, the failsafe PMD operates over the VF PMD, thereby bypassing the VM kernel and the host software stack. When the VF is revoked for serviceability, the

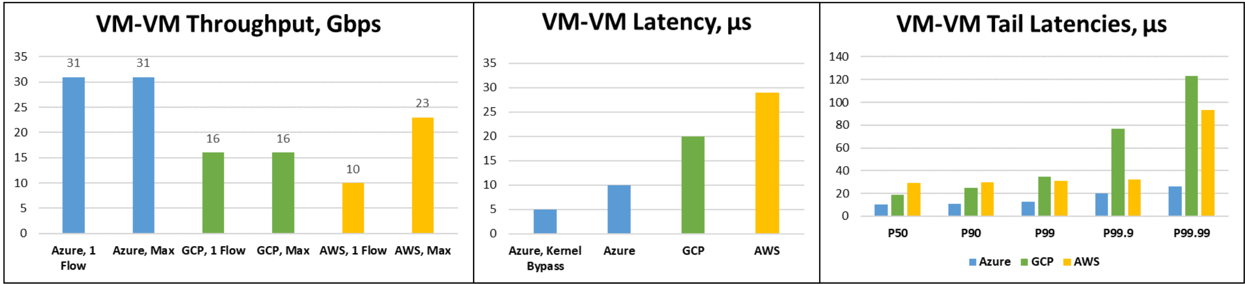


Figure 6: Performance of AccelNet VM-VM latencies vs. Amazon AWS Enhanced Networking and Google GCP Andromeda on Intel Skylake generation hardware.

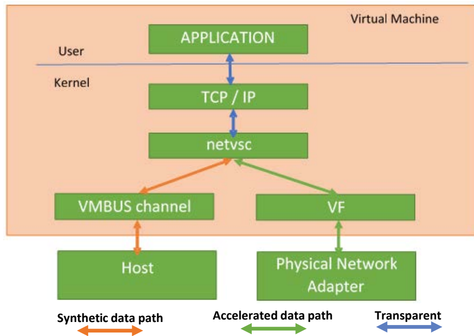


Figure 7: Transparent bonding between an SR-IOV interface and a synthetic interface

failsafe PMD starts operating over the synthetic path and packets flow through the VMBUS channels. Since the failsafe PMD exposes all DPDK APIs, the application does not see any difference except for a drop in performance for a short period of time.

For RDMA applications, this form of serviceability is harder and potentially involves many more queues. In practice, we found all our common RDMA applications are designed to gracefully fall back to TCP anyways, so we issue completions closing all RDMA queue pairs and let the app fail over to TCP. This isn't an issue for currently known workloads, but app-level transparency for RDMA serviceability remains an open question for the future if apps ever take a hard dependency on RDMA QPs staying alive.

Support for transparent VF bonding has been committed upstream in the Linux kernel (for NetVSC) and to dpdk.org for DPDK, and is natively available in Windows Server 2012 and later VMs. We've issued regular fleet-wide updates to all parts of the AccelNet stack, (VFP through GFT, the FPGA, and the PF driver), and found that transparent bonding works well in practice for our customers. While there is a short performance degradation while the synthetic path is active, apps stay alive and handle this well as they don't see a change in the network adapter they're bound to, or on active TCP connections. If an application is sensitive to this, we let VMs subscribe to an instance metadata service that sends notifications about upcoming maintenance and update events to the VM to enable it to prepare or move traffic elsewhere. If a VM is running behind the Azure load balancer, we can remove it from the active load balanced set during an update so that new external TCP/UDP flows are directed elsewhere for the duration of the update window.

7.2 Monitoring and Diagnostics

Monitoring is key to the reliability of a system like AccelNet at scale. Detection of early warning signs from both hardware and software and correcting them in an automated fashion is necessary to achieve production-quality reliability. We collect metrics from each component of AccelNet, including VFP, GFT, the SmartNIC and its driver, and the SR-IOV NIC and its driver — we have over 500 metrics per host (of which many are per-VM) collected in our scalable central monitoring system. Alerts and actions are triggered by combinations of these and we are constantly tweaking thresholds and actions as we get more experience and data over time with the system.

For diagnostics, we built programmable packet capture and tracing services at every interface on the SmartNIC — packet headers and data can be sampled at NIC/ToR ports on ingress/egress. We built a metadata interface along the network bus inside SmartNIC so that any module can emit diagnostic data about what exactly happened to a packet at that module, which is included in the capture. For example, in GFT we can trace how a packet was parsed, what flow it matched, what action it took, etc. We can collect hardware timestamps for these for accurate latency analysis. We also expose diagnostic information on key state machines as well as extensive counters, and automatically dump all critical internal state on an error.

8 Experiences

Azure SmartNICs and AccelNet have been deployed at scale for multiple years, with hundreds of thousands of customer VMs across our fleet. In our experience, AccelNet has improved network performance across the board for our customers without negatively impacting reliability or serviceability, and offering better throughput and latency than anything else we've measured in the public

cloud. We believe our design accomplished all the goals we set out in Section 3 :

1. We stopped burning CPU cores to run the network datapath for AccelNet VMs. Host cores show less than 1% utilization used for exception processing.
2. SDN controllers have continued to add and program new policy in VFP, agnostic of the hardware offload now underneath.
3. We measured the overhead of the FPGA on latency as $<1\mu$s vs our SR-IOV NIC alone, and achieve line rate. This is much better than CPU cores alone.
4. We've continued to add new actions and primitives to GFT on the FPGA to support new workloads, as well as new QoS primitives and more.
5. Changes have been rolled out across multiple types of servers and SmartNIC hardware.
6. We can achieve line rate on a single connection.
7. We believe our design scales well to 100Gb+.
8. We have done production servicing of our FPGA image and drivers regularly for years, without negatively impacting VMs or applications.

8.1 Are FPGAs Datacenter-Ready?

One question we are often asked is if FPGAs are ready to serve as SmartNICs more broadly outside Microsoft. We certainly do not claim that FPGAs are always the best and only solution for accelerating networking in all cloud environments. The development effort for FPGA programming is certainly higher than software — though can be made quite tractable and agile with a talented hardware team and support from multiple stakeholders.

When Microsoft started Catapult, FPGAs were far from cloud-ready. Because SmartNIC shares a common development environment and history with Catapult, much of the development effort was shared across teams. We've observed that necessary tooling, basic IP blocks, and general support have dramatically improved over the last few years. But this would still be a daunting task for a new team. We didn't find that the higher level languages for FPGAs we experimented with produced efficient results for our designs, but our trained hardware developers had no trouble rapidly iterating on our SystemVerilog code.

The scale of Azure is large enough to justify the massive development efforts — we achieved a level of performance and efficiency simply not possible with CPUs, and programmability far beyond an ASIC, at a cost that was reasonable because of our volume. But we don't expect this to be a natural choice for anyone beyond a large-scale cloud vendor until the ecosystem evolves further.

8.2 Changes Made

As we expected, we continued adding all kinds of actions over time as the SDN stack evolved that we could never have predicted when we started, such as new stateful tunneling modes and state tracking. We believe responding to these rapidly shifting requirements would never have worked in an ASIC development flow. A small selection of examples include:

- We've repeatedly extended our TCP state machine with more precise seq/ack tracking of every TCP flow in our system for various functional and diagnostic purposes. For example, an ask to inject TCP resets into active flows based on idle timeouts and other parameters necessitated VFP being aware of the latest valid sequence numbers of every flow.
- We created a number of new packet forwarding and duplication actions, for example supporting tap interfaces with their own encapsulation and SDN policy, complex forwarding actions for offloading gateways and software routers to hardware, and multicast semantics using fast hardware replication on a unicast underlay.
- We added SDN actions such as NAT46 with custom translation logic for our internal workloads, support for virtualizing RDMA, and new overlay header formats.
- As we saw pressure to improve connection setup performance, we repeatedly iterated on our offload path, moving many functions such as hashing and table insertion of flows from software into hardware over time based on production telemetry.
- We've used the FPGAs to add constant new datapath diagnostics at line rate, including programmable packet captures, packet tracing through stages in our FPGA for latency and correctness analysis, and extensive counters and telemetry of the kind that require support in hardware in the datapath. This is our most constant source of iteration.

8.3 Lessons Learned

Since beginning the Azure Accelerated Networking project, we learned a number of other lessons of value:

- **Design for serviceability upfront**. The topics in Section 7 were the hardest of anything here to get right. They worked only because the entire system, from hardware to software to VM integration, were designed to be serviceable and monitorable from day 1. Serviceability cannot be bolted on later.
- **Use a unified development team**. If you want Hardware/Software co-design, hardware devs should in the same team as software devs. We explicitly built our hardware team inside the existing Azure host networking group, rather than the traditional approach of having separate groups for HW and SW, to encourage frequent collaboration and knowledge sharing.
- **Use software development techniques for FPGAs**. One thing that helped our agility was viewing the host networking datapath stack as a single stack and ship vehicle across VFP and FPGA, reducing complex rollout dependencies and schedule mismatch. As much as possible, we treated and shipped hardware logic as if it was software. Going through iterative rings of software qualification meant we didn't need ASIC-levels of specification and verification up front and we could be

more agile. A few minutes in a live environment covers far more time and more scenarios than typical RTL verification flows could ever hope to cover.

- **Better perf means better reliability**. One of the biggest benefits of AccelNet for VMs is that the network datapath no longer shares cores or resources with the rest of the host, and is not subject to transient issues — we've seen much more reliable performance and lower variance as a result.
- **HW/SW co-design is best when iterative**. ASIC development traditionally means designing a specification and test methodology for everything that a system could possibly want to do over its lifetime upfront. FPGAs allowed our hardware developers to be far more agile in their approach. We can deploy designs directly to customers, collect data from real workloads with detailed counters, and use those to decide what functions should be in hardware vs. software in the next release, and where performance bottlenecks are. More importantly, we can allow the specifications to evolve constantly throughout the development process. For example, we changed the hashing and caching strategies several times after the initial release.
- **Failure rates remain low**, and were in line with other passive parts in the system, with the most frequently failing part being DRAM. Overall the FPGAs were reliable in datacenters worldwide.
- **Upper layers should be agnostic of offloads**. Because VFP abstracted out whether SDN policy was being offloaded or not from controllers and upper layers, AccelNet was much less disruptive to deploy.
- **Mitigating Spectre performance impact**. In the wake of the Meltdown and Spectre attacks on our CPUs, CPU-based I/O was impacted by common mitigations [29]. Because AccelNet bypasses the host and CPUs entirely, our AccelNet customers saw significantly less impact to network performance, and many redeployed older tenants to AccelNet-capable hardware just to avoid these impacts.

9 Related Work

Since we first deployed Azure SmartNICs and announced them at the Open Networking Summit in 2015, we've seen numerous different programmable NIC solutions with vSwitch offload come to market (recently many of these are labeled as "Smart NICs" too). Most follow the trends we discussed in 4.1. Some [30] are based on ASICs with internal match-action tables — these tend to not be very flexible or support some of the more advanced actions we've implemented over time in GFT, and give little room for growth as actions and policy change. Others [13, 14] do datapath processing entirely in embedded cores, either general purpose CPUs or network-specific ones, but we've found the performance of this model is not great and we don't see a path to scale this to 100G and beyond without requiring many cores. A newer trend is to combine an ASIC supporting some match-action function with a small SoC supporting a DPDK-style datapath for on-core packet processing. But we don't ultimately see that this solves the dilemma of ASICs vs CPUs — if you have a widely-applied action that the ASIC can't handle, you have to send all your packets up to the CPUs, and now your CPUs have to handle line rate processing.

Others [31] show that they can improve the performance of software stacks entirely in the host and suggest burning cores to do host SDN. While we believe this in practice requires multiple cores at line rate for our workloads, in IaaS even taking a very small number of cores is too costly for this to make sense for us, and the performance and latency aren't optimal. With FPGAs, we've found we're able to achieve sufficient programmability and agility in practice. Offloading functionality to the switches as in [32] was also explored, but since we have to store complex actions for every TCP connection in our system, and with the increase of VM and container density on a node, we found the min set of policy needed to be offloaded, even when reasonably compressed, is at least 2 orders of magnitude more than even the latest programmable switch ASICs can store in SRAM - at NIC speeds we can scale out to GBs of DRAM.

Another recent suggestion is to use P4 engines [33], thus far mostly implemented in switches, to create SmartNICs. The P4 specification offers flexible parsing and relatively flexible actions, many of which are similar to GFT. In fact, P4 could potentially serve as a way to specify some of the GFT processing flow. However, there are other SDN functions outside the scope of existing P4 engines and even the P4 language specification that are important for us to implement in AccelNet — functions such as scheduling, QoS, background state updates, any kind of programmable transport layer, and a variety of complex policies outside the scope of simple packet transformations. While we expect the P4 language to be extended to include many of these, using a programmable fabric like an FPGA to implement GFT or P4 functionality remains a good choice given the evolving nature of the SDN and cloud space. We expect much of the functionality outside of the core packet processor to harden over time, but expect SDN to remain soft for the foreseeable future.

10 Conclusion and Future Work

We detailed the Azure SmartNIC, our FPGA-based programmable NIC, as well as Accelerated Networking, our service for high performance networking providing cloud-leading network performance, and described our experiences building and deploying them.

This paper describes primarily functions we were already doing in software in host SDN and offloaded to hardware for great performance. Future work will describe entirely new functionality we've found we can support now that we have programmable NICs on every host.

Acknowledgements

We thank our shepherd Arvind Krishnamurthy and the anonymous reviewers for their many helpful comments that greatly improved the final paper.

Azure Accelerated Networking represents the work of many engineers, architects, product managers, and leaders across Azure and Microsoft Research over several years, more than we can list here. We thank the entire Azure Networking, Server Infrastructure, and Compute teams for their support in developing, iterating on, and deploying SmartNICs and the Accelerated Networking service.

We thank Parveen Patel, Pankaj Garg, Peter Carlin, Tomas Talius, Jurgen Thomas, and Hemant Kumar for their invaluable early feedback in deploying preview versions of our service, and KY Srinivasan, Stephen Hemminger, Josh Poulson, Simon Xiao, and Haiyang Zhang for supporting our Linux VM ecosystem's move to AccelNet. Finally, in addition to our leaders in the author list, we thank Yousef Khalidi, Mark Russinovich, and Jason Zander for their support.

References

[1] Microsoft Azure. `http://azure.microsoft.com`, 2018.

[2] Amazon. Amazon Web Services. `http://aws.amazon.com`, 2018.

[3] Google. Google Cloud Platform. `http://cloud.google.com`, 2018.

[4] Microsoft. Overview of Single Root I/O Virtualization (SR-IOV). `https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-`, Apr 2017.

[5] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with sr-iov. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–10, Jan 2010.

[6] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, 2017. USENIX Association.

[7] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Gopi Prashanth, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[8] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[9] Microsoft. TCP/IP Offload. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/tcp-ip-offload`, Apr 2017.

[10] Microsoft. Introduction to Receive Side Scaling. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling`, Apr 2017.

[11] Microsoft. Virtual Machine Queue (VMQ). `https://msdn.microsoft.com/en-us/windows/hardware/drivers/network/virtual-machine-queue--vmq-`, Apr 2017.

[12] Microsoft. Network Virtualization using Generic Routing Encapsulation (NVGRE) Task Offload. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/network-virtualization-using-generic-routing-encapsulation--nvgre--task-offload`, Apr 2017.

[13] Cavium. Cavium LiquidIO II Network Appliance Smart NICs. `http://www.cavium.com/LiquidIO-II_Network_Appliance_Adapters.html`.

[14] Netronome. Open vSwitch Offload and Acceleration with Agilio CX SmartNICs. `https://www.netronome.com/media/redactor_files/WP_OVS_Benchmarking.pdf`.

[15] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 249–261, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Yatish Turakhia, Kevin Jie Zheng, Gill Bejerano, and William J. Dally. Darwin: A hardware-acceleration framework for genomic sequence alignment. *bioRxiv*, 2017.

[17] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Christian Boehn, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Tamas Juhasz, Ratna Kumar Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Steve Reinhardt, Adam Sapek, Raja Seera, Balaji Sridharan, Lisa Woods, Phillip Yi-Xiao, Ritchie Zhao, and Doug Burger. Accelerating Persistent Neural Networks at Datacenter Scale. In *Hot Chips 27*, 2017.

[18] DPDK. DPDK: Data Plane Development Kit. `http://dpdk.org/about`, 2018.

[19] Gokhan Sayilar and Derek Chiou. Cryptoraptor: High throughput reconfigurable cryptographic processor. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, pages 154–161, Piscataway, NJ, USA, 2014. IEEE Press.

[20] J. Fowers, J. Y. Kim, D. Burger, and S. Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 52–59, May 2015.

[21] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 523–536, New York, NY, USA, 2015. ACM.

[22] Microsoft. Server/ProjectOlympus. `www.opencompute.org/wiki/Server/ProjectOlympus`, 2018.

[23] P. P. Deepthi and P. S. Sathidevi. Design, implementation and analysis of hardware efficient stream ciphers using lfsr based hash functions. *Comput. Secur.*, 28(3-4):229–241, May 2009.

[24] Microsoft. RSS Hashing Functions. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/rss-hashing-functions`, Apr 2017.

[25] Microsoft. Registered Input/Output (RIO) API Extensions. https://technet.microsoft.com/en-us/library/hh997032(v=ws.11).aspx, Aug 2016.

[26] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. CUBIC for Fast Long-Distance Networks. RFC 8312, February 2018.

[27] Messaging Accelerator (VMA). https://github.com/Mellanox/libvma, 2018.

[28] Microsoft. ExpressRoute overview. https://docs.microsoft.com/en-us/azure/expressroute/expressroute-introduction, Oct 2017.

[29] Microsoft. Securing Azure customers from CPU vulnerability. https://azure.microsoft.com/en-us/blog/securing-azure-customers-from-cpu-vulnerability/, 2018.

[30] Chloe Jian Ma and Erez Cohen. OpenStack and OVS: From Love-Hate Relationship to Match Made in Heaven. https://events.static.linuxfound.org/sites/events/files/slides/Mellanox%20OPNFV%20Presentation%20on%20OVS%20Offload%20Nov%2012th%202015.pdf, Nov 2012.

[31] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an openflow switch on the netfpga platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 1–9, New York, NY, USA, 2008. ACM.

[32] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. ACM.

[33] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

zkLedger: Privacy-Preserving Auditing for Distributed Ledgers

Neha Narula
MIT Media Lab

Willy Vasquez
University of Texas at Austin*

Madars Virza
MIT Media Lab

Abstract

Distributed ledgers (e.g. blockchains) enable financial institutions to efficiently reconcile cross-organization transactions. For example, banks might use a distributed ledger as a settlement log for digital assets. Unfortunately, these ledgers are either entirely public to all participants, revealing sensitive strategy and trading information, or are private but do not support third-party auditing without revealing the contents of transactions to the auditor. Auditing and financial oversight are critical to proving institutions are complying with regulation.

This paper presents zkLedger, the first system to protect ledger participants' privacy and provide fast, provably correct auditing. Banks create digital asset transactions that are visible only to the organizations party to the transaction, but are publicly verifiable. An auditor sends queries to banks, for example "What is the outstanding amount of a certain digital asset on your balance sheet?" and gets a response and cryptographic assurance that the response is correct. zkLedger has two important benefits over previous work. First, zkLedger provides fast, rich auditing with a new proof scheme using Schnorr-type non-interactive zero-knowledge proofs. Unlike zk-SNARKs, our techniques do not require trusted setup and only rely on widely-used cryptographic assumptions. Second, zkLedger provides *completeness*; it uses a columnar ledger construction so that banks cannot hide transactions from the auditor, and participants can use rolling caches to produce and verify answers quickly. We implement a distributed version of zkLedger that can produce provably-correct answers to auditor queries on a ledger with a hundred thousand transactions in less than 10 milliseconds.

1 Introduction

Institutions engage trusted third-party auditors to prove that they are complying with laws and regulation. Traditionally this is done by auditing companies like Deloitte, Pricewaterhouse Coopers, Ernst and Young, and KPMG (known as the "Big Four"), which together audit 99% of the companies in the S&P 500 [19]. This type of auditing is laborious and time-consuming, so regulators and investors do not get real-time access to information about the financial status of institutions. In addition, trusted third parties can make mistakes. The most well-known example of this is the collapse of Arthur Anderson in 2002, after it failed to catch Enron's $100 billion accounting fraud.

Recently, financial institutions are exploring distributed ledgers (or blockchains) to reduce verification and reconciliation costs in an environment with multiple distrusting parties. Distributed ledgers enable real-time validation by all participants (known as *public verifiability*), but at the cost of privacy—every participant must download all transactions in order to verify their integrity. This is untenable for institutions that rely on secrecy to protect strategy and intellectual property (e.g. trading strategies), and for organizations that have to comply with laws and regulation around data privacy (for example, the General Data Protection Regulation in Europe [24]).

Distributed ledgers that support privacy generally operate in one of two ways: either by only committing to hashes of transactions on the ledger, using trusted third parties to independently verify transactions [22, 23], or by using cryptographic commitment schemes to hide the content of transactions [17, 42, 47, 51]. The former class of ledgers suffers from the fact that participants can no longer verify the integrity of private transactions, eliminating the distributed ledger benefit. The latter class still has public verifiability, but either reveals the transaction graph [17, 42] or requires trusted setup, which, if compromised, would let an adversary undetectably create new assets [47, 51]. None of the existing privacy-preserving distributed ledgers offer an important property for real-world systems—efficient auditing.

This paper presents zkLedger, the first distributed ledger system to support strong transaction privacy, public verifiability, and practical, useful auditing. zkLedger provides strong transaction privacy: an adversary cannot tell who is participating in a transaction or how much is being transacted, and crucially, zkLedger does not reveal the *transaction graph*, or linkages between transactions. The time of transactions and the type of asset being transferred are public. All participants in zkLedger can still

* Work completed at the MIT Media Lab.

† Source code and full version of the paper: zkledger.org.

verify transactions are maintaining important financial invariants, like conservation of assets, and an auditor can issue a rich set of auditing queries to the participants and receive answers that are provably consistent with the ledger. zkLedger supports a useful set of auditing primitives including sums, moving averages, variance, standard deviation, and ratios. An auditor can use these primitives to measure financial leverage, asset illiquidity, counterparty risk exposures, and market concentration, for the system as a whole or for individual participants.

A set of banks might use zkLedger to construct a settlement log for an over-the-counter market trading digital assets. In these markets, buyers and sellers are matched via electronic exchanges, trades are frequent and fast settlement helps lower counterparty risk. Once a trade is confirmed, a bank can initiate the transfer of the asset as a transaction in zkLedger, which, when accepted in the ledger, settles the transaction. Each bank stores plain-text transaction data in its own private datastores. In zkLedger, instead of storing plain-text transactions, participants store value *commitments* on the distributed ledger. Importantly, these commitments can be homomorphically combined. A bank can prove to an auditor how much of an asset it has on its balance sheet by opening up the product of all transaction commitments it has referencing that asset. The auditor can confirm that the opened product is consistent with the product of the commitments on the ledger.

Designing zkLedger required overcoming three key challenges:

Providing privacy and auditing. The first challenge is to preserve privacy while still allowing an auditor to compute provably correct measurements over the data in the ledger. zkLedger is the first system to simultaneously achieve this, by combining several cryptographic primitives. To hide values, zkLedger uses Pedersen commitments [41]. Pedersen commitments can be homomorphically combined, so a verifier can, for example, confirm that the sum of the outputs is less than or equal to the sum of the inputs, conserving assets. More than that, an auditor can combine commitments to compute linear combinations of values in different rows in the ledger. Previous confidential blockchain systems also use Pedersen commitments to hide values but end up revealing linkages between transactions, and do not support private auditing [17, 34, 42].

zkLedger uses an interactive *map/reduce* paradigm over the ledger with non-interactive zero-knowledge proofs (NIZKs) to compute measurements that go beyond sums. These are Generalized Schnorr Proofs [48], which are fast and rely only on widely accepted cryptographic assumptions. Banks can provably recommit to functions over values in the ledger, such as $f: v \rightarrow v^2$, which lets the auditor compute measurements like variance, skew, and outliers without revealing individual transaction details.

Auditing completeness. Since an auditor cannot determine who was involved in which transactions, zkLedger must ensure that during auditing, a participant cannot leave out transactions to hide assets from the auditor. We call this property *completeness*. At the same time, we do not want to reveal to the auditor who was involved in which transactions. zkLedger uses a novel table-construction in the ledger. A transaction is a row which includes an entry for every participant, and an empty entry is indistinguishable from an entry involving a transfer of assets. All of a participant's transfers are in its column in the ledger. An auditor audits *every* transaction when auditing a participant, meaning a participant cannot hide transactions. This presents efficiency challenges, which zkLedger addresses by using *commitment caches* and *audit tokens*, described below.

Efficiency. The third challenge is supporting all of this efficiently. zkLedger implements a number of optimizations: every participant and the auditor keeps *commitment caches*, which are rolling products of every participants' column in the ledger; this makes it fast to generate asset proofs and to answer audits. To reduce communication costs, zkLedger is designed so that participants do not have to interact to construct the proofs for the transaction; the spender can create the transaction alone (this is similar to how other blockchain systems work). But a malicious spender could try to encode incorrect values in the commitments for other banks—we must ensure all of the commitments and proofs are correct and that every participant has what they need to later respond to an audit. To do this, we designed a set of proofs that everyone can publicly verify—transactions with incorrect proofs will be ignored. These proofs ensure that every participant has an *audit token*, which they can use to later open up commitments for that row, and that all proofs and commitments are *consistent*. The audit token and the consistency proofs are publicly verifiable, but do not leak any transaction information. They are also non-interactive, so zkLedger makes progress even if banks cannot communicate, and they are encoded for a specific bank, so a token for one bank cannot be used by another bank to lie to the auditor.

The slowest part of transaction creation and validation are range proofs, which ensure that an asset's value is in a pre-specified range, and prevent a malicious attacker from undetectably creating new assets. Range proofs are $10\times$ the size of the other proofs and take $5\times$ as much time to prove and verify. A naive implementation of

zkLedger might require multiple range proofs, but by using disjunctive proofs, we can multiplex different values into one range proof per entry.

In summary, the contributions of this paper are:

- zkLedger, the first distributed ledger system to achieve strong privacy and complete auditing;
- a design combining fast, well-understood cryptographic primitives using audit tokens and map/reduce to compute provably correct answers to queries;
- an evaluation of zkLedger showing efficient transaction creation and auditing; and
- an analysis of the types of queries zkLedger can support, suggesting that zkLedger can efficiently handle a useful set of auditing measurements.

2 Related Work

zkLedger is related to work in auditing or computing on private data and privacy-preserving blockchains. zkLedger achieves fast, provably correct auditing by creating a new distributed ledger table model and applying a new scheme using zero-knowledge proofs.

2.1 Computing on Private Data

Previous work proposed a multi-party computation scheme in which participants use a secure protocol to compute the results of functions which answer questions about systemic financial risk, the same problem which zkLedger aims to address [3, 10], and network security [14]. This work provides privacy benefits over existing analytics systems by allowing participants to keep their data secret. However, it only supports overall system auditing, it is not a solution to audit individual participants. There is also nothing preventing participants from lying in the inputs to the multi-party computation; they do not achieve completeness.

Provisions [21] is a way for Bitcoin exchanges to prove they are solvent without revealing their total holdings. Provisions uses Proof of Assets and Proof of Liabilities, which are very similar to the zero-knowledge proofs we use in zkLedger. However, in Provisions, an exchange could “borrow” private keys from another Bitcoin holder and thus prove assets they do not actually hold; in fact multiple exchanges could share the same assets. Moreover, Provisions does not provide completeness. By using a columnar construction with a distributed ledger, zkLedger achieves completeness.

In Prio [18], untrusted servers can compute privately on mobile client data. Prio does not operate on distributed ledgers, and thus does not guarantee public verifiability. Prio requires all servers to cooperate in order for client proofs to validate; zkLedger can tolerate non-cooperating participants.

Several systems provide private and correct computing using trusted hardware [4–6, 49, 52]. In our setting, we cannot guarantee that all participants will trust the same hardware provider. In addition, it would be a conflict of interest to use such a system to audit the company providing the trusted hardware.

There are many systems which compute on encrypted data to protect user confidentiality in the event of a server compromise [25, 31, 32, 40, 43, 44, 50]. These systems address a different problem than what zkLedger is trying to solve. Instead, we provide interactive, provably correct auditing over private data generated by many parties.

2.2 Privacy-preserving blockchains

Bitcoin, a decentralized cryptocurrency released in 2009, was the first blockchain [37]. Many companies have explored using a blockchain to record the transfer of assets. These systems are marked by the following characteristics: (1) Multiple, possibly distrusting participants, all with write permissions and no single point of failure or control; (2) A consensus protocol to construct an append-only, globally ordered log with a chain of hashes to prevent tampering with the past; and (3) Digitally signed transactions to indicate intent to transfer ownership.

In Bitcoin and most other blockchains, all transactions are public: every participant receives each transaction, and can verify all the details. Users create pseudonyms by generating one-time use public keys for payment addresses, but transaction amounts and the links between transactions are still globally visible. Confidential Transactions [34] and Confidential Assets [17, 42] are extensions to Bitcoin which blind the assets and amounts in transactions while still ensuring that all participants can validate transactions. Though these systems hide assets and amounts, they leak the transaction graph and do not support private auditing—an auditor would require access to *all* plain-text transactions in order to ensure completeness. The transaction graph alone leaks substantial information [36, 38, 45, 46]; for example, the FBI followed linked transactions to trace bitcoins and used this as evidence in court [28]. zkLedger provides stronger transaction privacy and private auditing, but at the cost of scalability. Transactions in zkLedger are sized order the number of participants in the whole system, requiring more time to produce and verify as the number of participants grows. This makes zkLedger more suitable to ledgers with fewer participants who require more privacy.

Solidus [16] is a distributed ledger system that uses Oblivious RAM to hide the transaction graph and trans-

action amount between bank customers. While this construction also provides private transactions, Solidus can only support auditing by revealing all of the keys used in the system to an auditor, and opening transactions. zkLedger achieves performance similar to Solidus while providing private auditing.

R3's Corda [22], and Digital Asset Holding's Global Synchronization Log (GSL) [23] are distributed ledgers geared towards financial institutions that rely on trusted third parties to pass through information. In Corda, notaries verify transactions and maintain privacy of participants, while GSL segments its ledger, only storing a hash of the values globally and limiting access to fine-grained transaction data. Neither support private auditing.

Another approach is that of Zerocash [47], and its related implementation Zcash [51], an anonymous cryptocurrency based on Bitcoin. Zerocash uses zk-SNARKs [7] to hide transaction amounts, participants, and the transaction graph. The zk-SNARKs as used in Zcash can be extended to handle policies to enforce regulations, KYC/AML laws, and taxes [27]. These policies do not support arbitrary queries, but instead put limits on the new types of transactions that can take place. These ideas have not yet been implemented in a practical system.

zk-SNARKs are quite efficient for some statements but unfortunately, the price of this efficiency is paid in setup assumptions: as of now, all concretely efficient zk-SNARKs require a trusted third party for setup. The consequences of incorrect or compromised setup are potentially disastrous: an adversary who can learn the secret randomness used during setup can make fraudulent proofs of false statements that are indistinguishable from proofs of true statements. In our setting (international banking), such proofs would permit unrestricted creation or destruction of financial assets or liabilities. There may not even be a viable party to perform the one-time trusted setup. For example, Russia might not trust the Federal Reserve or the European Central Bank, or it might not be politically expedient to be seen as doing so. While it is possible to mitigate this concern, e.g., by distributing the setup between multiple parties [8, 11, 12], this process is onerous and expensive. Ideally, the financial integrity of the system would not rely on trusted setup at all. We choose to base consensus-critical portions of zkLedger's design on standard NIZKs.

3 zkLedger Overview

3.1 Architecture

System participants. There are n participants which we call *banks* that issue transactions to transfer digital assets, $\mathsf{Bank}_1, \ldots, \mathsf{Bank}_n$ and an auditor Auditor, that verifies certain operational aspects of transactions performed by the banks (e.g. "is a particular bank Bank_i solvent?"). These roles are not distinct; a bank could also audit. A Depositor or set of Depositors can issue and withdraw assets from the system; for example, the European Central Bank might issue 1M € to Bank_i in the system. Issuance and withdrawal of all assets are controlled by the Depositors and are global, public events.

Transactions. Banks exchange assets by creating *transfer* transactions, whose details are hidden. A transfer transaction captures an event where Bank_i is transferring v shares of asset t to Bank_j. Our scheme supports a bank transferring to multiple other banks, but for simplicity we assume there is one spending and one receiving bank in each transaction. Banks determine the details of a transfer transaction outside of zkLedger, perhaps through an exchange. We assume they use encrypted channels.

Append-only ledger. Banks submit transactions to an append-only ledger, which globally orders all valid transactions. If a digital asset only exists on the ledger, then transfer on the ledger *is* change in legal custody of the digital assets, not merely a record of ownership change, and an Auditor is guaranteed a Bank is not hiding assets. This ledger could be maintained by a trusted third party, by the banks themselves, or via a blockchain like Ethereum or Bitcoin. Maintaining a fault-tolerant, globally ordered log is outside the scope of this paper, but can be done using standard techniques [15, 30, 39].

3.2 Cryptographic building blocks

Commitment schemes. To protect their privacy participant banks do not broadcast payment details, such as the transaction amount, in plain. Instead the banks post hiding *commitments* to the append-only ledger; in particular, zkLedger uses Pedersen commitments [41]. Let $\mathbb{G}$ be a cyclic group with $s = |\mathbb{G}|$ elements, and let g and h be two random generators of $\mathbb{G}$. Then a Pedersen commitment to an integer $v \in \{0, 1, \ldots, s-1\}$ is formed as follows: pick commitment randomness r, and return the commitment $\mathsf{cm} := \mathsf{COMM}(v, r) = g^v h^r$.

Pedersen commitments are perfectly hiding: the commitment cm reveals nothing about the committed value v. In a similar way, the commitments are also computationally binding: if an adversary can open a commitment cm in two different ways (for the same r, two different values v and v'), then the same adversary can be used to compute $\log_h(g)$ and thus break the discrete logarithm problem in $\mathbb{G}$. In zkLedger we choose $\mathbb{G}$ to be the group of points on the elliptic curve secp256k1.

A very useful property of Pedersen commitments is that they are *additively homomorphic*. If cm_1 and cm_2 are

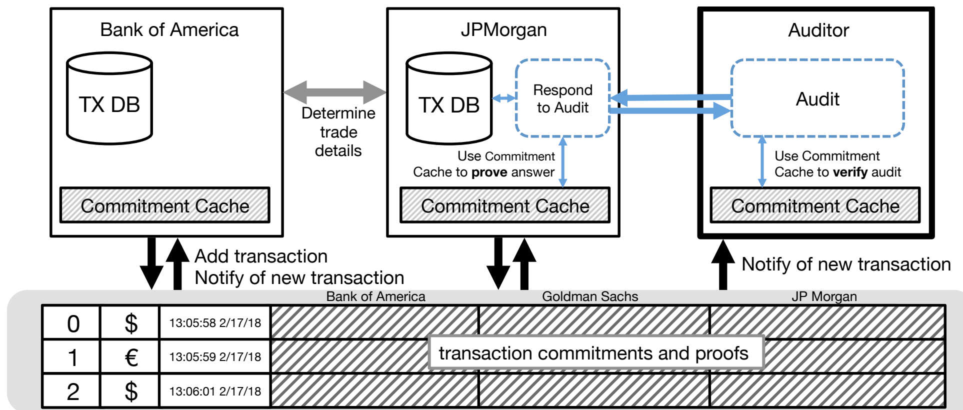


Figure 1: Overall zkLedger system design showing the interactions between the three main entities (banks, auditor, and the shared ledger) in our system. Each bank maintains private state, consisting of the transaction database for transactions the bank originated, and the bank's secret key.

two commitments to values v_1 and v_2, using commitment randomness r_1 and r_2, respectively, then $\mathsf{cm} := \mathsf{cm}_1 \cdot \mathsf{cm}_2$ is a commitment to $v_1 + v_2$ using randomness $r_1 + r_2$, as $\mathsf{cm} = (g^{v_1}h^{r_1}) \cdot (g^{v_2}h^{r_2}) = g^{v_1+v_2}h^{r_1+r_2}$. To speed up transaction generation and auditing zkLedger makes extensive use of the ability to additively combine commitments.

Public-key encryption. Every bank i also generates a Schnorr signature keypair keypair consisting of a secret key sk_i and public key $\mathsf{pk}_i := h^{\mathsf{sk}_i}$, and distributes the public key pk_i to all other system participants.

Non-interactive zero-knowledge proofs. To make privacy-preserving assertions about payment details zkLedger relies on non-interactive zero-knowledge proofs (NIZKs) [9]. In brief, zero-knowledge proofs concern two parties: the prover, who holds some private data, and the verifier, who wishes to be convinced of some property about this private data. For example, the prover might know the opening of a commitment cm, and wish to convince the verifier that the committed value v is in some range, e.g., $0 \leq v < 10^6$. Using NIZKs, the prover can produce a binary string π, the proof, that simultaneously persuades the verifier, yet does not reveal anything else about v. Verifying π does not require any interaction between the prover and the verifier, and the prover can append π to the ledger, where it can be verified by any party of the system.

In theory, NIZK proof systems exist for all properties in NP whereas the practical feasibility of NIZKs is highly-dependent on the complexity of the property at hand. In particular, algebraic properties in cyclic groups, such as, knowledge of discrete logarithm, equality of values committed in Pedersen commitments, or similar have very efficient NIZK proof systems. The design of zkLedger is carefully structured so that all NIZK proofs have particularly efficient constructions.

3.3 Security Goals

The goals of zkLedger are to hide the amounts, participants, and links between transactions while maintaining a verifiable transaction ledger, and for the Auditor to receive *reliable* answers to its queries. Specifically, zkLedger lets banks issue hidden transfer transactions which are still publicly verifiable by all other participants; every participant can confirm a transaction conserves assets and assets are only transferred with the spending bank's authority. For example, if Bank_i transfers 10,000 € to Bank_j, both the banks and amount are hidden. The asset (€) and time of the transaction are not hidden. zkLedger also hides the transaction graph, meaning which previous transaction(s) supplied the 10,000 € to Bank_i in the first place.

An Auditor can query a Bank about its contents on the ledger, for example "How many euros does Bank_j hold?" A bank should be able to produce commitments which will convince the auditor that the bank's answer to the auditing query is correct, meaning consistent with the transactions on the ledger. zkLedger ensures that if a bank gives the Auditor an answer that is inconsistent with the ledger, the Auditor will catch such attempt of cheating with high probability (and of course, a trustworthy answer must always be accepted).

3.4 Threat model

zkLedger does not assume that banks will behave honestly—they can attempt to steal assets, hide assets, manipulate their account balances, or lie to auditors. We assume banks can arbitrarily collude. zkLedger keeps the amounts and participants of transactions private as long as neither the spender nor receiver in a transaction collude with an observer, like the Auditor. We assume that the ledger does not omit transactions and is available. zkLedger does not protect against an adversary who observes traffic on the network; for example, if only two banks are exchanging messages, it's reasonable to assume the transactions in the ledger involve those banks. Nothing

beyond what is necessarily leaked by an audit is revealed. However, frequent auditing might reveal transaction contents; e.g. if an auditor asks for banks' assets after every transaction.

4 Design

The challenge in creating zkLedger is to practically support complete, confidential auditing—an Auditor should not be able to see individual bank transactions, but a Bank should not be able to hide assets from the Auditor during an audit, and the auditor should be able to detect an incorrect answer.

Figure 1 shows a general overview of zkLedger. There are banks which determine transactions out of band and then settle them by appending transactions to the ledger. The ledger makes sure all banks and any auditors see new transactions. Each bank and auditor maintains a *commitment cache*, which are commitments to summed values used to make creating transactions and responding to audits faster. Each bank also has private stores of plain-text transaction data.

The rest of this section describes the zkLedger transaction format, how banks create transactions, and how a bank can answer a simple query from the auditor.

4.1 Transactions

The ledger in zkLedger is a table where transactions correspond to *rows*, and Banks correspond to *columns*. Each transaction has an entry for each Bank. Figure 2 shows a ledger with n banks. Each entry in a transaction includes a commitment to a value which is the amount of the asset that is being debited or credited to the bank. For example, if Bank_i wants to transfer 100 shares of an asset to Bank_j, i's entry in the transaction would contain a commitment to -100 and j's would contain a commitment to 100. All other entries in the transaction would contain commitments to 0, since none of the other bank balances were changed. This scheme has the nice property that an outside observer can look at a bank's entire column and know that this represents the entirety of the bank's holdings.

Hiding amounts. As described in §3.2, zkLedger does not include the value in plain-text in the transaction. Instead, zkLedger uses Pedersen commitments to commit to the value in transfer transactions. This makes value commitments completely indistinguishable—an outside observer cannot tell the difference between a commitment to a positive value, a negative value, or 0. Recall that a commitment to a value v is $\mathsf{cm} := \mathsf{COMM}(v,r) = g^v h^r$. If desired, a prover can reveal v and r to a verifier who knows cm and the verifier can confirm this is consistent.

Since a transaction in zkLedger contains an entry for every Bank, there is a size-n vector $\vec{\mathsf{cm}}$ committing to values in $\vec{v}$. Each commitment cm_k uses a fresh commitment randomness r_k. Most of the entries will contain commitments to 0, for banks that are not involved with the transaction, but this is not apparent to an outside observer.

zkLedger maintains the following financial invariants:

- A transfer transaction cannot create or destroy assets
- The spending bank must give consent to the transfer and must actually own enough of the particular asset to execute this transaction

In a public blockchain, the validators could simply confirm that these things are true by looking at the history of transactions and the current transaction, and making sure the spending bank has the funds to spend. However, in zkLedger these values are not public. Instead, we create a set of proofs that the spender can create to prove the invariants are maintained. The spender can create a transaction without interacting with any of the other banks.

First, zkLedger introduces a *Proof of Balance* (π^{B}). This is a proof that the transaction conserves assets; no assets are created or destroyed (of course, public issuance and withdrawal transactions do not have such proofs). More formally, the committed values should satisfy $\sum_{k=1}^{n} v_k = 0$. To prove this, the prover chooses the r_k carefully: it should also be the case that $\sum_{k=1}^{n} r_k = 0$. If this is true and the values also sum to 0, then the verifier can check to make sure that $\prod_{k=1}^{n} \mathsf{cm}_k = 1$ for the commitments in the row.

Next, zkLedger must ensure that the spending bank actually has the assets to transfer. To do this, zkLedger introduces a *Proof of Assets* (π^{A}). Other privacy-preserving blockchain systems use *Unspent Transaction Outputs*, or UTXOs, to show proof of assets and prevent double spending. For example, if Alice wants to send a coin to Bob, she chooses one of her coins, creates a new transaction addressing the coin to Bob, and includes a pointer to the previous transaction where she received the coin. This previous transaction is an *output*. All of the validators in the system maintain the invariant that outputs can only be spent once. Unfortunately, in systems without zk-SNARKs, this leaks the transaction graph. In zkLedger, a bank proves it has assets by creating a commitment to the *sum* of the values for the asset in its column, including this transaction. If the sum is greater than or equal to 0, then the bank has assets to transfer. Note that this is true since the bank's column represents all the assets it has received or spent, and the Pedersen commitments can be homomorphically added in columns as well as in rows. In

order to produce a proof with the correct sum, the bank must have seen every previous transaction. This implies that banks must create transactions *serially*. In its own entry where the value is negative, the bank includes proof of knowledge of secret key to show that it authorized the transaction. This requires creating a disjunctive proof—either the committed value for entry i has $v_i \geq 0$, or the creator of the transaction knows the secret key for Bank_i.

Range proofs. Because commitment values are in an elliptic curve group and rely on modulus, we need to make sure that the commited values are within an acceptable range. To see why, note that if N is the order of the group, then $\mathsf{COMM}(v,r) = \mathsf{COMM}(v+N,r)$; there is no way to distinguish between the two. Without a check to make sure the commited value is within the range $[0, N-1]$, a malicious bank could undetectably create assets. To address this, we use range proofs as described in Confidential Assets [42], which uses Borromean ring signatures [35]. zkLedger supports asset value amounts up to a trillion. Range proofs are the most expensive part of the transaction; as described, our scheme requires two range proofs—one for the commitment value, and another for the sum of assets in the column. We can squash the two range proofs down to one range proof by introducing an auxiliary commitment, cm'_i. cm'_i is *either* a re-commitment to the value in cm_i or the sum of the values in the column up to row m, $\sum_{k=0}^{m} v_k$, which can be achieved by computing the product of the commitments in the column, $\prod_{k=0}^{m} \mathsf{cm}_k$. Then, we can do one range proof on the value in cm'_i. Either this is the spending bank, in which case cm'_i must be a commitment to the sum, or it is another bank which is receiving funds or not involved, in which case cm'_i could be either (and it does not matter which it is).

This satisfies the financial invariants described above. However, a particular design choice we made in zkLedger is that a spending bank can create a transaction spending its own assets without interacting with other banks. This means that a malicious bank could create transactions which maintain financial invariants but are ill-formed. We will address this problem after describing how auditing works.

Once created, a bank broadcasts the transaction, and it will be appended to the ledger. If the banks are maintaining the ledger, each bank is responsible for validating the transaction before accepting it to the ledger. If a third party is maintaining the ledger, then the third party should verify the proofs in a transaction before accepting it.

Example transaction. Figure 2 shows a transaction where Bank of America is transferring one million euros to Goldman Sachs. Bank of America creates the transfer transaction, publishing the transaction id, timestamp, and asset type (euros) publicly. Bank of America commits to the amount deducted from its own assets, $-1,000,000$, in its own entry and $1,000,000$ in Goldman Sachs's entry. For every other bank, Bank of America commits to 0. This serves to hide the banks involved in the transfer; no one except Bank of America can distinguish between the commitments to determine which are commiting to nonzero values. Bank of America then broadcasts the transaction to the ledger. The ledger maintainer validates the transaction and appends it to the ledger. Once accepted to the ledger, this serves as a complete transfer of $1,000,000$ euros from Bank of America to Goldman Sachs.

4.2 Auditing Protocol

The $\mathsf{Auditor}$ has a copy of the ledger and interacts with the banks to calculate functions on their private data, in order to get a view of the financial system represented by the ledger.

The $\mathsf{Auditor}$ audits Bank_i by issuing a query to Bank_i, for example, "How many euros do you hold at time t?". Bank_i responds to the auditor with an answer and a proof that the answer is consistent with the transactions on the ledger. The $\mathsf{Auditor}$ multiplies the commitments on the ledger in Bank_i's column for euros, and verifies the proof and answer with the total. This is a commitment to the the total amount of euros Bank_i holds.

The key insight here is that given this table construction, the $\mathsf{Auditor}$ can read bank i's column and know that it is seeing every asset transfer involving i. There is no way for i to "hide" assets on the ledger without *actually transferring assets* and giving control to another bank. In contrast, during a traditional audit, a bank could simply not show the auditor some of its balance sheet.

Banks could collude to hold assets for each other temporarily; for example Bank_j might transfer assets to Bank_i and take them back later. For that time period, the assets would be part of Bank_i's holdings. But banks cannot collude after an $\mathsf{Auditor}$ poses a query because the $\mathsf{Auditor}$ has already specified the time t at which the query applies. Any transfer would necessarily have to be after t. So at this point, it is too late for a malicious bank to create a new transaction transferring assets to another bank.

As described above, a bank can create a transaction transferring its own assets without any interaction. This is common with most blockchain systems, where only the signature of the sender is required to create a valid transaction. There is no in-protocol way for a receiver to object to a transfer. Given our table construction, every bank is affected by every transaction, because a bank must

Metadata			Bank$_1$	Bank$_2$	Bank$_3$	...	Bank$_n$
ID	Asset	Time	(Bank of America)	(Goldman Sachs)	(JPMorgan)	...	(Citigroup)
1	€	13:06:01 2/17/18	$\mathsf{COMM}(-10^6, r_1)$ Token$_1$ $\pi_1^{\mathsf{A}}, \pi_1^{\mathsf{B}}, \pi_1^{\mathsf{C}}$	$\mathsf{COMM}(10^6, r_2)$ Token$_2$ $\pi_2^{\mathsf{A}}, \pi_2^{\mathsf{B}}, \pi_2^{\mathsf{C}}$	$\mathsf{COMM}(0, r_3)$ Token$_2$ $\pi_3^{\mathsf{A}}, \pi_3^{\mathsf{B}}, \pi_3^{\mathsf{C}}$	... ...	$\mathsf{COMM}(0, r_n)$ Token$_n$ $\pi_n^{\mathsf{A}}, \pi_n^{\mathsf{B}}, \pi_n^{\mathsf{C}}$

Figure 2: Contents of the ledger pertaining to a transaction that sends 10^6 euros from Bank$_1$ to Bank$_2$. Note that while asset type (euro) is visible as part of the metadata, the transaction amount (10^6 €) and participating institutions (Bank of America and Goldman Sachs) remain private. We use Pedersen commitments, so the commitment part of the row has values $g^{-10^6}h^{r_1}$, $g^{10^6}h^{r_2}$,and h^{r_3}, ..., h^{r_n}. Similarly, audit tokens pictured have values $(\mathsf{pk}_1)^{r_1}$, $(\mathsf{pk}_2)^{r_2}$, ... For each bank Bank$_i$, the corresponding column also includes proof-of-assets π_i^{A}, proof-of-balance π_i^{B}, and proof-of-consistency π_i^{C}.

total all of the commitments in its column to respond to the Auditor—even commitments for transactions in which it was not involved. A malicious Bank$_i$ could create a transaction and not inform the another Bank$_j$ of the r_j used in its entry, even if it is not transferring assets to Bank$_j$. Bank$_j$ would be unable to respond to the Auditor because it would not be able to open up the product of the commitments in its column.

In order to prove the integrity of a transfer transaction, zkLedger must ensure an additional invariant:

- All banks have enough information in the transaction to open up commitments for the Auditor

zkLedger does this by requiring the spending bank to include a publicly verifiable Token in every entry. This is defined as $\mathsf{Token}_k := (\mathsf{pk}_k)^{r_k}$. Bank$_k$ uses this token to open up the product of its commitments for the Auditor, without needing to know r_k.

Using audit tokens. Consider a query for a sum of values in a bank's column. One way of answering this query would be to reveal $\sum v_k$ and $\sum r_k$. Then the auditor would simply check that these plain values are consistent with the homomorphically computed value $\prod \mathsf{cm}_k = g^{\sum v_k} h^{\sum r_k}$.

However, a bank does not necessarily know all the commitment randomnesses r_k (in particular, these values are unknown for any transaction that the bank was not party to), so the naive approach does not work.

One approach could be to ask the preparer of the transaction (i.e. the sender) to encrypt r_k so that the non-participating bank Bank$_k$ can decrypt it. To prevent the sending bank from placing a "garbage" ciphertext on the ledger (and thus making Bank$_k$ fail the auditor's queries), one would need a zero-knowledge proof of consistency between the encrypted value and the commitment. Constructing a concretely efficient proof for this statement is non-trivial: in a nutshell, standard encryption schemes (e.g. ElGamal) embed plain-text in a group element, while Pedersen commitments would have this value in the exponent.

Our insight is that Bank$_k$ does not need to open $\sum r_k$ to prove that $\sum v_k$ is correct. Instead, suppose that Bank$_k$ wants to claim that $s = g^{\sum v_k} h^{\sum r_k}$ opens up to a value $\sum v_k$. To do so, the bank computes $s' = s/g^{\sum v_k} = h^{\sum r_k}$ and $t = \prod_k \mathsf{Token}_k = h^{\mathsf{sk} \sum r_k}$. Note that the auditor can also compute the values of s' and t from the ledger and the claimed answer $\sum v_k$.

It suffices for the bank to prove that $\log_{s'} t = \log_h \mathsf{pk}$. Observe that both logarithms evaluate to sk so a bank can produce this proof without knowing $\sum r_i$. Moreover, if this equation holds then $t^{1/\mathsf{sk}} = s' = s/g^{\sum v}$, but if the $\sum v$ was incorrect then knowledge of sk would reveal a linear relationship between g and h, which is ruled out by our security assumption.

To show that the r in the Token$_k$ is the same as the r in r_k, we require an additional *Proof of Consistency* (π^{C}). This is a zero-knowledge proof asserting that for each k the value r_k used to form cm_k and Token$_k$ is the same. (See Appendix B for details of how such proof is constructed.)

Note that audit tokens are only useful to the bank opening its commitment; though public, a malicious bank cannot use another bank's Token to successfully open an incorrect result or learn information about other bank's transactions.

4.3 Final transaction construction

For a transfer transaction in row m, each entry i contains the following items:

- **Commitment** (cm_i): ($g^{v_i}h^{r_i}$) a Pedersen commitment to the value we are transferring.
- **Audit Token** (Token$_i$): $(\mathsf{pk}_i)^{r_i}$. This is used to answer audits without knowing the randomness used in the commitment.
- **Proof of Balance** (π^{B}): a zero-knowledge proof asserting that the committed values satisfy $\sum_{k=1}^{n} v_k = 0$.
- **Proof of Assets** (π^{A}): a new commitment cm'_i, corresponding token Token$'_i$, and a zero-knowledge proof asserting that either cm'_i is a re-commitment of the value in cm_i or a recommitment to the sum of the values in $\prod_{j=0}^{m} \mathsf{cm}_j$, and cm'_i is in range $[0, 2^{40})$. If the committed value in cm_i is negative, the proof asserts bank i consented to the transfer.

- **Proof of Consistency** (π^C): two zero-knowledge proofs asserting the randomness used in cm_i and Token_i are the same, and the randomness used in $\mathsf{cm'}_i$ and Token'_i are the same. This is to prevent a malicious bank from adding data to the ledger that would stop another bank from being able to open its commitments for the Auditor.

Transactions may contain additional metadata in plaintext or not. For example, banks might want to include encrypted account numbers, addresses, or identifying information on behalf of a customer to satisfy the Travel Rule specified in the Bank Secrecy Act of 1970 [1]. zkLedger supports auditing over metadata in the transaction as well, but it does not have a way to publicly verify additional metadata.

4.4 Adding or removing banks

zkLedger can support dynamically adding or removing banks if done so publicly. The participants (or another authority) append a signed transaction to the ledger indicating which banks, and thus columns, should be added or removed. For example, to add a new bank to the ledger shown in Figure 2, the involved banks would append a transaction to the ledger indicating an intent to add Bank_{n+1}. From that point forward, all transactions should contain $n+1$ entries. The Proof of Assets for Bank_{n+1}'s entry in each transaction will start at the row where Bank_{n+1} was added. Similarly, if a bank is removed, later transactions should not include entries for that bank. Since all participants can see which banks were added or removed, they can adjust their proofs and verifications accordingly.

4.5 Optimizations

zkLedger employs several optimizations to make producing and verifying these proofs faster, and to support faster auditing. First, caching the product of the commitments in a bank's column improves auditing and proof creation speed. Each bank stores a rolling product of commitments by row and by asset so that it can quickly produce proofs of assets and answer queries from auditors. Using these caches, a bank can quickly answer an auditor's query on a subset of rows in the ledger.

Most transactions in zkLedger do not include every bank. Every bank can pre-generate many range proofs for the value 0. We speedup transaction throughput by parallelizing range proof generation and validation.

5 Auditing

Auditing is a critical component of the financial system, and regulators use various techniques to measure systemic financial risk. Through the use of sums, means, ratios, variance, co-variance, and standard deviation, an auditor in zkLedger can determine the following, among other measurements:

- **Leverage ratios**. zkLedger can show how much of an asset a bank has on its books compared to its other holdings. This is helpful to estimate counterparty risk.
- **Concentration**. Regulators use a measure called the Herfindahl-Hirschman Index (HHI) to measure how competitive an industry is [29].
- **Real-time price indexes**. Auditors can get a sense of the price of assets that are traded over-the-counter and thus not tracked through exchanges.

Natively, zkLedger supports sums, which means linear combinations of values stored in the ledger. This comes from the additive structure of Pedersen commitments. But zkLedger also supports a more general query model, which can be considered in two parts: A *map* step and a *reduce* step.

Basic auditing. Consider the basic example where an auditor wants to determine how much of an asset a bank has on its books. As described in §4.2, the auditor will filter the rows by asset, multiply the entries in the bank's column, and then ask the bank to open the commitment product. This only requires one round of communication between the auditor and the bank and the messages are a constant size, independent of the number of rows. Because of zkLedger's commitment caches, this is very fast.

Map/reduce. An auditor can issue more complex queries that might require the exchange of more data or might require the participants to look at most of the rows in the ledger. Let's consider an auditor which wants to know the mean transaction size for a given bank and asset. An auditor cannot verify a bank's answer by simply totaling the bank's column of commitments and dividing the opened value by the number of rows, because such a computation would have an incorrect denominator. Namely, when the bank is not involved in a transaction, its column in the row will be commitment to 0, and should be discounted. In order to determine the correct denominator, the auditor and the bank run the following protocol:

1. **Filter.** The bank will filter the rows by asset.
2. **Produce new commitments.** For each row, the bank will commit to a single bit b, 1 or 0, depending on if the bank was involved in the transaction or not, and create a proof that the bank has done this recommitment correctly. Crucially, the auditor cannot distinguish between these commitments and so the bank's transactions are not revealed. We call this act of producing new commitments the *map* step. The map step also requires producing proofs that the new values were

correctly computed; in our example, for each transaction, the bank would produce a NIZK proof that $b = 1$ if and only if the transaction value was not equal to 0.

3. **Compute number of non-zero transactions.** The bank computes the homomorphic sum of the new commitments to bits $\vec{b}$ and opens it to reveal how many transactions were non-zero. This is the *reduce* step. This is the correct denominator to compute the mean transaction size. The auditor cannot tell anything about the values in $\vec{b}$ beyond what is revealed by the sum.
4. **Respond to auditor.** The bank then sends the auditor the sum of the values in its column, the vector of bit commitments and corresponding NIZK proofs, the number of its non-zero transactions n, and the sum of the r values in the commitments.
5. **Verification.** The auditor verifies the map step by verifying the commitments were done correctly, and verifies the reduce step and the number of non-zero transactions by confirming that the product of the vector of bit commitments is $g^n h^{\sum_{k=1}^{N} r_k}$.
6. **Compute answer.** The auditor computes the mean from the sum of the bank's column and the number of non-zero transactions.

An auditor could ask a bank for outlier transactions using a similar technique. For each row, the bank will commit to a bit b where $b = 1$ if a transaction's value for that bank is outside a specified range. As when computing the mean, the auditor can verify these commitments were produced correctly and obtain the sum. The bank can then open only the transactions where $b = 1$, and the auditor knows exactly how many transactions should be opened.

More complex auditing queries require multiple *map* and *reduce* computations. For example, here is how an auditor can learn the variance of transaction values $v_1, \ldots, v_N$:

1. **Compute the average transaction value.** Execute the protocol described above to compute the number of non-zero transactions n, and their average value $\bar{v}$.
2. **Apply the squaring map.** For each entry v_i in its row, the bank produces a fresh commitment cm'_i to v_i^2 and sends these commitments to the auditor. The bank also supplies NIZK proofs that the value hidden in each cm'_i is exactly the square of the value v_i committed to on the ledger.
3. **Apply the reduce step.** The auditor computes the product of the commitments cm'_i, and the bank opens up this commitment as $V = v_1^2 + \cdots + v_N^2$ by revealing $R = \sum_{i=1}^{N} r_i$. The auditor confirms that the product of the commitments is equal to $g^V h^R$.

The auditor now computes the variance σ as follows: $\sigma^2 = \frac{1}{n} \sum_{v_i \neq 0} (v_i - \bar{v})^2 = \frac{1}{n} V - \bar{v}^2$.

We note that whereas the square mapping used above corresponds to the second moment (variance), zkLedger can also compute higher statistical moments (e.g. skewness and kurtosis) using similar techniques and using cubing and fourth power mappings, respectively. See Appendix A for a list of measurements zkLedger supports.

zkLedger can support limited information release by using more complex reduce mappings. For example, instead of releasing the sum of values, the bank can produce a commitment to the *rounded* sum of values (e.g. to the first two decimal places), and use range proofs, also implemented in zkLedger, to show that the rounding was done correctly. Revealing just the order of magnitude of the quantity at hand lets the parties balance the granularity of information disclosure.

6 Implementation

To evaluate zkLedger's design, we implemented a prototype of zkLedger in Go. Our prototype uses a modified version of the *btcec* library [2] that contains the parameters and methods to compute with the elliptic curve secp256k1. We use Go's built-in SHA-256 implementation for our cryptographic hash function, and deterministically pick g and h by applying point decompression to the "nothing-up-my-sleeve" strings `SHA256`(0) and `SHA256`(1). Our prototype consists of approximately 3,200 lines of code, of which 40% implement cryptographic tools used by zkLedger (zero-knowledge proofs, range proofs, etc).

The implementation of the curve in zkLedger uses Go's `big.Int` type, which we make no effort to compress or serialize in an efficient way. A more optimized implementation could compress curve points. Our range proofs implement the protocol used in Confidential Assets [42]. Our NIZKs are based on Generalized Schnorr Proofs, which are three move interactive protocols; to make them non-interactive we apply the Fiat-Shamir heuristic [26], where we instantiate the random oracle using the SHA-256 hash function. Our prototype implementation does not implement the complex queries described in §5, and thus we do not evaluate them in §7.

7 Evaluation

Our evaluation answers the following questions:

- How expensive is it to store, prove and verify the different proofs in zkLedger? (§7.2)
- How does auditing scale with the size of the ledger? (§7.3)
- How does zkLedger scale with the number of banks? (§7.4)

#	Component	Create	Verify	Size
$2k$	Commitment	0.5 ms	0.5 ms	64 B
$2k$	Consistency	0.7 ms	0.8 ms	224 B
k	Disjunctive	0.9 ms	0.9 ms	288 B
k	Range	4.7 ms	3.5 ms	3936 B

Table 1: Number of each proof component in a transaction for k banks. Size of and time to create and verify the components with 12 cores. The range proof create and verify benefit from the additional cores.

7.1 Experimental setup

Microbenchmarks. We run microbenchmarks on a 12 core Intel machine with i7-X980 3.33 GHz CPUs and 24GB of RAM, running 64-bit Linux 4.4.0 on Ubuntu 16.04.3. Each microbenchmark runs the same code a Bank runs to create and validate transactions.

Distributed experiment. We run the distributed experiments on a set of 12 virtual machines each with 4 cores of Intel Xeon E5-2640 2.5 GHz processors, 24GB of RAM, and the same software setup as above. There is one auditor, one server providing the service of the ledger, and a varying number of banks, one per server. Servers communicate using the `net/rpc` Go package over TCP. All experiments use Go version 1.9.

7.2 Proof overhead in zkLedger

Table 1 shows the time to prove and verify the proofs in a transaction in zkLedger. There are two commitments, two consistency proofs, and one each of the disjunctive and range proofs in a transaction entry. There is a transaction entry per Bank. Table 1 also shows the sizes of the various proofs, in bytes. These sizes are estimated based on the size of the underlying fields in the struct in memory; these proofs could be further compressed. Range proofs dominate the size of the transactions.

The left graph in Figure 3 shows the time it takes to create and verify a transaction varying the number of overall banks, which increases the number of entries per transaction. This indicates that as we increase the number of banks, both transaction creation and verification times per bank increase linearly, but parallelization helps. Proving and validating range proofs dominates transaction creation and verification, but this cost is also highly parallelizable. 12 cores gives a 2.8× speedup when creating a transaction with 20 banks; a bank can create or validate a transaction for up to 20 banks in less than 200ms.

As described in §4.1, zkLedger uses Borromean ring signatures to prove that a value is in a certain range, and supports values up to 2^{40}. Reducing the supported range of values would reduce range proof cost since that cost is linear in the number of bits in the size of the range. There are also newer proof systems, such as Bulletproofs, which might create much smaller range proofs [13]. We plan to evaluate zkLedger with Bulletproofs in future work.

7.3 Cost of auditing ledgers

The left graph in Figure 4 shows that for certain functions, the time to audit is independent of the number of transactions in the ledger. This is because the Auditor and Banks maintain commitment caches, which already have the commitment product necessary to prove to the auditor the sum of the values in its column. The audit function is measuring the Herfindhal-Hirschman Index, so the auditor communicates with each bank.

When the auditor cannot use a commitment cache, perhaps because it was offline, it must process the whole ledger to compute the commitment product. This also applies to more complex auditing like the types described in §5, when the auditor has to verify recommitments for every row in the ledger. These costs are shown in the middle graph in Figure 4. This graph shows how long it takes the auditor to compute the Herfindahl-Hirschman Index on a ledger of varying sizes without using the commitment caches, so the auditor must process every row of the ledger. In these measurements, the auditor does not verify each row. As expected, this time increases linearly with the number of rows. This indicates that maintaining commitment caches is important for real-time auditing. However, even without commitment caches, auditing time is reasonable: 3.5 seconds for 100K transactions. This suggests the complex auditing queries, in which the auditor computes a similar set of operations per row, will also be on the order of many seconds. zkLedger currently only maintains commitment product caches per asset per bank, but could maintain more.

For a fixed size ledger, this audit function costs order the number of banks. The right graph in Figure 4 demonstrates the auditing costs of computing the Herfindahl-Hirschman Index on a ledger of 2000 transactions as we vary the number of banks, both with and without commitment caches. The auditor audits the banks in parallel. Auditing cost for this function grows slightly with the number of banks, since more banks increase the variability in parallel auditing and the auditor must wait for the last bank to respond before computing the final answer.

In these figures, each point is the mean of running the auditing query 20 times, with error bars representing one standard deviation from the mean.

7.4 Scaling with more banks

There are two significant costs that grow with the number of banks in zkLedger: a serial step to create transactions that increases linearly, and verifying transactions which increases quadratically with the number of banks. As

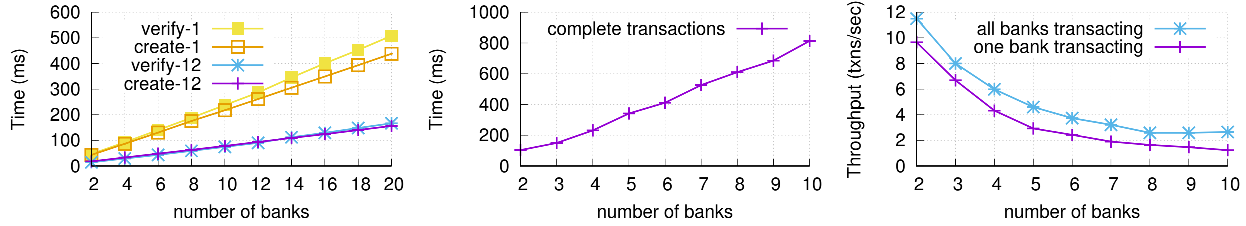


Figure 3: Transaction creation and verification time for one bank (left), varying the number of entries in the transaction. Single-threaded and multi-threaded performance, with 12 threads. Time to fully process a transaction including creation, broadcast to ledger, banks and auditor, and verification by all parties (middle). Throughput (right) varying the number of banks.

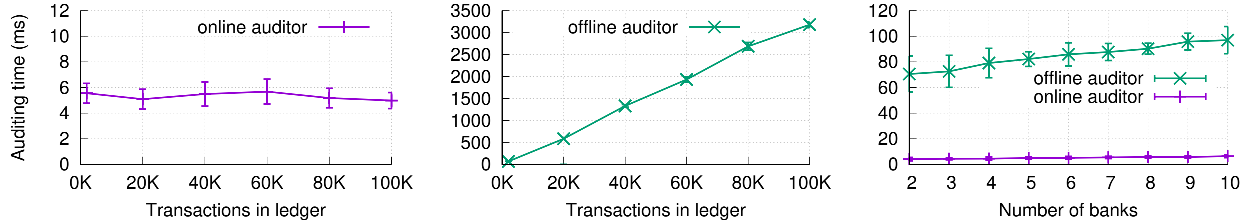


Figure 4: Time to audit ledgers of different sizes (4 banks), and with a varying number of banks (2000 row ledger). Audit time is independent of the size of the ledger (left) thanks to commitment caches maintained by the online auditor. When commitment cache optimization is turned off (middle) the audit time is linear in the size of the ledger. Audit time grows with the number of banks (right) and is much higher without commitment caches.

described in §4.1, a bank needs to use its entry from transaction $n-1$ to create transaction n. So though a bank can use many cores to produce the proofs for a single transaction in parallel, multiple banks cannot produce different transactions in parallel. In zkLedger, banks start creating transaction n before seeing $n-1$ but the bank cannot complete the transaction until $n-1$ is accepted to the ledger and verified, causing an inherent bottleneck.

The second major cost is around verification. Every bank must verify every transaction, so the more banks, the larger each transaction and thus the more work that needs to be done by each bank. The middle graph in Figure 3 measures the time it takes one bank to create and all participants in zkLedger to completely process a transaction. One bank creates a transaction and sends it to the ledger, which then broadcasts the transaction to all banks and an online auditor. The auditor and every bank verify the transaction. As we increase the number of banks, work increases quadratically; however, banks can verify transactions in parallel so the time to process transactions only increases linearly. The right graph in Figure 3 shows that as we surmised, zkLedger's throughput worsens with more banks. The one bank transacting line in this graph is the same data as the middle graph.

Since range proofs dominate the costs of transaction creation and verification, we are optimistic that a faster range proof implementation will directly improve performance. zkLedger's current performance is comparable to Solidus, a privacy-preserving distributed ledger which achieves 3-4 transactions per second with online validation but, unlike zkLedger, does not support auditing.

8 Future work

zkLedger focuses on providing provably correct auditing over private transaction data, but zkLedger does not have a way to recover if the distributed ledger is corrupted. In this case, the parties maintaining the ledger would have to come together to recreate historical transactions. zkLedger also does not provide recourse if a bank commits an unintended transaction to the ledger. A future version of zkLedger might provide rectifying transactions or participant agreed-upon rollback.

9 Conclusion

zkLedger is the first distributed ledger system to provide strong transaction privacy, public verifiability, and complete, provably correct auditing. zkLedger supports a rich set of auditing queries which are useful to measure the financial health of a market. We developed a design using non-interactive zero-knowledge proofs to prove transactions maintain financial invariants and to support auditing. Our evaluation shows that zkLedger has reasonable performance for transaction settlement and auditing.

10 Acknowledgements

We thank Alexander Chernyakhovsky, Thaddeus Dryja, David Lazar, Ronald L. Rivest, C.J. Williams, and our shepherd and reviewers for helpful comments. The research leading to these results has received funding from: the Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370; and the Ethics and Governance of Artificial Intelligence Fund.

References

[1] Bank secrecy act of 1970, October 1970. 12 U.S.C. 103.

[2] Package btcec implements support for the elliptic curves needed for Bitcoin., July 2017. https://godoc.org/github.com/btcsuite/btcd/btcec.

[3] ABBE, E. A., KHANDANI, A. E., AND LO, A. W. Privacy-preserving methods for sharing financial risk exposures. *The American Economic Review 102*, 3 (2012), 65–70.

[4] ARASU, A., BLANAS, S., EGURO, K., KAUSHIK, R., KOSSMANN, D., RAMAMURTHY, R., AND VENKATESAN, R. Orthogonal security with cipherbase. In *CIDR* (2013).

[5] BAJAJ, S., AND SION, R. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering 26*, 3 (2014), 752–765.

[6] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 8.

[7] BEN-SASSON, E., CHIESA, A., GENKIN, D., TROMER, E., AND VIRZA, M. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 90–108.

[8] BEN-SASSON, E., CHIESA, A., GREEN, M., TROMER, E., AND VIRZA, M. Secure sampling of public parameters for succinct zero knowledge proofs. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (2015), SP '15, pp. 287–304.

[9] BLUM, M., FELDMAN, P., AND MICALI, S. Non-interactive zero-knowledge and its applications. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing* (1988), STOC '88, pp. 103–112.

[10] BOGDANOV, D., TALVISTE, R., AND WILLEMSON, J. Deploying secure multi-party computation for financial data analysis. In *International Conference on Financial Cryptography and Data Security* (2012), Springer, pp. 57–64.

[11] BOWE, S., GABIZON, A., AND GREEN, M. A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK. Cryptology ePrint Archive, Report 2017/602, 2017.

[12] BOWE, S., GABIZON, A., AND MIERS, I. Scalable multi-party computation for zk-SNARK parameters in the random beacon model. Cryptology ePrint Archive, Report 2017/1050, 2017.

[13] BÜNZ, B., BOOTLE, J., BONEH, D., POELSTRA, A., AND MAXWELL, G. Bulletproofs: Short proofs for Confidential Transactions and more. In *Security and Privacy (SP), 2018 IEEE Symposium on* (2018), IEEE.

[14] BURKHART, M., STRASSER, M., MANY, D., AND DIMITROPOULOS, X. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. *Network 1*, 101101 (2010).

[15] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186.

[16] CECCHETTI, E., ZHANG, F., JI, Y., KOSBA, A., JUELS, A., AND SHI, E. Solidus: Confidential distributed ledger transactions via pvorm.

[17] CHAIN, I. Confidential assets. https://blog.chain.com/hidden-in-plain-sight-transacting-privately-on-a-blockchain-835ab75c01cb.

[18] CORRIGAN-GIBBS, H., AND BONEH, D. Prio: Private, robust, and scalable computation of aggregate statistics. *arXiv preprint arXiv:1703.06255* (2017).

[19] COUNCIL, F. R. Developments in audit 2016/2017 full report, 2017. http://www.frc.org.uk/getattachment/915c15a4-dbc7-4223-b8ae-cad53dbcca17/Developments-in-Audit-2016-17-Full-report.pdf.

[20] CRAMER, R., DAMGÅRD, I., AND SCHOENMAKERS, B. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings* (1994), pp. 174–187.

[21] DAGHER, G. G., BÜNZ, B., BONNEAU, J., CLARK, J., AND BONEH, D. Provisions: Privacy-preserving proofs of solvency for Bitcoin exchanges. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, CO, 2015), ACM, pp. 720–731.

[22] Corda, 2017. https://github.com/corda/corda.

[23] Digital asset holdings, 2017. http://digitalasset.com.

[24] Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union L119* (May 2016), 1–88.

[25] FELDMAN, A. J., ZELLER, W. P., FREEDMAN, M. J., AND FELTEN, E. W. Sporc: Group collaboration using untrusted cloud resources. In *OSDI* (2010), vol. 10, pp. 337–350.

[26] FIAT, A., AND SHAMIR, A. How to prove yourself: practical solutions to identification and signature problems. In *Proceedings of the 6th Annual International Cryptology Conference* (1987), CRYPTO '87, pp. 186–194.

[27] GARMAN, C., GREEN, M., AND MIERS, I. Accountable privacy for decentralized anonymous payments. Cryptology ePrint Archive, Report 2016/061, 2016. http://eprint.iacr.org/2016/061.

[28] GREENBERG, A. Fbi says it's seized $28.5 million in bitcoins from ross ulbricht, alleged owner of silk road. *Forbes 25* (2013).

[29] HERFINDAHL, O. C. *Concentration in the steel industry*. PhD thesis, Columbia University New York, 1950.

[30] LAMPORT, L., ET AL. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[31] LI, J., KROHN, M. N., MAZIERES, D., AND SHASHA, D. E. Secure untrusted data repository (sundr). In *OSDI* (2004), vol. 4, pp. 9–9.

[32] MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS) 29*, 4 (2011), 12.

[33] MAURER, U. Unifying zero-knowledge proofs of knowledge. *Proceedings of the 2nd International Conference on Cryptology in Africa* (2009), 272–286.

[34] MAXWELL, G. Confidential transactions. `https://people.xiph.org/~greg/confidential_values.txt` *(Accessed 8/2017)* (2015).

[35] MAXWELL, G., AND POELSTRA, A. Borromean ring signatures. `https://raw.githubusercontent.com/Blockstream/borromean_paper/master/borromean_draft_0.01_34241bb.pdf` *(Accessed 6/2017)* (2015).

[36] MEIKLEJOHN, S., POMAROLE, M., JORDAN, G., LEVCHENKO, K., MCCOY, D., VOELKER, G. M., AND SAVAGE, S. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference* (2013), ACM, pp. 127–140.

[37] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system, 2008.

[38] OBER, M., KATZENBEISSER, S., AND HAMACHER, K. Structure and anonymity of the bitcoin transaction graph. *Future internet 5*, 2 (2013), 237–250.

[39] ONGARO, D., AND OUSTERHOUT, J. K. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference* (2014), pp. 305–319.

[40] PAPADIMITRIOU, A., BHAGWAN, R., CHANDRAN, N., RAMJEE, R., HAEBERLEN, A., SINGH, H., MODI, A., AND BADRINARAYANAN, S. Big data analytics over encrypted datasets with seabed. In *OSDI* (2016), pp. 587–602.

[41] PEDERSEN, T. P. Non-interactive and information-theoretic secure verifiable secret sharing. In *Proceedings of the 11th Annual International Cryptology Conference* (1992), CRYPTO '91, pp. 129–140.

[42] POELSTRA, A., BACK, A., FRIEDENBACH, M., MAXWELL, G., AND WUILLE, P. Confidential assets, 2017. 4th Workshop on Bitcoin and Blockchain Research.

[43] POPA, R. A., REDFIELD, C., ZELDOVICH, N., AND BALAKRISHNAN, H. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 85–100.

[44] POPA, R. A., STARK, E., HELFER, J., VALDEZ, S., ZELDOVICH, N., KAASHOEK, M. F., AND BALAKRISHNAN, H. Building web applications on top of encrypted data using mylar. In *NSDI* (2014), pp. 157–172.

[45] REID, F., AND HARRIGAN, M. An analysis of anonymity in the bitcoin system. In *Security and privacy in social networks*. Springer, 2013, pp. 197–223.

[46] RON, D., AND SHAMIR, A. Quantitative analysis of the full bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security* (2013), Springer, pp. 6–24.

[47] SASSON, E. B., CHIESA, A., GARMAN, C., GREEN, M., MIERS, I., TROMER, E., AND VIRZA, M. Zerocash: Decentralized anonymous payments from Bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 459–474.

[48] SCHNORR, C.-P. Efficient signature generation by smart cards. *Journal of cryptology 4*, 3 (1991), 161–174.

[49] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 38–54.

[50] TU, S., KAASHOEK, M. F., MADDEN, S., AND ZELDOVICH, N. Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment* (2013), vol. 6, VLDB Endowment, pp. 289–300.

[51] Zcash, 2017. `http://z.cash`.

[52] ZHENG, W., DAVE, A., BEEKMAN, J. G., POPA, R. A., GONZALEZ, J. E., AND STOICA, I. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI* (2017), pp. 283–298.

A Auditing Queries

Figure 5 is a list of the types of measurements zkLedger supports, including the estimated running time and the data beyond the measurement that is leaked. For example, as described in §5, computing transaction size variance requires leaking the mean transaction size and number of transactions per bank.

B Zero-knowledge proofs and privacy guarantees

To build our zero-knowledge protocols we rely on the following general result of Maurer (Theorem 3, [33]):

Theorem B.1. *Let* $(H_1, \star)$ *and* $(H_2, \otimes)$ *be two (not-necessarily commutative) groups and* $f : H_1 \to H_2$ *be a group homomorphism:* $f(x \star y) = f(x) \otimes f(y)$. *Let* $\ell \in \mathbb{Z}$, $u \in H_1$, $\mathscr{C} \subset \mathbb{Z}$ *be such that:*

1. $\gcd(c_1 - c_2, \ell) = 1$ *for all* $c_1, c_2 \in \mathscr{C}$ *(with* $c_1 \neq c_2$*), and*
2. $f(u) = z^{\ell}$.

There exists a 2*-extractable* Σ*-protocol for language* $\mathscr{L} := \{z : \exists w \text{ s.t. } z = f(w)\}$. *Moreover, a protocol con-*

Measurement	Time	Additional information leaked
Sum total of asset(s) per bank	$O(1)$	none
Outlier transactions per bank	$O(n)$	none
Concentration	$O(k)$	Sum totals per bank
Ratio holdings	$O(k)$	Sum totals per bank, number of transactions per bank
Mean transaction size per bank	$O(kn)$	Number of transactions per bank
Variance, skew, kurtosis	$O(kn)$	Mean per bank, number of transactions per bank
Real-time price averages	$O(kn)$	Number of transactions and average per bank over time period t

Figure 5: Types of supported auditing queries, their running time to audit based on the number of banks k and the number of rows in the ledger n, and a description of what information is leaked to the auditor.

sisting of s rounds is proof-of-knowledge if $1/|\mathscr{C}|^s$ *is negligible, and zero-knowledge if* $|\mathscr{C}|$ *is polynomially bounded.*

Using Theorem B.1 we can now unify the treatment of most of the zero-knowledge proofs used in our system. For example, the consistency proofs π^{C} rely on the following result:

Theorem B.2. *Let* $\mathbb{G}$ *be an order-r cyclic group and* g, h, pk *be any three elements of* $\mathbb{G}$. *There exists a* 2*-extractable* Σ*-protocol for language* $\mathscr{L}_{\mathsf{aux}} := \{(\mathsf{cm}, \mathsf{Token}) : \exists v, r \text{ s.t. } \mathsf{cm} = g^v h^r \wedge \mathsf{Token} = \mathsf{pk}^r\}$.

Proof. Consider $H_1 = \mathbb{Z}_r \times \mathbb{Z}_r$, defining the group operation to be component-wise addition, and let $H_2 = \mathbb{G} \times \mathbb{G}$, similarly defining group operation to be component-wise. Then $f(x,y) := (g^x h^y, \mathsf{pk}^y)$ is a group homomorphism between H_1 and H_2. Indeed, $f(x_1+x_2, y_1+y_2) = (g^{x_1+x_2} h^{y_1+y_2}, \mathsf{pk}^{y_1+y_2}) = f(x_1,y_1) \otimes f(x_2,y_2)$. Furthermore, setting $\ell = r$ and $u = (0,0)$ we have that for all $z \in H_2$ the following holds: $z^\ell = (1,1) = f(u)$. Therefore, we can apply Theorem B.1 and conclude that $\mathscr{L}_{\mathsf{aux}}$ has a 2-extractable Σ-protocol. □

To summarize, the three proofs in zkLedger (see Section 4.3) that relate commitments $\mathsf{cm}_i :=: g^{v_i} h^{r_i}$ and audit tokens $\mathsf{Token}_i := (\mathsf{pk}_i)^{r_i}$ have the following form:

- **Proof of Assets (π^{A}).** This proof consists of a new commitment $\mathsf{cm'}_i$, together with an audit token Token'_i, and a zero-knowledge proof asserting that either $\mathsf{cm'}_i$ is a re-commitment of the value in cm_i or a recommitment to the sum of the values in $\prod_{j=0}^m \mathsf{cm}_j$. To create this proof zkLedger relies on Theorem B.1 for constituent proofs; as these are Sigma-protocols we apply the standard OR-composition [20] to get the final disjunctive zero-knowledge proof. To prove that the commited value is in the range we use the range proofs in Confidential Assets [42]. We are investigating more recent proof systems (e.g. Bulletproofs [13]) to further reduce the proof size.
- **Proof of Balance (π^{B}).** In our implementation this proof is an empty string: the prover simply chooses the commitment randomness subject to condition $\sum r_i = 0$. With such a choice the auditor homomorphically adds the commitments and checks that this addition results in the neutral element of the group $\prod \mathsf{cm}_i = g^{\sum v_i} h^{\sum r_i} = g^0 h^0 = 1$.
- **Proof of Consistency (π^{C}).** We use two proofs derived from Theorem B.2 to assert that the randomness used in cm_i and Token_i are the same, and the randomness used in $\mathsf{cm'}_i$ and Token'_i are the same.

C Privacy in the combined system

Pedersen commitments provide information-theoretic privacy. In zkLedger Pedersen commitments are published together with authentication tokens and zero-knowledge proofs. We note that zero-knowledge proofs indeed don't spoil the information-theoretic privacy of committed values: the output of the zero-knowledge proof simulator is identical to the output produced by parties in the system. However, when combining Pedersen commitments and authentication tokens, the privacy guarantees become computational as we now explain.

The commitment, audit token, and public key triple $(\mathsf{cm}, \mathsf{Token})$ is of the form $(g^v h^r, \mathsf{pk}^r, \mathsf{pk}) = (g^v h^r, h^{\mathsf{sk} \cdot r}, h^{\mathsf{sk}})$, and these three values uniquely determine the v. That is, if an adversary could break the discrete logarithm problem, it could solve for sk, use that and value of Token to infer r, and finally recover v. That said, under the Decisional Diffie-Hellman (DDH) assumption, no information is leaked. Furthermore, the DDH assumption is widely assumed to hold in zkLedger's elliptic curve group.

Recall, that DDH holds if no polynomially-bounded adversary can distinguish between tuples of the form (h, h^a, h^b, h^{ab}) and (h, h^a, h^b, h^c) for a randomly chosen generator h and exponents a, b, c. Assume that a stateful adversary $\mathscr{A}_{\mathsf{zkL}}$, when given input (g, h, pk) is able to produce two values v_1 and v_2 such that it can distinguish commitments (and associated audit tokens) to v_1 from commitments (and audit tokens) to v_2, i.e. the adversary is able to distinguish the distributions $(g^{v_1} h^r, h^{\mathsf{sk} \cdot r}, h^{\mathsf{sk}})$

and $(g^{v_2}h^r, h^{\mathsf{sk}\cdot r}, h^{\mathsf{sk}})$. We now show how to use $\mathscr{A}_{\mathsf{zkL}}$ to construct an adversary $\mathscr{A}_{\mathsf{DDH}}$ breaking the DDH assumptions.

After receiving its challenge (h,x,y,z), where (x,y,z) is distributed either as (h^a,h^b,h^{ab}) or as (h^a,h^b,h^c), the adversary $\mathscr{A}_{\mathsf{DDH}}$ proceeds as follows. It samples a random generator g and calls $\mathscr{A}_{\mathsf{zkL}}$ on input (g,h,x), x now serving the role of the bank's public key. When $\mathscr{A}_{\mathsf{zkL}}$ returns two values v_1 and v_2, the DDH adversary $\mathscr{A}_{\mathsf{DDH}}$ picks a random $k \in \{1,2\}$ and prepares $\mathsf{cm}_k = g^{v_k}y$, $\mathsf{Token} = z$ and sends $(\mathsf{cm}_k, \mathsf{Token})$ to $\mathscr{A}_{\mathsf{zkL}}$. Finally, if $\mathscr{A}_{\mathsf{zkL}}$'s guess for k is correct, $\mathscr{A}_{\mathsf{DDH}}$ responds that the DDH challenge was of the form (h,h^a,h^b,h^{ab}) (i.e. a DDH quadruple), otherwise it responds that the DDH challenge was of the form (h,h^a,h^b,h^c) (i.e. a random quadruple).

Note that when $\mathscr{A}_{\mathsf{DDH}}$'s challenge is a DDH quadruple, the zkLedger adversary $\mathscr{A}_{\mathsf{zkL}}$ is run on a distribution it expects. In particular, all of its inputs are correctly formed with respect to $\mathsf{sk} = a$ and $r = b$. Whereas, when $\mathscr{A}_{\mathsf{DDH}}$'s challenge is a random quadruple, the inputs to $\mathscr{A}_{\mathsf{zkL}}$ have *information-theoretically* no information about the committed value: indeed, $\mathsf{Token} = h^c$ is unrelated to $\mathsf{cm} = g^v h^b$. Therefore, if the zkLedger adversary $\mathscr{A}_{\mathsf{zkL}}$ wins the commitment hiding game with non-negligible advantage, so does $\mathscr{A}_{\mathsf{DDH}}$ in the DDH game. Note that the proof extends to the multiple entry case by a standard hybrid argument.

Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization

Yilong Geng[1], Shiyu Liu[1], Zi Yin[1], Ashish Naik[2],
Balaji Prabhakar[1], Mendel Rosunblum[1], and Amin Vahdat[2]

[1]Stanford University, [2]Google Inc.

Abstract

Nanosecond-level clock synchronization can be an enabler of a new spectrum of timing- and delay-critical applications in data centers. However, the popular clock synchronization algorithm, NTP, can only achieve millisecond-level accuracy. Current solutions for achieving a synchronization accuracy of 10s-100s of nanoseconds require specially designed hardware throughout the network for combatting random network delays and component noise or to exploit clock synchronization inherent in Ethernet standards for the PHY.

In this paper, we present HUYGENS, a software clock synchronization system that uses a *synchronization network* and leverages three key ideas. First, coded probes identify and reject impure probe data—data captured by probes which suffer queuing delays, random jitter, and NIC timestamp noise. Next, HUYGENS processes the purified data with Support Vector Machines, a widely-used and powerful classifier, to accurately estimate one-way propagation times and achieve clock synchronization to within 100 nanoseconds. Finally, HUYGENS exploits a natural network effect—the idea that a group of pair-wise synchronized clocks must be transitively synchronized—to detect and correct synchronization errors even further.

Through evaluation of two hardware testbeds, we quantify the imprecision of existing clock synchronization across server-pairs, and the effect of temperature on clock speeds. We find the discrepancy between clock frequencies is typically 5-10μs/sec, but it can be as much as 30μs/sec. We show that HUYGENS achieves synchronization to within a few 10s of nanoseconds under varying loads, with a negligible overhead upon link bandwidth due to probes. Because HUYGENS is implemented in software running on standard hardware, it can be readily deployed in current data centers.

1 Introduction

Synchronizing clocks in a distributed system has been a long-standing important problem. Accurate clocks enable applications to operate on a common time axis across the different nodes, which, in turn, enables key functions like consistency, event ordering, causality and the scheduling of tasks and resources with precise timing. An early paper by Lamport [13] frames the question of ordering events in distributed systems and proposes a solution known for obtaining partial orders using “virtual clocks,” and Liskov [16] describes many fundamental uses of synchronized clocks in distributed systems.

Our work is motivated by several compelling new applications and the possibility of obtaining very fine-grained clock sychronization at an accuracy and cost that is much less than provided by current solutions. For example, in finance and e-commerce, clock synchronization is crucial for determining transaction order: a trading platform needs to match bids and offers in the order in which they were placed, *even if they entered* the trading platform from different gateways. In distributed databases, accurate clock synchronization allows a database to enforce external consistency [8] and improves the throughput and latency of the database. In software-defined networks, the ability to schedule tasks with precise timing would enforce an ordering of forwarding rule updates so that routing loops can be avoided [18]. In network congestion control, the ability to send traffic during time slots assigned by a central arbiter helps achieve high bandwidth and near-zero queueing delays [28]. Indeed, precisely synchronized clocks can help to revise the “clockless” assumption underlying the design of distributed systems and change the way such systems are built.

Consider distributed databases as an example. Spanner [8] provides external consistency[1] at a global scale

[1]A database is said to be *externally consistent* if it can ensure for each transaction A that commits before another transaction B starts, A is serialized before B.

using clocks synchronized to within T units of time, typically a few milliseconds. In order to achieve external consistency, a write-transaction in Spanner has to wait out the clock uncertainty period, T, before releasing locks on the relevant records and committing. Spanner can afford this wait time because T is comparable to the delay of the two-phase-commit protocol across globally distributed data centers. However, for databases used by real-time, single data center applications, the millisecond-level clock uncertainty would fundamentally limit the database's write latency, throughput and performance. Thus, if a low latency database, for example, RAMCloud [24], were to provide external consistency by relying on clock synchronization, it would be critical for T to be in the order of 10s of nanoseconds so as not degrade the performance.

This relationship between clock synchronization and database consistency can been seen in CockroachDB[1], an open source scalable database. In CockroachDB, uncertainty about clocks in the system can cause performance degradation with read requests having to be retried. That is, a read issued by server A with timestamp t for a record at server B will be successful if the last update of the record at B has a timestamp s, where $s \leq t$ or $s > t+T$. Else, clock uncertainty necessitates that A retry the read with timestamp s. For example, in an experimental CockroachDB cluster of 32 servers we read 128 records, each updated every 25 ms. We found that as the clock uncertainty T was reduced from 1 ms to 10 us and then to 100 ns, the retry rate fell from 99.30% to 4.74% and to 0.08% in an experiment with 10,000 reads for each value of T.

Thus, while it is very desirable to have accurately synchronized clocks in distributed systems, the following reasons make it hard to achieve in practice. First, transaction and network speeds have shortened inter-event times to a degree which severely exposes clock synchronization inaccuracies. The most commonly used clocks have a quartz crystal oscillator, whose resonant frequency is accurate to a few parts per million at its ideal operating temperature of 25-28°C [34]. When the temperature at a clock varies (in either direction), the resonant frequency *decreases quadratically* with the temperature (see [34] for details). Thus, a quartz clock may drift from true time at the rate of 6-10 microseconds/sec. But the one-way delay (OWD), defined as the raw propagation (zero-queuing) time between sender and receiver, in high-performance data centers is under 10 μs. So, if the clocks at the sender and receiver are not frequently and finely synchronized, packet timestamps are rendered meaningless! Second, "path noise" has made the nanosecond-level estimation of the OWD, a critical step in synchronizing clocks, exceedingly difficult. Whereas large queuing delays can be determined and removed from the OWD calculation, path noise—due to small fluctuations in switching times, path asymmetries (e.g., due to cables of different length) and clock timestamp noise, which is in the order of 10s–100s of ns is not easy to estimate and remove from the OWD.

The most commonly used methods of estimating the OWD are the Network Time Protocol (NTP) [21], the Precision Time Protocol (PTP) [4], Pulse Per Second (PPS) [25]—a GPS-based system, and the recently proposed Data center Time Protocol (DTP) [14]. We review these methods is more detail later; for now, we note that they are either cheap and easy to deploy but perform poorly (NTP) or provide clock synchronization to an accuracy of 10s–100s of nanoseconds in data center settings but require hardware upgrades (PTP, DTP and PPS) which impose significant capital and operational costs that scale with the size of the network.

The algorithm we propose here, HUYGENS, achieves clock synchronization to an accuracy of 10s of nanoseconds at scale, and works with current generation network interface cards (NICs) and switches in data centers without the need for any additional hardware. A crucial feature of HUYGENS is that it processes the transmit (Tx) and receive (Rx) timestamps of probe packets exchanged by a pair of clocks *in bulk*: over a 2 second interval and simultaneously from multiple servers. This contrasts with PTP, PPS and DTP which look at the Tx and Rx timestamps of a single probe–ack pair individually (i.e., 4 timestamps at a time). By processing the timestamps in bulk, HUYGENS is able to fully exploit the power of inference techniques like Support Vector Machines and estimate both the "instantaneous time offset" between a pair of clocks and their "relative frequency offset". These estimates enable HUYGENS to be not bound by rounding errors arising from clock periods.

Contributions of the paper. The goal of our work is to precisely synchronize clocks in data centers, thereby making "timestamping guarantees" available to divers applications as a fundamental primitive alongside bandwidth, latency, privacy and security guarantees. We have chosen to synchronize clocks (e.g. the PTP Hardware Clocks, or PHCs [2]) in the NICs attached to servers. By accurately synchronizing NIC clocks, we obtain globally accurate timestamps for data, protocol messages and other transactions between different servers. NIC-to-NIC probes encounter the minimum amount of noise in the path propagation time as compared to server-to-server probes which also suffer highly variable stack latencies. Our main contributions are:

(1) A comprehensive and large-scale study of clock discrepancies in real-world networks. The major findings are: (i) pairwise clock rates can differ by as much as 30μs/sec; (ii) clock frequencies vary at time scales of minutes due to temperature effects, but are fairly constant

over 2–4 second intervals; and (iii) a quantification of the effect of queueing delays, path noise and path asymmetry on clock synchronization.
(2) The HUYGENS algorithm and its real-time extension HUYGENS-R, which can respectively be used by applications for aligning timestamps offline or for real-time clock synchronization.
(3) A NetFPGA-based verification in a 128-server, 2-stage Clos data center network shows that HUYGENS and HUYGENS-R achieve less than 12–15ns average error and under 30–45ns 99th percentile error at 40% network load. At 90% load, the numbers increase to 16–23ns and 45–58ns, respectively.
(4) We propose a lightweight implementation of HUYGENS that runs "in-place" with the probe data. In a 40 Gbps data center testbed, HUYGENS only takes around 0.05% of the link bandwidth and less than 0.44% of a server's CPU time.

2 Literature survey

As mentioned in the Introduction, all methods of determining clock offsets involve estimating the OWD. In order to estimate the OWD between clocks A and B, A sends a probe packet to B containing the transmission time of the probe. The OWD can either be estimated directly by determining the time spent by the probe at each element en route from A to B (e.g., as in PTP), or by estimating the RTT (where B sends a probe back to A). In the latter case, assuming the OWD is equal in both directions, halving the estimated RTT gives the OWD. Using the estimate of the OWD and the probe's transmit time, B can work out the time at A and synchronize with it. We survey the four methods mentioned previously for estimating the OWD between a pair of clocks.
NTP. NTP [21] is a widely-used clock synchronization protocol. It estimates the offset between two clocks by considering multiple probe-echo pairs, picking the three with the smallest RTTs, and taking half their average to get the OWD. It achieves an accuracy of tens of milliseconds [23] to 10s of microseconds [26], depending on the network type (e.g., wide-area vs data center).

NTP uses simple methods to process the probe data, hence it only achieves a coarse-grained clock synchronization. HUYGENS does stronger processing of the same probe data to extract a much more refined estimate of the offset between a pair of clocks. It then uses the network effect to obtain a further 3x reduction in the estimation error.
PTP. PTP [4] uses hardware timestamps to counter stack delays. It uses "transparent" switches which are able to record the ingress and egress time of a packet to accurately obtain packet dwell times at switches. With more extensive hardware support at switches and a dedicated network for carrying PTP packets the White Rabbit system [22] can achieve sub-nanosecond precision. However, the accuracy in a conventional fully "PTP-enabled network" ranges from a few tens to hundreds of nanoseconds [32]. If the network is not fully PTP-enabled, synchronization accuracy can degrade by 1000x even when the two clocks are only a few hops apart [32]. Detailed tests conducted in [14] show that PTP performs poorly under high load, corroborating similar findings in [32].
DTP. The DTP protocol [14] sidesteps the issue of estimating time-varying queue sizes, stack times, and most noise variables by making a clever observation: The IEEE 802.3 Ethernet standards provide a natural clock synchronization mechanism between the transmitter and receiver PHYs at either end of a wire. Therefore, DTP can achieve a very fine-grained clock synchronization without increasing network traffic and its performance is not load-dependent. It is limited by the clock-granularity of the standard: for a 10Gbps network link the granularity is 6.4ns, and since four timestamps are involved in calculating OWD, a single hop synchronization accuracy of 25.6ns can be achieved. DTP requires special extra hardware at every PHY in the data center, necessitating a fully "DTP-enabled network" for its deployment.
PPS. PPS obtains accurate (atomic) time using a GPS receiver antenna mounted on the roof of the data center. It brings this signal to a multi-terminal distribution box using cables with precisely measured lengths. The multi-terminal box amplifies and relays the clock over cables (also with precisely known lengths) to NICs which are capable of receiving PPS signals. This makes PPS prohibitively expensive to deploy at scale, most installations have a designated "stratum 1" zone with just a few (typically tens of) servers that have access to PPS.

In summary, current methods of synchronizing clocks in a data center are either not accurate enough or require hardware modifications to almost every element of a data center, making them very expensive to deploy.

3 Our approach

The HUYGENS algorithm exploits some key aspects of modern data centers[2] and uses novel estimation algorithms and signal processing techniques. We look at these in turn.
Data center features. Most data centers employ a symmetric, multi-level, fat-tree switching fabric [30, 29]. By symmetry we mean that the number of hops between any pair of servers, A and B, is the same in both directions. We *do not require the paths to involve identically the*

[2] Even though this paper is focused on clock synchronization in data centers, we believe the principles extend to wide area networks, possibly with a loss in synchronization accuracy.

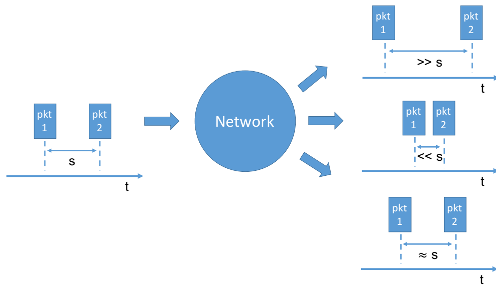


Figure 1: Coded probes

same switches in both directions.[3] Symmetry essentially equalizes the OWD between a pair of NICs in each direction, except for a small amount of "path noise" (quantified in Section 4). Furthermore, these propagation times are small, well-bounded by 25–30μs. Abundant bisection bandwidth and multiple paths between any pair of servers ensure that, even under 40-90% load, there is a reasonably good chance probes can traverse a network without encountering queueing delays. Finally, there are many servers (and NICs), making it possible to synchronize them in concert.

Algorithms and techniques. HUYGENS sets up a synchronization network of probes between servers; each server probes 10–20 others, regardless of the total number of servers in the network.

Coded probes. Naturally, probes which encounter no queueing delays and no noise on the path convey the most accurate OWDs. To automatically identify such probes, we introduce *coded probes*: a pair of probe packets going from server i to j with a small inter-probe transmission time spacing of s. If the spacing between the probe-pair when they are received at server j is very close to s, we deem them as "pure" and keep them both. Else, they are impure and we reject them. In Figure 1 the first two possibilities on the receiver side show impure probes and the third shows pure probes. Coded probes are very effective in weeding out bad probe data and they improve synchronization accuracy by a factor of 4 or 5.[4]

Support Vector Machines. The filtered probe data is processed by an SVM [9], a powerful and widely-used classifier in supervised learning. SVMs provide much more accurate estimates of propagation times between a pair of NICs than possible by the simpler processing methods employed by NTP and PTP. In Section 4 we shall see that "path noise" is small in magnitude and pathological in the sense that it has "negative-delay" components. Thus, simple techniques such as estimating the min-RTT or linear regression to process the probe data do not work. They can filter out large delays but cannot cope with small-magnitude path noise and are adversely affected by the negative-delay components which artifically shrink the OWD. The combination of coded probes and SVMs copes well with these problems.

Network effect.[5] Even though a data center network increases the path noise between clocks A and B because of multiple hops, it can simultaneously *increase the signal* by providing other clocks and new, potentially non-overlapping, paths for A and B to synchronize with them. Therefore, it is better—more accurate and scalable—to synchronize many clocks simultaneously than a pair of them at a time, as explained below. A significant byproduct of using the network effect is that it is particularly good at detecting and correcting path asymmetries.

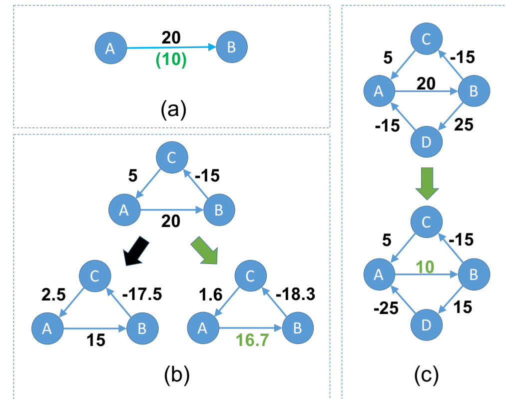


Figure 2: The Network Effect. How more clocks can help identify and reduce synchronization errors

Consider Figure 2. In (a), after pair-wise synchronization, clocks A and B believe that B is ahead of A by 20 units of time ($A \xrightarrow{20} B$). However, the truth (shown in green) is that B is ahead of A by only 10 units of time. A and B can never discover this error by themselves. In (b), a third clock C has undergone pairwise synchronization with A and B, and the resulting pairwise offsets are shown on the directed edges. Going around the loop A→B→C→A, we see that there is a *loop offset surplus* of 10 units! This immediately tells all three clocks there are errors in the pairwise synchronization. The bottom of Figure 2 (b) shows two possible corrections to the pairwise estimates to remove the loop surplus. Of these two choices, the HUYGENS algorithm will pick the one on the right, $A \xrightarrow{16.7} B \xrightarrow{-18.3} C \xrightarrow{1.6} A$. This is the *minimum-norm* solution and evenly distributes the loop surplus of 10 onto the 3 edges. Of course, if a fourth clock, D, joins the network, the two loop surpluses, equal to 10 and 30, are redistributed according to the minimum-norm solution by

[3] In Section 4 we show that a real-world 40 Gbps network has highly symmetric paths, for symmetry as defined here.

[4] The idea of using a pair of closely-spaced packets to determine the *available bandwidth* on a path was introduced in [12], see also [10]. Whereas that use case needs the separation between probes to increase in order to determine available bandwidth, we require no separation.

[5] Synchronization must be reflexive, symmetric and transitive; network effect is a deliberate exploitation of the transitivity property.

HUYGENS and this ends up correcting the discrepancy on edge A→B to its correct value of 10 units.

In general, the minimum-norm solution does not completely correct synchronization errors, nor does it evenly distribute loop surpluses. Its effect is to significantly pull in outlier errors (see Section 5).

The network effect is related to co-operative clock synchronization algorithms in the wireless literature [31, 15, 17, 19]. In the wireless setting, synchronization is viewed from the point of view of achieving consensus; that is, nodes try to achieve a global consensus on time by combining local information, and they use "gossip algorithms" to do so. Thus, a node exchanges information on time with neighboring nodes and this information gradually flows to the whole network with time synchronization as the outcome. This approach is appropriate in ad hoc wireless networks where nodes only know their neighbors. However, convergence times can be long and it can be hard to guarantee synchronization accuracy. Like [31], HUYGENS uses the network effect to directly verify and impose the transitivity property required of synchronization. In the wired setting in which HUYGENS operates, far away nodes are connected via the synchronization network, typically over a data center fabric. Thus, the probes have to contend with potentially severe network congestion, but the synchronization network can be chosen and not be restricted by network connectivity as in the ad hoc wireless case. The synchronization algorithm can be central and hence very fast (one-shot matrix multiplication—see Section 5) and yield provable synchronization error reduction guarantees.

4 Clocks in the real-world

In this section we use two testbeds and empirically study the degree to which pairs of clocks differ and drift with respect to one another, the effect of temperature in altering clock frequencies, and a characterization of queueing delays and path noise affecting the accurate measurement of the end-to-end propagation time of probes.

Testbed T-40. This is a 3-stage Clos network, all links running at 40Gbps. T-40 has 20 racks each with a top-of-the-rack (TOR) switch, and a total of 237 servers with roughly 12 servers per rack. There are 32 switches at spine layer 1. Each TOR switch is connected to 8 of these switches while each spine layer 1 switch is connected to 5 TOR switches. Thus, there is a 3:2 (12:8) oversubscription at the TOR switches. Each spine layer 1 switch is connected to 8 spine layer 2 switches and vice versa (there are 32 spine layer 2 switches). T-40 represents a state-of-the-art data center.

Testbed T-1. T-1 is a 2-stage Clos network with all links running at 1Gbps. It consists of 8 racks, each rack has a 1Gbps TOR switch and 16 *logical* servers. The 16 logical servers are built out of 4 Jetway network appliances (JNA) [11], 4 logical servers per JNA, as explained below. Each TOR switch has 16 downlinks and 8 uplinks, each uplink connecting it to one of 8 logically distinct spine switches. Thus, there is a 2:1 oversubscription at the TOR switches. The 8 logically distinct spine switches are built from 4 48-port 1Gbps physical switches using VLAN configurations [33]. T-1 respresents a low-end commodity data center.

For reasons of economy—in monetary, space and heat dissipation terms—we use JNAs to build servers in T-1. Each JNA has a 4-core Intel Celeron J1900 CPU, 8GB RAM, 250GB of disk storage, and ten 1Gbps Ethernet ports. Each Ethernet port has an Intel I211 NIC. The logical servers in a single JNA share the CPU, the RAM and the PCIe buses. Even though there can be 10 logical servers per JNA (one per NIC), to avoid overwhelming the CPU we build 4 logical servers per JNA. Each logical server is built inside a Docker container [20], giving them complete independence of operation. The servers implement a probing/responding application as well as various workload traffic generation applications. The different capabilities of T-40 and T-1 necessitated different probing and timestamping mechanisms, as explained below.

Probing. Recall that a probe is actually a pair of packets, called *coded probes*. We use 64-byte UDP packets for probing. Each server probes K other randomly chosen servers once every T seconds. Probing is bidirectional: servers which are probed send back probes every T seconds. In T-40, $K = 20$ and $T = 500\mu$s, and in T-1, $K = 10$ and $T = 4$ms.

Timestamping. The receive timestamps in T-40 and T-1 are recorded upon the receipt of the probes. In T-40, the transmit timestamp is equal to the time the probe's TX completion descriptor is written back into the host memory. The write-back time is often nearly equal to the transmission time, but, occasionally, it can be a few 10s or 100s of nanoseconds *after* the probe transmit time. This gives rise to a "negative-delay" timestamp noise; i.e., noise which can lead to probe propagation times *strictly smaller* than the absolute minimum possible. In T-1, the JNA's architecture makes the write-back approach perform much worse. Instead, the Intel I211 NIC in the JNA places the transmit start time of a probe in its payload and forwards it to the receiving NIC, where it is extracted.[6]

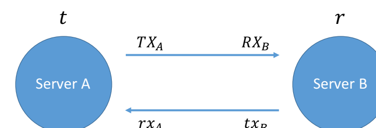


Figure 3: Signaling between clocks.

[6]This feature is not supported by the NICs in T-40.

Signaling. Consider Figure 3. Let T and R be the absolute times at which a probe was transmitted at NIC A and received at NIC B. Let TX_A and RX_B be the corresponding transmit and receive timestamps taken by the NICs. Define $\Delta_A = TX_A - T$ and $\Delta_B = RX_B - R$. Since $R > T$, we get $RX_B - \Delta_B > TX_A - \Delta_A$. Rearranging terms, we get

$$\Delta_B - \Delta_A < RX_B - TX_A. \quad (1)$$

From a probe (or echo) in the reverse direction, we get

$$tx_B - rx_A < \Delta_B - \Delta_A < RX_B - TX_A. \quad (2)$$

Thus, each probe gives either an upper bound or a lower bound on the discrepancy between two clocks depending on its direction; the tightest bounds come from probes encountering zero queueing delay and negligible noise. The discrepancy is time-varying due to different clock frequencies.

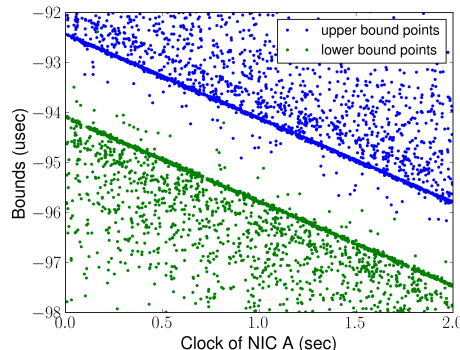


Figure 4: Bounds on the discrepancy between clocks in T-40

4.1 Clock frequencies across servers

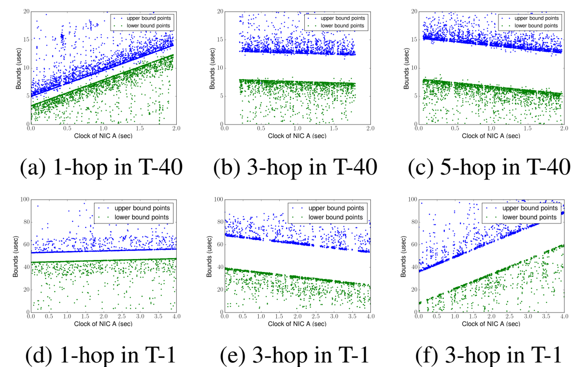
Figure 4 shows upper and lower bounds on $\Delta_B - \Delta_A$ in T-40 plotted against the time at NIC clock A: each blue dot is an upper bound and each green dot is a lower bound. The following observations can be made:

1. The set of blue points delineating the "least upper bound" of the data points lie on a straight line over short timescales, 2 seconds in this case. This line is parallel to the straight line on which the green dots delineating the "greatest lower bound" lie. We refer to the region between the lines as the "forbidden zone." There are a number of blue dots and green dots in the forbidden zone. These are due to the "negative-delay" NIC timestamp noise mentioned previously. It is crucial to filter out these points.

2. Since the dots on the lines bounding the forbidden zone capture the smallest one-way propagation time of a probe (equal only to wire and switching times), the spacing between them (roughly equal to 1700 ns in Figure 4) is the smallest RTT between the two NICs. Assuming symmetric paths in both directions, half of the spacing will equal the smallest one-way propagation time (roughly 850 ns).

3. The upper and lower bound lines have a non-zero slope and intercept. The slope in Figure 4 is close to -1.6μs/sec and it measures the "drift" in the frequencies of the clocks at A and B. That is, when clock A measures out one second of time, clock B would have measured out $1-(1.6\times10^{-6})$ second. The average of the two intercepts is the offset between the two clocks: when clock A's time is 0, clock B's time is roughly -93.3μs.

Remark. The slope captures the discrepancy in the clock frequencies and represents the intrinsic pull away from synchronism between the two clocks. *When the clocks are completely synchronized, the slope and the average of the two intercepts should both be 0.*

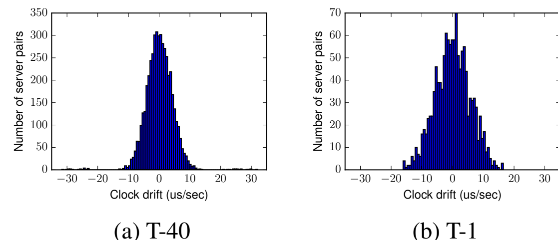


(a) 1-hop in T-40 (b) 3-hop in T-40 (c) 5-hop in T-40

(d) 1-hop in T-1 (e) 3-hop in T-1 (f) 3-hop in T-1

Figure 5: Examples of clock drifts

Figure 5 shows more examples of relative clock drifts in T-40 and T-1. Figures 6 (a) and (b) show the histograms of the drifts between pairs of clocks in T-40 and T-1, respectively. The number of pairs considered in each testbed and a numerical quantification of the data in Figure 6 is in Table 1. While most pair-wise clock drifts are around 6-10μs/sec, the maximum can get as high as 30μs/sec.



(a) T-40 (b) T-1

Figure 6: Distribution of the relative frequency difference between pairs of clocks

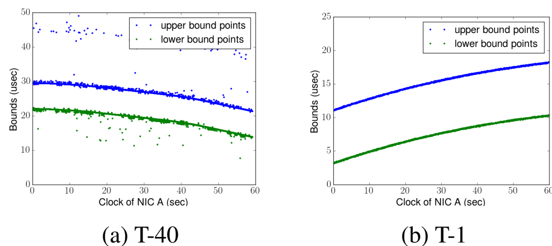
Variation in clock frequencies due to temperature

When considering longer timescales, in the order of minutes, one can sometimes observe nonlinearities in the upper and lower bound curves, see Figure 7. This is due

Testbed	#Servers	#Clock pairs	St.dev. of slope (drift)	Max abs. of slope (drift)
T-40	237	4740	4.5μs/sec	32.1μs/sec
T-1	128	1280	5.7μs/sec	16.5μs/sec

Table 1: Summary statistics of clock drift rates

to temperature changes which affect the resonance frequency of the clocks. The temperature can change when a NIC sends a lot of data or otherwise dissipates a lot of power. The temperature of a NIC varies slowly with time. Therefore, even though there is nonlinearity in the order of minutes or longer, at timescales of 2-4 seconds the upper and lower bound curves are essentially linear. We shall approximate the nonlinear curves by piecewise linear functions over 2-4 seconds.



(a) T-40 (b) T-1

Figure 7: Nonlinear drifts in clock frequencies

4.2 Queueing delays and path noise

1. **Queueing delay.** This is the total queueing delay on the path and can be 100s of microseconds to a few milliseconds, depending on the load. However, we shall see that even at 90% loads there are enough points with zero queueing delay (hence, on the boundary of the forbidden zone) to accurately synchronize clocks.

2. **Path noise.** We distinguish two types of path noise.
Switching noise. Even under 0 load, a probe's switch-traversal time can be jittered due to random delays from passing through multiple clock domains, input arbitration, or other hardware implementation reasons specific to the switch (e.g., store-and-forward versus cut-through switching). "Dark matter packets" are another source of switch noise. These are protocol packets (e.g., spanning tree protocol (STP) [5], SNMP [3], link-layer discovery protocol (LLDP) [6]) not emitted by the end-hosts, hence invisible to them. When probes queue behind them at switches, small and random switching delays are added.
NIC timestamp noise. This is the discrepancy between the timestamp reported by the NIC and the true time that the probe was transmitted or received. This can be caused by faulty hardware, or the way timestamping is implemented in the NIC. As described in Section 4, the latter can cause *negative-delay*, giving rise to points in the forbidden zone.

Empirically, path noise larger than 50ns degrades performance in T-40. In T-1 that number is 2μs. Packet transmission times in switches, in the PHYs, etc, are 40 times longer on T-1 than T-40, and this corresponds with an increase in noise magnitude.

4.3 Path symmetry

T-40 is a fat-tree network with 1, 3 and 5 hops between server pairs. By investigating the statistics of the RTTs between servers at different hop distances, we can test the veracity of our path symmetry assumption. Recall that path symmetry means that servers separated by a certain number of hops have more or less the same OWD, *regardless* of the particular paths taken in the forward and reverse direction to go between them.

	1 hop	3 hops	5 hops
# server-pairs	390	1779	6867
Ave. ZD-RTT	1570 ns	4881 ns	7130 ns
Min. ZD-RTT	1512 ns	4569 ns	6740 ns
Max. ZD-RTT	1650 ns	4993 ns	7253 ns

Table 2: Zero-delay-RTTs (ZD-RTTs) in T-40

In Table 2, we consider the zero-delay-RTT (ZD-RTT) between server pairs at different hops from one another. The forward and reverse routes between a pair of servers in T-40 are *arbitrary and not identical.* Nevertheless, we see from the minimum, average and maximum values of the ZD-RTT that it is tightly clustered around the mean. Since this was taken over a large number of server-pairs, we see empirical evidence supporting path symmetry.

If asymmetry exists in a network, it will degrade the overall synchronization accuracy by half the difference in the forward and reverse path delays. Fortunately, the network effect can be used to identify asymmetric paths and potentially replace them with symmetric paths (or paths that are less asymmetric).

5 The Huygens Algorithm

The Huygens algorithm synchronizes clocks in a network every 2 seconds in a "progressive-batch-delayed" fashion. Probe data gathered over the interval [0, 2) seconds will be processed during the interval [2, 4) seconds (hence batch and delayed). The algorithm completes the processing before 4 seconds and corrections can be applied to the timestamps at 1 sec, the midpoint of the interval [0, 2) seconds. Then we consider probe data in the interval [2, 4) seconds, and so on. By joining the times at the midpoints of all the intervals with straight lines, we obtain the corrections at all times. Thus, the corrected time is available as of a few seconds before the present

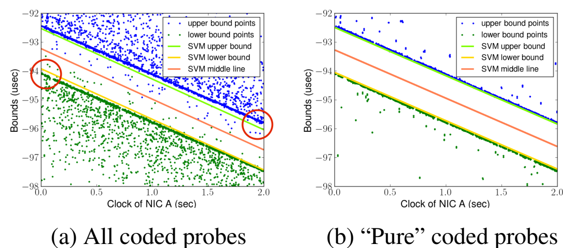
time (hence progressive).[7]

Coded probes. Presented informally previously, coded probes are a pair of packets, P_1 and P_2, with transmit timestamps t_1 and t_2 and receive timestamps r_1 and r_2, respectively. A coded probe is "pure" if $r_2 > r_1$ and $|(r_2 - r_1) - (t_2 - t_1)| < \varepsilon$, where $\varepsilon > 0$ is a prescribed guard band. Else, it is "impure". The timestamps of both of the pure probes are retained, and those of both impure probes are discarded. If either P_1 and P_2 is dropped, we discard the coded pair.

Support Vector Machines. SVMs are a widely used and powerful tool for linear and nonlinear classification in supervised learning settings [9]. A *linear SVM* is supplied with a collection of points (x_i, l_i) for $1 \le i \le N$, where x_i is a point in $\mathbb{R}^2$ and l_i is a binary label such as "upper bound point" or "lower bound point". It classifies points of like label, separating them with a hyperplane of maximal margin; that is, a hyperplane which is at a maximum distance from the closest point of either label in the data.

When used in our context, the SVM is given the upper and lower bound points derived from the probe data between clocks A and B over a 2-second interval. If the data is clean, i.e., if there are no points in the forbidden zone (defined in Section 4.1) and there are enough points with noise- and delay-free propagation time in both probing directions, the SVM will return a straight line with slope α_{AB} and intercept β_{AB}.

We use *soft-margin SVMs* which can tolerate points in the forbidden zone and other noise and delays. However, the performance of the SVM is sensitive to these artifacts, and *especially* to points in the forbidden zone. Therefore, even if the SVM returns a line in the noisy case, it will not be an accurate estimate of the discrepancy $\Delta_B - \Delta_A$. For this reason we first treat the data with the coded probe filter and extract pure probes to which we apply the SVM. For example, in Figure 8 (a) the SVM processes *all probe data* between a pair of clocks in T-40 and it is clear that its estimation of the upper and lower bound lines (the support vectors) are inaccurate. In (b) we see the significant improvement from using coded probes to filter out bad probe data.



(a) All coded probes (b) "Pure" coded probes

Figure 8: Effectiveness of coded probes. In T-40, coded probes reduce synchronization error by 4-5x.

[7] In Section 8 we obtain a real-time extension of the HUYGENS.

5.1 Network Effect

Suppose there are n clocks, $C_1, \ldots, C_n$, connected through a data center fabric. WLOG let C_1 be the "reference" clock with which all clocks will be synchronized. At time 0 a probe mesh is set up between the n clocks, each one probing K others. This is done using messages between the nodes and results in the "probing graph," $G = (V, E)$. The "owner" of edge (i, j) is the node who initiated the probing, ties broken at random. Once the probe mesh has been set up, we construct the Reference Spanning Tree (RST) on G with C_1 as the root in a breadth-first fashion.

In every 2-second interval, we synchronize C_i to C_1 at the midpoint of the interval.[8] The midpoints of adjacent intervals are then connected with straight lines to obtain a piecewise linear synchronization of C_i with C_1. Accordingly, consider the interval $[2j, 2(j+1))$ seconds for some $j \ge 0$. For ease of notation, denote $2j$ by L, $2(j+1)$ by R, and the midpoint $2j+1$ by M.

Iteration

1. *Coded probes and SVMs.* Timestamp data collected during $[L, R)$ is processed using the coded probes filter and bad probes are discarded. For each edge $(i, j) \in G$, where i is the owner of the edge, the filtered data is processed by an SVM to yield the slope, α_{ij}, and the intercept, β_{ij} of the hyperplane determined by the SVM. Let $(\vec{\alpha}, \vec{\beta})$ be the vectors of the $\alpha_{i,j}$ and $\beta_{i,j}$ for $(i, j) \in G$. The equations

$$\alpha_{ji} = \frac{-\alpha_{ij}}{1 + \alpha_{ij}} \quad \text{and} \quad \beta_{ji} = \frac{-\beta_{ij}}{1 + \alpha_{ij}} \tag{3}$$

relate the slopes and intercepts in one direction of (i, j) to the other.

2. *Preliminary estimates at time M.* Use the RST and the $(\vec{\alpha}, \vec{\beta})$ to obtain the preliminary, group-synchronized time at clock C_i with respect to the reference clock's time of M_1 sec. This is done as follows. First consider C_i to be a neighbor of C_1 on the RST. Then,

$$M_i^P = M_1 + \alpha_{1i} M_1 + \beta_{1i}.$$

Proceed inductively down the tree to obtain M_j^P at each clock C_j when C_1 equals M_1.

3. Obtain $\Delta_{ij}^P \triangleq \alpha_{ij} M_i^P + \beta_{ij}$ for every i and j, with the convention $M_1^P = M_1$. Gather the Δ_{ij}^P into a vector Δ^P, the "preliminary pair-wise clock discrepancy vector."

4. *Network effect: loop correction.* Apply loop correction to Δ^P to determine the degree of inconsistency in the pair-wise clock estimates, and obtain Δ^F, the "final pair-wise clock discrepancy vector."

$$\Delta^F = \left[I - A^T \left(A A^T \right)^{-1} A \right] \Delta^P, \text{where the} \tag{4}$$

"loop-composition matrix", A, is defined below.

5. Obtain the final estimates M_i^F at C_i when the time at

[8] For concreteness, time instances are taken with reference to C_1.

C_1 equals M_1. For a neighbor C_i of C_1 on the RST:

$$M_i^F = M_1 + \Delta_{1i}^F.$$

Proceed inductively down the RST to obtain M_j^F at each clock C_j when C_1 equals M_1.

End Iteration

Elaboration of Steps 2–4

Steps 2 and 3. In Step 2 preliminary midpoint times are obtained using only the probe data on the edges of the RST. These midpoint estimates are used in Step 3 to get preliminary discrepancies, Δ^P, between the clock-pairs across all edges of the probing graph, G.

Step 4. Using equation (4), loop-wise correction is applied to Δ^P to obtain the final pair-wise clock discrepancy vector, Δ^F. Given $G = (V,E)$, the matrix A is obtained as follows. The number of columns of A equals $|E|$, each column corresponds with a *directed edge* $(i \to j) \in E$. Each row of A represents a loop of G, traversed in a particular direction. For example, suppose loop L is traversed along edges $(i \to j)$, $(j \to k)$, $(k \to l)$ and $(l \to i)$. Then, the row in A corresponding to loop L will be a 1 at columns corresponding to edges $(i \to j)$, $(j \to k)$ and $(k \to l)$, a -1 corresponding to column $(i \to l)$, and 0 elsewhere. The number of rows of A equals the largest set of *linearly independent* loops (represented in the row-vector form described) in G.

Derivation of equation (4). The quantity $A\Delta^P$ gives the total surplus in the preliminary pair-wise clock discrepancy in each loop of A. For example, for loop L defined above, this would equal:

$$\Delta_{ij}^P + \Delta_{jk}^P + \Delta_{kl}^P - \Delta_{il}^P \stackrel{\triangle}{=} y_L.$$

Let $Y = A\Delta^P$ represent the vector of loop-wise surpluses. In order to apply the loop-wise correction, we look for a vector N which also solves $Y = AN$ and posit the correction to be $\Delta^F = \Delta^P - N$. Now, A has full row rank, since the loops are all linearly independent. Further, since the number of linearly independent loops in G equals $|E| - |V| + 1$ which is less than $|E|$, the equation $Y = AN$ is under-determined and has multiple solutions. We look for the *minimum-norm solution* since this is most likely the best explanation of the errors in the loop-wise surpluses.[9] Since the pseudo-inverse, $N = A^T(AA^T)^{-1}Y = A^T(AA^T)^{-1}A\Delta^P$, is well-known to be the minimum-norm solution [27], we get

$$\Delta^F = \Delta^P - N = [I - A^T(AA^T)^{-1}A]\Delta^P,$$

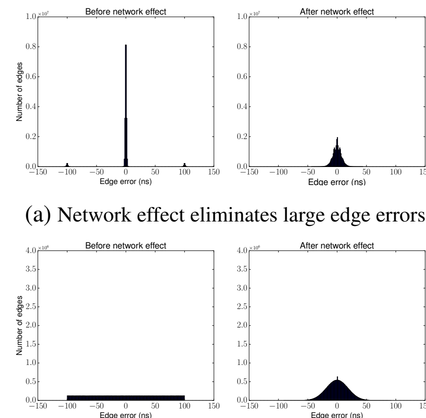
which is equation (4).[10]

[9] It is most likely that the loop-wise surpluses are due to a lot of small errors on the edges rather than a few large ones.

[10] Under a Gaussian assumption on the noise in the discrepancy vector Δ^P, we've shown that the standard deviation of the noise in Δ^F is a factor $\frac{1}{\sqrt{K}}$ of the noise in Δ^P. Numerically, this means a reduction of the errors by 68.4% and 77.6% when $K = 10$ and 20, respectively. Due to a shortage of space, we omit the proof here.

Remark. The minimum-norm solution gives the optimal (Maximum Likelihood) estimate of edge errors if they were distributed as independent and identically distributed (IID) Gaussians *before* applying the network effect. Even when edge errors are not IID Gaussians, the minimum-norm solution ensures that the edge errors after the network effect are closer to Gaussian with a much smaller variance ($\frac{1}{\sqrt{K}}$ of the pre-network-effect variance). Most importantly, this is true whether the initial edge errors occured due to significant path asymmetries (which can be large and systematic) or due to path noise (which is typically small and nearly zero mean).

Figure 9 illustrates the point. In a network of 256 nodes, each probing 10 other nodes, we see the network effect reduces the standard deviation of edge errors (after coded probes and SVMs) in two cases: (a) the edge errors are typically small but some times can be as large as 100 ns, and (b) distribted uniformly between -100 ns and 100 ns. In both cases the errors after appying the minimum-norm solution are clustered around 0 in a bell-shaped distribution.



(a) Network effect eliminates large edge errors

(b) Network effect compresses uniformly distributed edge errors

Figure 9: Examples of network effect: N=256, K = 10, collective results across 10000 runs

6 Implementation and Evaluation

The probing phase of HUYGENS is obviously distributed: it must occur along the edges of G. Many of the other basic operations can also by implemented distributedly at the sensing nodes, hence significantly reducing data movement overheads. Or, they can be implemented on a dedicated computing system separate from the sensing system. We describe the distributed implementation.

6.1 A lightweight, scalable implementation

We implement HUYGENS as an "in-place" distributed system, making it lightweight and scalable. There are

three main components—the master, the slaves and the sensors. A global master node initiates the probing phase and runs loop-wise correction (Steps 2–5). There is a sensor and a slave node at each server for conducting probing, data collection, coded probe filtering and running SVMs. The sensor sends the probes and collects timestamps, the slave runs all the computations. This implementation eliminates overheads due to data transfer and makes HUYGENS scalable because each sensor–slave combination only needs to send the α- and β-values to the master node and *not the probe data*.

Probing bandwidth and CPU overhead. Probing at the rates we have implemented (see Probing in Section 4), HUYGENS takes 0.05% of the bandwidth at each node in T-40 and 0.25% of the bandwidth at each node in T-1. The fact that HUYGENS takes a much smaller percentage of the bandwidth in T-40 than in T-1 is due to an important property: it sends roughly the same number of probes per unit time regardless of the line rate.

The CPU overhead imposed by HUYGENS is negligible. On an 2.8 GHz Intel i5 (5575R) CPU, using only *one core* and running SVM for 2 seconds probe data from one edge takes less than 7ms. Therefore, the in-place distributed implementation would take less than 0.44% of CPU time in a modern 32-core server even when $K = 20$.

6.2 Evaluation

We run experiments on T-1 to evaluate: (i) the accuracy of HUYGENS using NetFPGAs, (ii) the efficacy of the network effect, and (iii) HUYGENS' performance under very high network load.

Network load. We use the traffic generator in [7]: each server requests files simultaneously from a random number (30% 1, 50% 2 and 20% 4) of other servers, called the "fanout" of the request. The gap between adjacent requests are independent exponentials with load-dependent rate. The file sizes are distributed according to a heavy-tailed distribution in the range [10KB, 30MB] with an average file size of 2.4MB. This traffic pattern can mimic a combination of single flow file transfers (fanout = 1) and RPC style incast traffic (fanout > 1).

Evaluation Methodology

NetFPGA-CML boards provide two natural ways to verify HUYGENS' accuracy in synchronizing clocks:

(i) **Single FPGA.** Each NetFPGA has four 1GE ports connected to a *common clock*. We take two of these ports, say P_1 and P_2, and make them independent servers by attaching a separate Docker container to each. P_1 and P_2 then become two additional servers in the 128-server T-1 testbed. Using HUYGENS we obtain the "discrepancy" in the clocks at P_1 and P_2, whereas the ground truth is that there is 0 discrepancy since P_1 and P_2 have the same clock.

Remark. To make it more difficult for HUYGENS to synchronize P_1 and P_2, we do not allow them to directly probe each other. They are at least 2 or 3 hops away on the RST.

(ii) **Different FPGAs.** This time P_1 and P_2 are ports on different FPGAs, with clocks C_1 and C_2, say. They can be synchronized using HUYGENS on the T-1 testbed. They can also be synchronized using a *direct channel* by connecting the GPIO pins on the two NetFPGAs using copper jumper wires. The direct channel provides us with an essentially *noise- and delay-free, short RTT*[11] signaling method between C_1 and C_2. The pin-to-pin signals are sent and echoed between the NetFPGAs every 10ms. We process the TX and RX timestamps of the pin-to-pin signals using a linear regression and obtain the discrepancy between C_1 and C_2 using the direct channel.

In both (i) and (ii), the HUYGENS probe data is processed using the algorithm described in Section 5. We take C_1 (the clock at P_1) as the root of the RST. C_2 is then another node on the RST and HUYGENS gives a *preliminary* and a *final estimate* of its midpoint in a 2 second probing interval with respect to C_1's midpoint (as described in Steps 2–5). This is compared to the ground truth or direct channel discrepancy between C_1 and C_2. Even though (i) gives us a ground truth comparison, we use (ii) because it gives a sense of the degree of synchonization possible between two different clocks that are connected directly.

7 Verification

We consider a 10-minute experiment.[12] For the single FPGA comparison, we obtain HUYGENS' preliminary and final estimates of the midpoint times at C_2 in successive 2-second intervals with respect to C_1. We compare these estimates with the ground truth discrepancy, which is 0. There are 300 2-second intervals in total; we report the average and the 99th percentile discrepancies. In the case of different FPGAs, we compare the estimates from HUYGENS with the estimate from the direct channel. Table 3 contains the results.

	Single NetFPGA		Different NetFPGAs	
	Prelim	Final (net effect)	Prelim	Final (net effect)
Mean of abs. error (ns)	41.4	11.2	47.8	13.4
99th percentile of abs. error (ns)	91.0	22.0	92.1	30.2

Table 3: HUYGENS synchronization accuracy: 16-hour experiment in T-1 at 40% load with $K = 10$

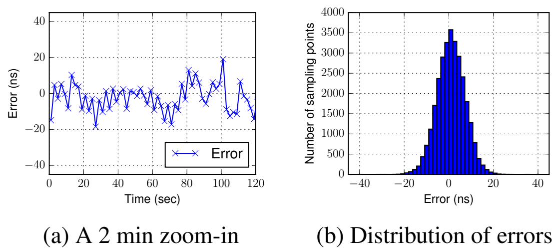
We see that the network effect (loop-wise correction) is quite powerful, yielding a 3-4x improvement in

[11] The RTT between the FPGAs is 8 clock cycles (64ns) or smaller.

[12] Section 8 presents results from a 16 hour run.

the accuracy of clock synchronization. The most compelling demonstration is the single FPGA comparison in which the ground truth is unambiguously 0. The different FPGA comparison shows HUYGENS with loop correction obtains the same performance as an almost error-free, direct connection between the two clocks even though in HUYGENS the clocks are connected through a network of intermediaries.

Figure 10 shows a 2 minute sample trajectory of the errors as well as the distribution of the errors over 10 minutes in the same FPGA comparison.

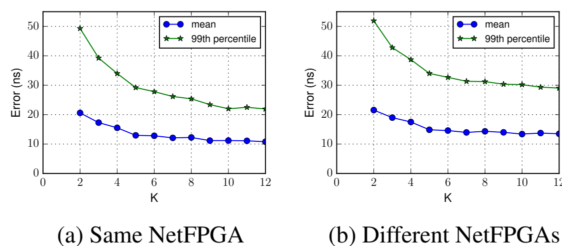


(a) A 2 min zoom-in (b) Distribution of errors

Figure 10: HUYGENS vs ground truth for 2 ports on the same NetFPGA: 16-hour experiment at 40% load with $K = 10$

7.1 The network effect

We consider the benefit of the network effect by increasing K from 2 to 12. Under an assumed theoretical model, it was stated in Section 5 that this would reduce the standard deviation of the clock synchronization error by a factor $\frac{1}{\sqrt{K}}$. Figure 11 quantifies the benefit of the network effect for the mean of the absolute error and the 99th percentile errors as K gets larger.

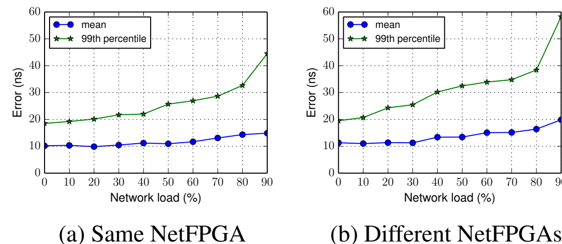


(a) Same NetFPGA (b) Different NetFPGAs

Figure 11: Mean and 99th percentile synchronization error in T-1 as K varies with 40% load

7.2 Performance under high load

Figure 12 shows the accuracy of HUYGENS under different network loads. As can be seen, even at 90% load HUYGENS is still able to synchronize clocks with 99th percentile error smaller than 60 nanoseconds. HUYGENS is robust to very high network load for the following reasons: (i) It applies intensive statistical procedures on each 2 seconds of probe data; the 2 second interval happens to be long enough that, even at very high load, a small number of probes go though empty queues, allowing HUYGENS to obtain accurate estimates. (ii) HUYGENS takes advantage of the redundant connectivity in the network: even if one probing pair is blocked due to congestion, the two clocks in this probing pair will still quite likely be able to reach other clocks and synchronize with them. (iii) Loop-wise correction is able to identify and compensate probing pairs which are badly affected by high load. We couldn't load T-1 at more than 90% because the switches in T-1 have shallow buffers and drop large numbers of packets, leading to excessive TCP timeouts.



(a) Same NetFPGA (b) Different NetFPGAs

Figure 12: HUYGENS error at different loads in T-1, $K = 10$.

7.3 Comparision with NTP

Since NTP is a widely-used algorithm and does not require specialized hardware, it is worth comparing it to HUYGENS on T-1. For fairness, we let NTP use the same hardware timestamps as HUYGENS, although almost all real-world implementations of NTP use CPU timestamps.

Load	Method	Single NetFPGA		Different NetFPGAs	
		Mean of abs. error (ns)	99th percentile of abs. error (ns)	Mean of abs. error (ns)	99th percentile of abs. error (ns)
0%	HUYGENS	10.2	18.5	11.3	19.5
	NTP	177.7	558.8	207.8	643.6
40%	HUYGENS	11.2	22.0	13.4	30.2
	NTP	77975.2	347638.4	93394.0	538329.9
80%	HUYGENS	14.3	32.7	16.4	38.4
	NTP	211011.7	778070.4	194353.5	688229.1

Table 4: A comparison of HUYGENS and NTP.

Table 4 shows that, even with hardware timestamps, NTP's error is 4 orders of magnitude larger than HUYGENS's under medium to high network loads. Note that although some implementations of NTP use a DAG-like graph to synchronize a clock with clocks which are upstream on the DAG, this operation is local when compared to the more global loop-wise correction step in the network effect.

8 Real-time Huygens

We now extend HUYGENS to get a real-time version, HUYGENS-R. Underlying this extension is the empiri-

cal observation in Section 4 that clock frequencies are slowly varying. Therefore, a linear extrapolation of the estimates of HUYGENS over a few seconds will yield a reasonably accurate real-time clock synchronization algorithm.

The extension is best described using Figure 13. Consider clocks C_1 and C_i. Let I_k be the time interval $[2(k-1), 2k)$. Step 5 of the HUYGENS algorithm yields the final offset of the midpoint of each I_k at C_i from the midpoint of the same interval at C_1. For example, $T_2 + 3$ is the time at C_i when the time at C_1 is 3 seconds; in general, $T_k + (2k-1)$ is the time at C_i when the time at C_1 is $2k-1$ seconds. By connecting the T_k with straight lines, we get the HUYGENS offsets between C_i and C_1 at all times.

The green points, O_l, are the offsets between clocks C_i and C_1 for the real-time version. They are obtained as shown in the figure: O_l lies on the line from T_l to T_{l+1} when the time at C_1 equals $2l+6$. Thus, O_1 is on the line from T_1 to T_2 when C_1's time equals 8, etc. Connect the successive points O_l and O_{l+1} using straight lines to obtain the green curve. This curve gives the HUYGENS-R offsets between clocks C_i and C_1 at all times after 8 seconds. Since HUYGENS-R is an extrapolation of HUYGENS, it is not defined until some time after HUYGENS has been operational. As defined above, the earliest time at which HUYGENS-R can provide synchronized clocks is when C_1's time equals 8 seconds.

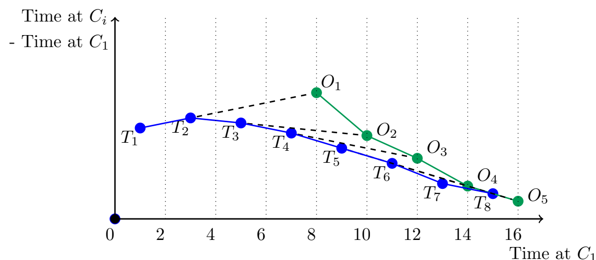
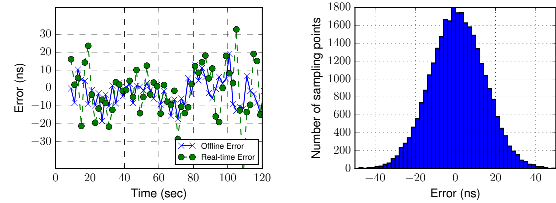


Figure 13: From HUYGENS to HUYGENS-R

Figure 14 quantifies the accuracy of HUYGENS-R using a single NetFPGA. In part (a) of the figure we see that while HUYGENS-R's errors are worse than HUYGENS' errors, they are not much worse. Part (b) of the figure gives a distribution of the errors. Under 40% load, the average value and the 99th percentile of the absolute error are 14.1ns and 43.5ns for HUYGENS-R, slightly larger than the corresponding numbers of 11.0ns and 22.7ns for HUYGENS. These numbers increase to 22.1ns and 55.0ns respectively, versus 14.3ns and 32.7ns for HUYGENS under 80% network load.

Deployment. Even though HUYGENS and HUYGENS-R work effectively at 90% load, the desire for "high availability" in practice may require the use of a dedicated IEEE 802.1 QoS priority for the probes. This not only isolates probe traffic (thereby ensuring that HUYGENS and HUYGENS-R run at all loads), but by giving probe traffic the highest priority, we can also reduce the effect of queueing delays.



(a) A 2 min zoom-in (b) Distribution of errors

Figure 14: HUYGENS-R vs HUYGENS: 16-hour experiment at 40% load with $K = 10$

9 Conclusion

In this paper, we investigated the practicality of deploying accurate timestamping as a primitive service in data center networks, in support of a number of higher-level services such as consistency in replicated databases and congestion control. We set out to achieve synchronization accuracy at the granularity of tens of nanoseconds for all servers in a data center with low overhead among a number of dimensions, including host CPU, network bandwidth, and deployment complexity. When compared with existing approaches to clock synchronization, we aimed to support currently functioning data center deployments with no specialized hardware, except NICs that support hardware timestamping (e.g., PHC). With these goals, we introduced HUYGENS, a probe-based, end-to-end clock synchronization algorithm. By using coded probes, Support Vector Machines and the network effect, HUYGENS achieves an accuracy of 10s of nanoseconds even at high network load. HUYGENS can scale to the whole data center since each server only needs to probe a constant number (10–20) of other servers and the resulting data can be processed in-place. In particular, the parameters and the 2-second update times HUYGENS uses remain the same regardless of the number of servers. In a 40 Gbps data center testbed HUYGENS only consumes around 0.05% of the servers bandwidth and less than 0.44% of its CPU time. Since it only requires hardware timestamping capability which is widely-available in modern NICs, HUYGENS is ready for deployment in current data centers. We are currently exploring the performance of HUYGENS in wide area settings and are seeing promising results.

References

[1] CockroachDB. `https://github.com/cockroachdb/cockroach`. Accessed: 2017-9-22.

[2] PTP hardware clock infrastructure for Linux. `https://www.kernel.org/doc/Documentation/ptp/ptp.txt`. Accessed: 2017-9-22.

[3] Simple Network Management Protocol. `https://en.wikipedia.org/wiki/Simple_Network_Management_Protocol`. Accessed: 2017-9-22.

[4] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Standard 1588* (2008).

[5] IEEE Standard for Local and metropolitan area networks–Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks–Amendment 20: Shortest Path Bridging. *IEEE Standard 802.1AQ* (2012).

[6] IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery. *IEEE Standard 802.1AB* (2016).

[7] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP (DCTCP). In *ACM SIGCOMM computer communication review* (2010), vol. 40, ACM, pp. 63–74.

[8] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W. C., 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: Googles globally distributed database. *ACM Transactions on Computer Systems 31*, 3 (2013), 8.

[9] CORTES, C., AND VAPNIK, V. Support-vector networks. *Machine learning 20*, 3 (1995), 273–297.

[10] JAIN, M., AND DOVROLIS, C. Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement* (2004), ACM, pp. 272–277.

[11] JETWAY COMPUTER CORPORATION. Cableless & Fanless Embedded Barebone Celeron J1900 / 10 Intel Gigabit LAN . `http://www.jetwaycomputer.com/spec/JBC390F541AA.pdf`, 11 2016. Accessed: 2017-9-22.

[12] KESHAV, S. *The packet pair flow control protocol*. International Computer Science Institute, 1991.

[13] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[14] LEE, K. S., WANG, H., SHRIVASTAV, V., AND WEATHERSPOON, H. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 454–467.

[15] LENG, M., AND WU, Y.-C. Distributed clock synchronization for wireless sensor networks using belief propagation. *IEEE Transactions on Signal Processing 59*, 11 (2011), 5404–5414.

[16] LISKOV, B. Practical uses of synchronized clocks in distributed systems. *Distributed Computing 6*, 4 (1993), 211–219.

[17] MAGGS, M. K., O'KEEFE, S. G., AND THIEL, D. V. Consensus clock synchronization for wireless sensor networks. *IEEE sensors Journal 12*, 6 (2012), 2269–2277.

[18] MAHAJAN, R., AND WATTENHOFER, R. On consistent updates in software defined networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks* (2013), ACM, p. 20.

[19] MALLADA, E., MENG, X., HACK, M., ZHANG, L., AND TANG, A. Skewless network clock synchronization without discontinuity: Convergence and performance. *IEEE/ACM Transactions on Networking (TON) 23*, 5 (2015), 1619–1633.

[20] MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal 2014*, 239 (2014), 2.

[21] MILLS, D. L. Internet time synchronization: the network time protocol. *IEEE Transactions on communications 39*, 10 (1991), 1482–1493.

[22] MOREIRA, P., SERRANO, J., WLOSTOWSKI, T., LOSCHMIDT, P., AND GADERER, G. White rabbit: Sub-nanosecond timing distribution over ethernet. In *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication* (2009), IEEE, pp. 1–5.

[23] MURTA, C. D., TORRES JR, P. R., AND MOHAPATRA, P. QRPp1-4: Characterizing Quality of Time and Topology in a Time Synchronization Network. In *IEEE Globecom 2006* (2006), IEEE, pp. 1–5.

[24] OUSTERHOUT, J. K., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G. M., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for ramclouds: scalable high-performance storage entirely in dram. *Operating Systems Review 43*, 4 (2010), 92–105.

[25] PARKINSON, B. W. *Progress in astronautics and aeronautics: Global positioning system: Theory and applications*, vol. 2. AIAA, 1996.

[26] PÁSZTOR, A., AND VEITCH, D. Pc based precision timing without gps. In *ACM SIGMETRICS Performance Evaluation Review* (2002), vol. 30, ACM, pp. 1–10.

[27] PENROSE, R. A generalized inverse for matrices. In *Mathematical proceedings of the Cambridge philosophical society* (1955), vol. 51, Cambridge Univ Press, pp. 406–413.

[28] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 307–318.

[29] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 123–137.

[30] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), vol. 45, pp. 183–197.

[31] SOLIS, R., BORKAR, V. S., AND KUMAR, P. A new distributed time synchronization protocol for multihop wireless networks. In *Decision and Control, 2006 45th IEEE Conference on* (2006), IEEE, pp. 2734–2739.

[32] WATT, S. T., ACHANTA, S., ABUBAKARI, H., SAGEN, E., KORKMAZ, Z., AND AHMED, H. Understanding and applying precision time protocol. In *2015 Saudi Arabia Smart Grid (SASG)* (2015), IEEE, pp. 1–7.

[33] YUASA, H., SATAKE, T., CARDONA, M. J., FUJII, H., YASUDA, A., YAMASHITA, K., SUZAKI, S., IKEZAWA, H., OHNO, M., MATSUZAKI, A., ET AL. Virtual LAN system, July 4 2000. US Patent 6,085,238.

[34] ZHOU, H., NICHOLLS, C., KUNZ, T., AND SCHWARTZ, H. Frequency accuracy & stability dependencies of crystal oscillators. *Carleton University, Systems and Computer Engineering, Technical Report SCE-08-12* (2008).

SnailTrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows*

strymon.systems.ethz.ch

Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, Timothy Roscoe

Systems Group, Department of Computer Science, ETH Zürich

firstname.lastname@inf.ethz.ch

Abstract

We rigorously generalize critical path analysis (CPA) to long-running and streaming computations and present `SnailTrail`, a system built on Timely Dataflow, which applies our analysis to a range of popular distributed dataflow engines. Our technique uses the novel metric of *critical participation*, computed on time-based snapshots of execution traces, that provides immediate insights into specific parts of the computation. This allows `SnailTrail` to work *online* in real-time, rather than requiring complete offline traces as with traditional CPA. It is thus applicable to scenarios like model training in machine learning, and sensor stream processing.

`SnailTrail` assumes only a highly general model of dataflow computation (which we define) and we show it can be applied to systems as diverse as Spark, Flink, TensorFlow, and Timely Dataflow itself. We further show with examples from all four of these systems that `SnailTrail` is fast and scalable, and that critical participation can deliver performance analysis and insights not available using prior techniques.

1 Introduction

We present a generalization of Critical Path Analysis (CPA) to online performance characterization of long-running, distributed dataflow computations.

Existing tools which aggregate performance information from servers and software components into visual analysis and statistics [2, 30] can be useful in showing what each part of the system is doing at any point in time, but are less helpful in explaining which components in a complex distributed system need improvement to reduce end-to-end latency.

On the other hand, tools which capture detailed individual traces through a system, such as Splunk [9] and VMware LogInsight [3], can isolate specific instances of performance loss, but lack a "big picture" view of what really matters to performance over a long (possibly continuous) computation on a varying workload.

In this paper, we show that the design space for useful performance analysis of so-called "big data" systems is much richer than currently available tools would suggest.

Critical Path Analysis is a proven technique for gaining insight into the performance of a set of interacting processes [36], and we review the basic idea in Section 2. However, CPA is not directly applicable to long-running and streaming computations for two reasons. Firstly, it requires a complete execution trace to exist before analysis can start. In modern systems, such a trace may be very large or, in the case of stream processing, unbounded. Secondly, in a continuous computation, there exist many critical paths (as we show later on), which also change over time, and there is no established methodology for choosing one of them. It is therefore important to aggregate the paths both spatially (across the distributed computation) and temporally (as an evolving picture of the system's performance).

According to prior work [5, 37], the accuracy of CPA increases with the number of critical paths considered. However, existing approaches require full path materialization in order to aggregate information from multiple critical paths. Thus, they restrict analysis to k critical paths, where k is much smaller than the total number of paths in the trace. In open-ended computations where analysis is performed on trace snapshots and all paths are of equal length, materializing all paths is impractical, especially if the analysis needs to keep up with real time. For instance, in our experiments, the number of paths in a 10-sec snapshot of Spark traces is in the order of 10^{21}.

This paper's first contributions (in Section 3) are definitions of *Transient Critical Path*, a modification of classical critical path applicable to continuous unbounded computations, and *Critical Participation* (CP), a metric which captures the importance of an execution activity

*This work was partially supported by the Swiss National Science Foundation, Google Inc., and Amadeus SA.

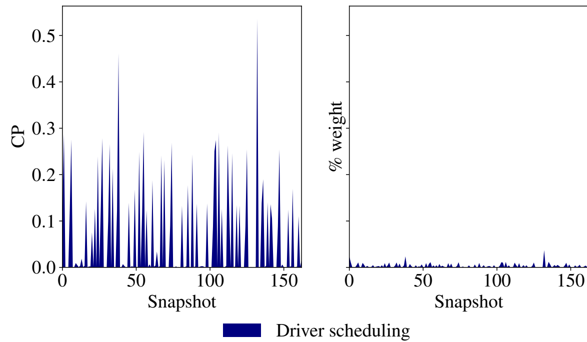


Figure 1: *CP*-based (left) and conventional profiling (right) summaries of Spark's driver activity on BDB [1] from [27] for 64 s snapshots. Spikes indicate coordination between workers and the driver.

in the transient critical paths of computation, and which can be used to generate new time-varying performance summaries. The CP metric can be computed *online* and aggregates information from *all* paths in a snapshot without the need to materialize any path.

Our next contribution (in Section 4) is a model for the execution of distributed dataflow programs sufficiently general to capture the execution (and logging) of commonly-used systems—Spark, Flink, TensorFlow, and Timely Dataflow—and detailed enough for us to define Transient Critical Paths and CP over each of these.

We then show (in Section 5) an algorithm to compute CP online, in real time, and describe `SnailTrail`, a system built (itself as a Timely Dataflow program) to do this on traces from the four dataflow systems listed above. In Section 7 we evaluate `SnailTrail`'s performance, demonstrate online critical path analysis using all four reference systems with a variety of applications and workloads, and show how CP is more informative than existing methods, such as conventional profiling and single-path critical path analysis (Sections 7.4 and 7.5).

Figure 1 gives a flavor of how CP compares with conventional profiling techniques. The key difference is that our approach highlights activities that contribute significantly to the performance of the system, while discarding processing time that lies outside the critical path.

We believe `SnailTrail` is the first system for online real-time critical path analysis of long-running and streaming computations.

2 Critical Path Analysis background

CPA has been successfully applied to high-performance parallel applications like MPI programs [11, 32], and the basic concepts also apply to the distributed dataflow systems we target in this paper. In this section we review classical CPA applied to batch computations as a prelude to our extension of CPA to online and continuous computations in the next section. Table 1 summarizes the notation we use in this section and the rest of this paper.

We view distributed computation as executed by individual system *workers* that perform *activities* (e.g. data transformations or communication). The *critical path* is defined as the sequence of activities with the longest duration throughout the execution. More formally:

Definition 1 Activity: *a logical operation performed at any level of the software stack, and associated with two timestamps* $\texttt{[start,end]}$, $\texttt{start} \leq \texttt{end}$*, that denote the start and end of its execution with respect to a clock* C.

An activity can be either an operation performed by a worker (*worker activity*) or a message transfer between two workers (*communication activity*). Typically, worker activities correspond to the execution of some code, but can also be I/O operations performed by the worker (e.g. reads/writes to/from disk). Communication activities correspond to worker interactions, e.g. message passing.

Different systems have different concepts (threads, VMs, etc.) corresponding to workers. For consistency, we define workers as follows:

Definition 2 Worker: *a logical execution unit that performs an ordered sequence of activities with respect to a clock* C.

We require that no two activities of the same worker a_i:$\texttt{[start}_i\texttt{,end}_i\texttt{]}$ and a_j:$\texttt{[start}_j\texttt{,end}_j\texttt{]}$ (where $i \neq j$) can overlap in time, i.e. either $\texttt{end}_i \leq \texttt{start}_j$ or $\texttt{start}_i \geq \texttt{end}_j$.

Central to CPA is the *Program Activity Graph* (PAG):

Definition 3 Program Activity Graph: *A PAG* $G = (V, E)$ *is a directed labeled acyclic graph where:*

- *V is the set of vertices. A vertex $v \in V$ represents an event corresponding to the start or end of an activity. Each vertex v has a* timestamp $v[t]$ *and a* worker id $v[w]$.
- *$E \equiv E_w \cup E_c \subset V \times V$, $E_w \cap E_c = \emptyset$, is the set of directed edges. An edge $e = (v_i, v_j) \in E$ represents an activity a:$\texttt{[start,end]}$, where $v_i[t] = \texttt{start}$ and $v_j[t] = \texttt{end}$. An edge e has a type $e[p]$ and a weight $e[w]$ indicating the activity duration in time units, so that $e[w] = v_j[t] - v_i[t] = \texttt{end} - \texttt{start} \geq 0$. An edge $e \in E_w$ denotes a worker activity whereas an edge $e \in E_c$ denotes a communication activity.*

The direction of an edge $e = (v_1, v_2) \in E$ from node $v_1 \in V$ to node $v_2 \in V$ denotes a `happened-before` relationship between the nodes [24]. The critical path is then defined as the *longest path* in the program activity graph:

Symbol	Description
a:[start,end]	Activity a with start and end timestamps
G	Activity graph
$G_{[t_s,t_e]}$	Snapshot of activity graph G in the time interval $[t_s, t_e]$
$\prod_{t_s}^{t_e}(e)$	Projection of edge e on the time interval $[t_s, t_e]$
$v[w]$	Worker id of vertex v
$v[t]$	Timestamp t of vertex v
$e[w]$	Weight w of edge e
$e[p]$	Type p of edge e
$\|\vec{p}\|$	Total weight of edges in path $\vec{p}$
E_w	Set of worker activities
E_c	Set of communication activities
$c(e)$	transient path centrality of edge e
CP_e	critical participation of edge e

Table 1: Notation used throughout this paper

Definition 4 Critical Path: *Given a program activity graph* $G = (V,E)$*, the critical path is a path* $\vec{p} \in G$ *such that* $\nexists \vec{p}' \in G : \|\vec{p}'\| > \|\vec{p}\|$*, where* $\|\vec{p}\| = \sum_{\forall e \in \vec{p}} e[w]$ *and* $\|\vec{p}'\| = \sum_{\forall e \in \vec{p}'} e[w]$ *is the sum of all edge weights in* $\vec{p}$ *and* $\vec{p}'$ *respectively.*

3 Online Critical Path Analysis

Offline processing in traditional CPA is not feasible for long-running or continuous computations like streaming applications or machine learning model training. In these cases, neither the program activity graph nor the critical path can be defined as in Section 2.

Instead, we define online CPA on PAG *snapshots*, performing it on user-defined *time windows*: slices of the PAG that contain activities within a specified time interval. This enables not only performance analysis of running applications, but also targeting specific parts of the computation like the model training phase in a TensorFlow program or a specific time window in a Flink stream.

To achieve this, we show here how to define a time-based program activity graph snapshot and a *transient critical path* on this graph. We then define the *critical participation* performance metric, and we provide the intuition behind it in Section 3.3.

3.1 Transient Critical Paths

To retrieve a snapshot of the PAG, we first assign activities to time windows. Given an edge in a graph, we call its corresponding edge in a snapshot an *edge projection*:

Definition 5 Edge Projection: *Let* $e = (v_i, v_j)$ *be an edge of an activity graph* $G = (V,E)$*, where* $e \in E$ *and* $v_i, v_j \in V$*. Let also* $[t_s,t_e]$*,* $t_s \le t_e$*, be a time interval with respect to a clock* C*. Let* u_s *be a copy of* v_i *with* $u_s[t] = t_s$ *and* u_e *a copy of* v_j *with* $u_e[t] = t_e$*. The projection of* e *on* $[t_s,t_e]$ *is an edge of the same type as* e *and is defined only whenever* $\left[v_i[t], v_j[t]\right]$ *overlaps with* $[t_s,t_e]$ *as follows:*

$$\prod_{t_s}^{t_e}(e) = \left(\arg\max_{[t]}(v_i,u_s), \arg\min_{[t]}(v_j,u_e)\right)$$

Activities entirely within the time interval $[t_s,t_e]$ are unchanged by the projection, whereas activities that straddle the boundaries are truncated to fit the interval. We can now define a snapshot as follows:

Definition 6 PAG Snapshot: *Let* $G = (V,E)$ *be a program activity graph, and* $[t_s,t_e]$*,* $t_s \le t_e$*, be a time interval with respect to a clock* C*. The snapshot of* G *in* $[t_s,t_e]$ *is a directed labeled acyclic graph* $G_{[t_s,t_e]} = (V',E')$ *that is constructed by projecting all edges of* G *on* $[t_s,t_e]$*.*

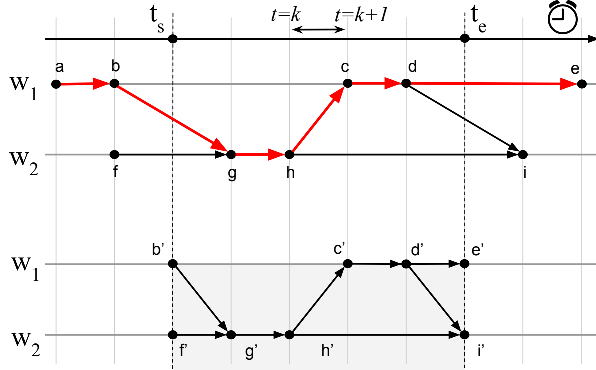
The snapshot $G_{[t_s,t_e]}$ is that part of the PAG which can be observed in the time window $[t_s,t_e]$. Figure 2a shows this applied to the activity timelines of two worker threads, w_1 and w_2, with time flowing left to right. The complete PAG is shown at the top with the critical path in red. Below is the projection of the PAG into the interval $[t_s,t_e]$. The activities straddling the window (e.g. $\prod_{t_s}^{t_e}(b,g) = (b',g')$) are projected to fit in the snapshot.

The key observation is that we cannot define a single critical path in a PAG snapshot since there exist *multiple* longest paths with the same total weight: $t_e - t_s$. All paths starting at t_s and ending at t_e are *potentially* parts of the evolving global critical path. For this reason, we define the notion of *transient critical path*:

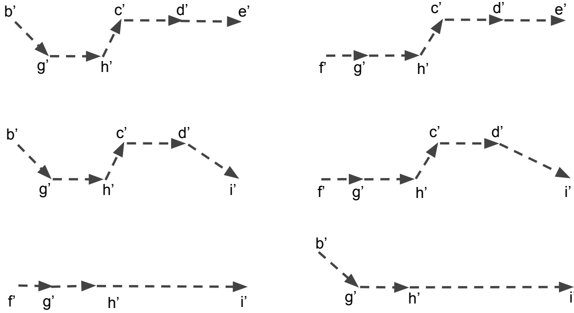
Definition 7 Transient Critical Path: *Let* $G_{[t_s,t_e]} = (V,E)$ *be the snapshot of an activity graph* G *in the time interval* $[t_s,t_e]$*. We define the set of paths* $\mathcal{P}$ *on* $G_{[t_s,t_e]}$ *as* $\mathcal{P} \equiv \{\vec{p} \subseteq E \mid \nexists \vec{p}' : \|\vec{p}'\| > \|\vec{p}\|\}$*, where* $\vec{p}$ *denotes a path in* $G_{[t_s,t_e]}$*, and* $\|\vec{p}\|$ *denotes the total weight of all edges in* $\vec{p}$*, i.e.,* $\|\vec{p}\| = \sum_{\forall e \in \vec{p}} e[w]$*.*

Any path $\vec{p} \in \mathcal{P}$ *is a transient critical path of the activity graph* G *in the time interval* $[t_s,t_e]$*.*

Figure 2b shows all six transient critical paths for the snapshot in Figure 2a. Since each could potentially participate in the evolving global critical path, we need a metric that can aggregate information from all paths and rank activities according to their impact on computation performance. In offline CPA such a ranking is trivial since there is only one critical path for the entire computation.



(a) Program activity timelines of a distributed execution with two workers. The vertical lines divide the timeline into intervals of one time unit. The critical path is highlighted in red in the top timeline. The bottom timeline shows the PAG snapshot into the time interval $[t_s, t_e]$.



(b) Transient critical paths for the graph snapshot of Figure 2a.

Figure 2: A program activity graph, its snapshot in the interval $[t_s, t_e]$, and its transient critical paths.

Since all transient paths can potentially be part of the evolving global critical path, an activity that appears on many transient paths is more likely to be critical and should be ranked high. In Figure 2b, edge (d', i') appears in two paths, while edge (g', h') belongs to all six. The performance metric we define next incorporates this information and ranks activities based on their potential contribution to the global critical path.

3.2 Critical Participation (CP metric)

Given the duration of an activity $e[w]$ and the total length $\|\vec{p}\|$ of the critical path $\vec{p}$, the *participation* of e to $\vec{p}$ is defined as:

$$q_e = \frac{e[w]}{\|\vec{p}\|} \in [0,1] \tag{1}$$

and is easily computed for all activities in a single $\vec{p}$ pass.

We correspondingly define average *critical participation* (CP) of an activity e in a transient critical path as:

$$CP_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} \in [0,1] \tag{2}$$

where q_e^i is the participation of e to the i-th transient critical path (given by Eq. 1), and N is the total number of transient critical paths in the graph snapshot.

A straightforward way to compute CP_e is to materialize all N transient paths and compute the participation of each activity in every path. However, path materialization is not viable in an online setting because a single graph snapshot might contain too many paths to maintain. Instead, we exploit the fact that the CP of an activity actually depends on the total number of transient paths this activity belongs to. Hence, we define the *transient path centrality* as follows:

Definition 8 Transient Path Centrality: *Let $\mathcal{P} = \{\vec{p}_1, \vec{p}_2, ..., \vec{p}_N\}$ be the set of N transient paths of snapshot $G_{[t_s,t_e]}$ with length $\|\vec{p}\| = t_e - t_s$. The transient path centrality of an edge $e \in G_{[t_s,t_e]}$ is defined as*

$$c(e) = \sum_{i=1}^{N} c_i(e), \text{ where } c_i(e) = \begin{cases} 0 : e \notin \vec{p}_i \\ 1 : e \in \vec{p}_i \end{cases}$$

The following holds:

$$CP_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} = \frac{c(e)}{N} \cdot \frac{e[w]}{\|\vec{p}\|} \tag{3}$$

Eq. 3[1] indicates that the computation of CP_e can be reduced to the computation of $c(e)$, which requires no path materialization and can be performed in parallel for all edges in $G_{[t_s,t_e]}$. Section 5 provides an algorithm for transient path centrality and CP without materialization. Note that we can normalize by the number of paths N and their length $\|\vec{p}\|$ because of Definition 7 guaranteeing that all paths have the same length.

We can now compute the transient path centrality and critical participation for the example in Figure 2. For instance, $c(d', i') = 2$ and $c(g', h') = 6$. Respectively, since $t_e - t_s = 5$ and $N = 6$, $CP_{(d',i')} = 0.066$ and $CP_{(g',h')} = 0.2$.

The CP of Eq. 2 can be generalized for activities of a specific type c as:

$$\sum_{\forall e: e[p]=c} CP_e \tag{4}$$

and the following holds[1]:

$$\sum_{\forall c \in G} \sum_{\forall e: e[p]=c} CP_e = 1 \tag{5}$$

Intuitively, Eq. 5 states that the estimated contribution of an activity type, e.g., serialization, to the critical path of the computation is *normalized* over the contribution of all other activity types in the same snapshot.

[1] We provide proofs of Eqs. 3 and 5 in the Appendix A.3.

3.3 Comparison with existing methods

Figure 3 illustrates by example a comparison of CP-based performance analysis with two existing methods: conventional profiling and traditional critical path analysis.

Conventional profiling summaries aggregate activity durations by type or by worker timeline. Such summaries provide information on how much time (i) a program spends on a certain activity type (e.g. serialization) or (ii) a worker spends executing an activity type as compared to other workers. Since conventional profiling summaries rely solely on durations and do not capture execution dependencies, they cannot reveal bottlenecks and execution barriers. Conventional profiling in the execution of Figure 3 would rank activities (a,b) and (c,d) high since they both have a duration of 3 time units, larger than all other activities. However, optimizing those activities cannot result into any performance benefit for the parallel computation as they are both followed by a waiting state (denoted with a dashed line).

On the other hand, CPA captures execution dependencies and can accurately pinpoint activities which influence performance. However, traditional CPA is not directly applicable in a continuous computation as the critical path is not known by just inspecting a snapshot of the execution traces. In a snapshot like the one of Figure 3, all paths starting at s_i and finishing at e_i have equal length in time units, thus traditional CPA would choose one of them at random. We have highlighted such a path in Figure 3 in red color. Although this randomly selected path does not contain the activities (a,b) and (c,d), whose optimization would certainly not improve the latency of the computation, it misses several important activities, such as (x,u) and (v,z), whose optimization would do so.

The CP metric overcomes the limitations of both conventional profiling and traditional CPA by ranking activities based on their potential contribution to the evolving critical path of the computation, which in turn reflects potential benefits from optimization.

Given a snapshot and no knowledge of the execution timelines outside of it, any path between the s_i and e_i points in Figure 3 is *equally probable* to be part of the critical path. CP is a *fairer* metric compared to existing methods in that it aggregates an activity's contribution over *all* transient critical paths and normalizes by the number of paths and the activity's duration. The more paths an activity contributes to, the higher the probability it is a part of the evolving critical path and, hence, the higher its CP metric is. In Figure 3, activities (a,b) and (c,d) do not contribute to any path and thus have zero transient path centrality and CP values. On the other hand, activities (x,u), (u,v), and (v,z) will be ranked as top-three by CP, since they participate in six, nine, and six transient critical paths respectively.

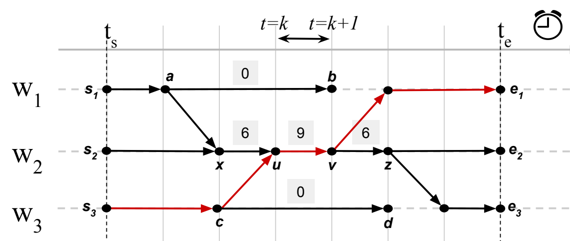


Figure 3: A program activity graph snapshot with three workers. The vertical lines divide the timeline into intervals of one time unit. A randomly chosen critical path is highlighted in red. Edge annotations correspond to transient path centrality (Definition 8).

In Section 7.4 we empirically compare CP-based performance summaries to conventional profiling and traditional CPA, and demonstrate how the results of the latter can be misleading. Further, in Section 7.5, we show how CP can detect and help optimize execution bottlenecks like the one represented by activity (u,v) in Figure 3.

4 Applicability to dataflow systems

Here we show the applicability of our applicability to a range of modern dataflow systems. We provide details on the model assumptions and the instrumentation requirements in the Appendix.

Spark, Flink, TensorFlow, and Timely are superficially different, but actually similar with regard to CPA: all execute dataflow programs expressed as directed graphs whose vertices are operators (e.g. map, reduce) and whose edges denote data dependencies. During runtime, a logical dataflow graph is executed by one or more workers, which can be threads or processes in a machine or a cluster. Each worker has a copy of the graph and processes a partition of the input data in parallel with other workers.

4.1 Activity types

We define a small set of *activity types* we use to classify both the activity of a worker at any given point in time, and communication of data between workers/operators. We consider the following types of *worker* activities:

Data Processing: The worker is computing on data in an operator, which usually has a unique ID. We also include low-level (de)compression operations.

Scheduling: Deciding which operator a worker will execute. In Spark and Flink, scheduling is done by special workers (the Driver and the JobManager).

Barrier Processing: The worker is processing information which coordinates the computation (e.g distributed progress tracking in Timely or watermarks in Flink).

Buffer Management: The worker is managing buffers between operators (e.g. Flink's FIFO queues) or buffering data moving to/from disk (e.g. Spark). The activity may include copying data into/out of buffers, locking, recycling buffers (e.g. Flink) and dynamically allocating them (e.g. Timely).

Serialization: Data is being (un)marshaled, an operation common to all dataflow systems when messages are sent between processes and/or machines.

Waiting: The worker is waiting on some message (data or control) from another worker, and is therefore either *spinning* (as in Timely) or *blocked* on an RPC (as in TensorFlow). Waiting in our model is always a consequence of other, concurrent, activities [21], and so is a key element of critical path analysis: a worker does not produce anything *useful* while waiting, and so *waiting activities can never be on the critical path*.

I/O: The worker is waiting on an external (uninstrumented) system, (e.g. Spark waiting for HDFS, or Flink spilling large state to disk). I/O activities have no special meaning, but capture cases where performance of the reference system is limited by an external system.

Unknown: Anything else: gaps in trace records and any worker activity not captured by the instrumentation. A large number of unknown activities usually indicates inadequate instrumentation [21].

In contrast, interaction between workers is modeled as a *communication* activity, which captures either: (i) **application data exchange** over a communication channel, or (ii) **control messages** conveying metadata about worker state or progress and exchanged between pairs of workers (as in Timely) or through a master (as in Spark, Flink).

4.2 Instrumenting specific systems

We applied our approach to Spark, TensorFlow, Flink, and Timely Dataflow, mapping each to our taxonomy of activities. In some cases we used existing instrumentation, whereas in others we added our own. Space precludes a full discussion of either the structure of these systems or their instrumentation; we provide only brief summaries here and we give more details in [21].

Timely Dataflow [26] required us to add explicit instrumentation, and was the first system we addressed (in part because `SnailTrail` is written in Timely). Timely's progress tracking corresponds to our "barrier" activity, discrete (de)serialization is performed on both data records and control messages, and Timely's cooperative scheduling means that any otherwise unclassified worker activity corresponds to "scheduling".

Apache Flink [10] adopts (unlike Timely) a master-slave architecture for coordination. We treat Flink's

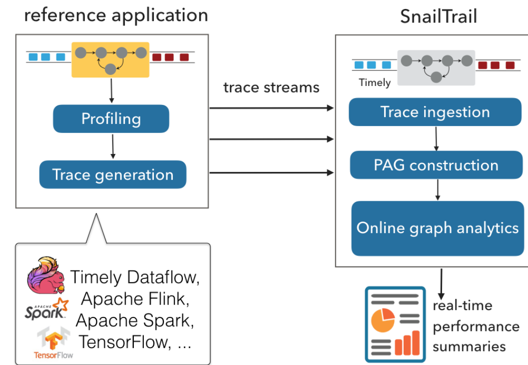


Figure 4: `SnailTrail` overview.

`JobManager`, `TaskManagers`, and `Tasks` all as workers, and Flink's runtime has clear activities corresponding to buffer management and serialization. Scheduling is performed in the `JobManager`, barrier processing corresponds to the watermark mechanism, and control messages correspond to communication between the `JobManager` and `TaskManagers`.

TensorFlow [4] has its own instrumentation based on "Timeline" objects, which we reuse unchanged. While enough to generate meaningful results, it also shows how even a well-considered logging system can easily omit information vital for sophisticated performance analysis.

Spark [38] also has native instrumentation which we use to model both the Spark driver and executors as workers. The logs provide information on the lineage of Resilient Distributed Datasets (RDDs) facilitating construction of the PAG. Since executor scheduling is not instrumented, we assume greedily that a task is started on the most recently used thread, which aligns with Spark's observed behavior.

5 `SnailTrail` system implementation

CP is implemented in `SnailTrail`, itself a data-parallel streaming application written in Rust using Timely Dataflow (Figure 4). It reads streams of activity traces via sockets, files, or message queues from a reference application and outputs a stream of performance summaries. `SnailTrail` operates in four pipeline stages: it (i) ingests logs, (ii) slices the stream(s) into windows $[t_s, t_e]$ and constructs PAG snapshots, (iii) computes the CP of the snapshots, and (iv) outputs the summaries we show in Section 6.

Traces are sent to `SnailTrail` which ingests a stream S of performance events corresponding to vertices in the activity graph. The snapshots are constructed using Algorithm 1. First, `SnailTrail` extracts from S the events in

the time window $[t_s, t_e]$ (line **1**). These are then grouped by the worker that recorded them (line **2**). Each group corresponds to a worker timeline in Figure 2a. Then, `SnailTrail` sorts the events in each timeline by time (line **4**), and scans each timeline in turn to create the set of edges E_w (line **6**) that correspond to worker activities (cf. Section 4.1). Meanwhile, communication activities are partially initialized based on `send` and `receive` at each worker (line **7**). Then (line **8**), partial edges are grouped by the attributes $\left(w_{id}^{src}, w_{id}^{dst}, c_{id}\right)$; note that w_{id}^{src} is the sender worker id, w_{id}^{dst} is receiver id, and c_{id} is generated to uniquely identify a message. These pairs of partial edges are concatenated to create the final communication edges in E_c, and the output is the union of sets E_w and E_c (line **9**).

Algorithm 1: Graph Snapshot Construction

Input : A stream S of logs and a window $[t_s, t_e]$;
Output : The graph snapshot $G_{[t_s,t_e]}$;

1. let $S_{[t_s,t_e]}$ be the logged events from S in $[t_s, t_e]$;
2. group events in $S_{[t_s,t_e]}$ by worker;
3. **for** *each worker timeline in* $S_{[t_s,t_e]}$ **do**
4. sort events by time;
5. scan events and generate:
6. (a) the set E_w of edges for worker activities;
7. (b) a set E_h of half edges for `send` and `receive` events;
8. group half edges in E_h by $\left(w_{id}^{src}, w_{id}^{dst}, c_{id}\right)$ and create the set E_c of edges for communication activities;
9. **return** $E_w \cup E_c$

Algorithm 1 requires two shuffles of the incoming log stream: one on worker id (before line **2**), and a second on the triple $\left(w_{id}^{src}, w_{id}^{dst}, c_{id}\right)$ (before line **8**). The most expensive step is sorting the timeline (line **4**), requiring $O(|T| \cdot \log|T|)$ time, where $|T|$ is the number of events in the timeline. Parallelism is limited by the number of workers in the reference system (usually many more than `SnailTrail`) and the density of the graph. We emphasize that edges in the PAG represent real `happened-before` dependencies given by the instrumentation. More details in the way edges are created in lines **6-7** are given in the Appendix along with a discussion on clock alignment.

For each graph snapshot, the CP metric is computed using Algorithm 2. `SnailTrail` collects 'start' and 'end' nodes (lines **1-2**) as seeds to traverse $G_{[t_s,t_e]}$. V_s (resp. V_e) includes the node(s) with the minimum (resp. maximum) timestamp $v[t]$ in $G_{[t_s,t_e]}$. Typically, $|V_s| = |V_e| = \ell$, where ℓ is the number of timelines, and so all nodes in V_s have timestamp t_s whereas all nodes in V_e have timestamp t_e.

Algorithm 2 computes the transient path centrality $c(e)$ of Eq. 3 for all edges in $G_{[t_s,t_e]}$. Observe that $c(e) = c_1 \cdot c_2$, where c_1 is the number of paths from the source of e to any node in V_s, and c_2 is the number of paths from the destination of e to any node in V_e. The algorithm thus performs two simple traversals of $G_{[t_s,t_e]}$ in parallel, computing c_1 and c_2 for each edge (lines **3-4**). Each traversal outputs pairs (e, c_i) and these are finally grouped by e to give CP values (lines **6-7**).

Note that, while traversing $G_{[t_s,t_e]}$, we visit each edge in $G_{[t_s,t_e]}$ *only once* by propagating the *final* value c_1 (resp. c_2) from each edge to all its adjacent edges. This reduces the intermediate results of the computation significantly. We compute the *CP* according to Equation 3, which does not require path materialization.

Algorithm 2 requires two partitions of $G_{[t_s,t_e]}$: one on source, and one on destination ids. Worst-case time complexity is $O(d)$, where d is the diameter of $G_{[t_s,t_e]}$ in number of edges, i.e., the maximum number of edges in any transient critical path.

Algorithm 2: Critical Participation (CP Metric)

Input : An activity graph snapshot $G_{[t_s,t_e]} = (V, E)$;
Output : A set $S = \{(e, \mathrm{CP}) \mid e \in G_{[t_s,t_e]}\}$ of *CP* values;

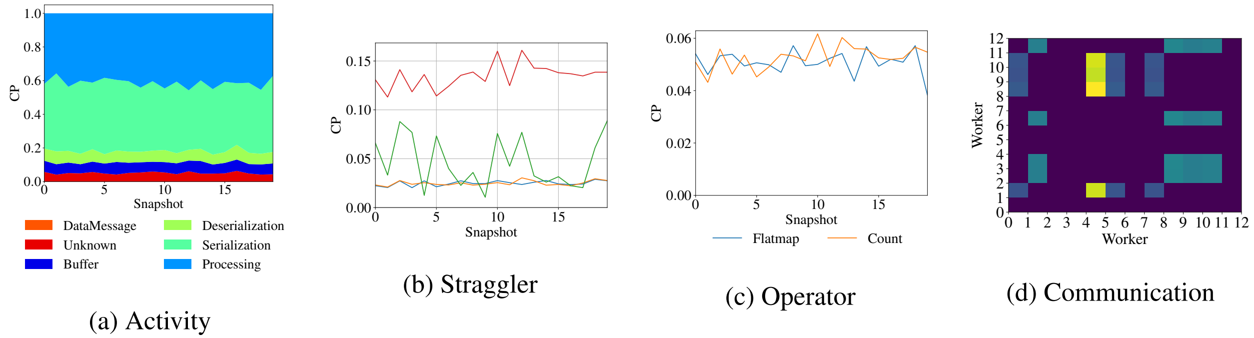
1. let $V_s \equiv \{v \in V \mid \nexists v' \in V : v'[t] < v[t]\}$; `//start nodes`
2. let $V_e \equiv \{v \in V \mid \nexists v' \in V : v'[t] > v[t]\}$; `//end nodes`
   `//Both traversals are performed in parallel`
3. traverse $G_{[t_s,t_e]}$ starting from V_s, and count the total number of times each edge is visited, let c_1;
4. traverse $G_{[t_s,t_e]}$ backwards, starting from V_e, and count the total number of times each edge is visited, let c_2;
5. $S = \emptyset$;
6. **for** *each edge* $e \in E$ **do**
7. $S = S \cup \{(e, \frac{c_1 \cdot c_2 \cdot e[w]}{N \cdot (t_e - t_s)})\}$
8. **return** S

Performance summaries are constructed by user-defined groupings on the edge attributes and summing CP values over each group.

`SnailTrail`'s accuracy depends on the quality of the instrumentation. A more complete set of dependencies increases the accuracy of the *CP* metric. We leave a worst-case error bound analysis for future work.

6 CP-based performance summaries

The CP metric provides an indication of an activity's contribution to the evolving critical path. `SnailTrail` can be configured to generate different types of performance summaries using the CP metric. Each summary type targets a specific aspect of an application's performance and is designed to reveal a certain type of bottleneck. In particular, `SnailTrail` provides four performance summaries which can answer four types of questions: (i) *Which activity type is on the critical path?* (ii) *Is there data skew?* (iii) *Is there computation skew?* (iv) *Is there communication*



(a) Activity
(b) Straggler
(c) Operator
(d) Communication

Figure 5: Examples of SnailTrail summary types for the Dhalion [18] benchmark on Flink with 1s snapshots.

skew? The performance summaries not only indicate potential bottlenecks, but also provide immediate actionable feedback on which activities to optimize, which workers are overloaded, which dataflow operator to re-scale, and how to minimize network communication.

Figure 5 shows examples of the four summary types for the Dhalion [18] benchmark on Flink with 1s snapshots. In the rest of this section, we describe each summary type in detail and we discuss how to use them in practical scenarios to improve an application's performance.

Activity summary. *Is the fault-tolerance mechanism in the critical path when taking frequent checkpoints? Is coordination among parallel workers an overhead when increasing the application's parallelism?* An activity summary can answer this sort of questions about an application's performance. This summary plots the proportional CP value of selected *activity types* with respect to the other activity types in a given snapshot. Activity reveal bottlenecks inherent to the system or its configuration. Having a ranking of activity types based on their critical participation essentially gives us an indication on which activities have the higher potential for optimization benefit. For example, if we find that serialization is on the critical path, we might want to try a different serialization library. The activity summary ranking can also help us choose good configurations for our application, like how to adjust the checkpoint interval or the parallelism. The activity summary of Figure 5 shows that serialization and processing have the higher potential for optimization. Activity summaries can be configured to plot selected activities only, as in Figure 1 where we only show the Spark driver's scheduling.

Straggler summary. *Is there data skew? If so, which worker is the straggler?* `SnailTrail` can answer these questions with a straggler summary, which plots the critical participation of a worker's *timeline* in a certain snapshot. The straggler summary relies on the observation that if a worker is a straggler then many transient critical paths pass through its timeline. Hence, we can compare how critical a worker's activities are as compared to the other workers in the computation and reveal computation imbalance. This ranking can serve as input to a work-stealing algorithm or guide a data re-distribution technique. The straggler summary of Figure 5 clearly shows one straggler worker in the Flink job. In Section 7.5, we look closer into detecting skew with `SnailTrail`.

Operator summary. *Will re-scaling my dataflow improve performance? And if yes, which operator in the dataflow to re-scale?* An operator summary plots the critical participation of each operator's processing activity in a snapshot, normalized by the number of parallel workers executing the operator. This summary reveals bottlenecks in the dataflow caused by resource underprovising and serves as a good indicator for scaling decisions. Traditional profiling methods fail to detect that an operator might be limiting the end-to-end throughput of a dataflow even if its parallel tasks are perfectly balanced. Such bottlenecks are hard to detect by looking at traditional metrics such as queue sizes, throughput, and backpressure. The operator summary of Figure 5 shows that both operators have similar critical participation, thus the parallelism of the job is properly configured. In Section 7.5, we present a detailed use-case where operator summaries guide scaling decisions for streaming applications.

Communication summary. *Is there communication skew? And if yes, which communication channels to optimize?* A communication summary plots the critical participation of communication activities between each pair of workers within a given snapshot. Contrary to traditional communication summaries, this CP-based summary does not rely on communication frequency or absolute message sizes. Instead, it ranks communication edges by their critical importance: the more often a communication edge belongs to a transient critical path, the higher it will be ranked by the summary. Communication summaries can be used to minimize network delays and optimize distributed task placement. If we find that a pair of workers' communication is commonly on the critical path, it

is probably a good idea to physically deploy these two workers on the same machine. For example, the communication summary of Figure 5 indicates that colocating worker 5 with workers 11-13 could benefit performance.

7 Evaluation

To show generality, we evaluate SnailTrail analyzing four different reference systems: Timely Dataflow (version 0.1.15), Apache Flink (1.2.0), Apache Spark (2.1.0), and TensorFlow (1.0.1). Our evaluation is divided into four categories. First, in Section 7.2 we show the instrumentation SnailTrail needs does not cause significant impact on the performance of reference systems. Second, in Section 7.3 we investigate SnailTrail's performance and show it can deliver results in real time with high throughput and low latency. Third, we compare the quality of SnailTrail's analysis and the utility of the *CP* metric with both conventional profiling and traditional critical path analysis (Section 7.4). Finally, we present use cases for SnailTrail with analysis results (Section 7.5).

7.1 Experimental setting

SnailTrail uses the latest Rust version of Timely Dataflow [25] compiled with Rust 1.17.0. In all experiments, SnailTrail ran on an Intel Xeon E5-4640 2.40 GHz machine with 32 cores (64 threads) and 512G RAM running Debian 7.8 ("wheezy"), and was configured to produce results by ingesting execution traces from a reference system on a different cluster.

Benchmarks. We compare SnailTrail to existing approaches with several traces generated by Flink, Spark, and TensorFlow using the following benchmarks. For Flink, we use the Yahoo Streaming Benchmark (YSB) [12] and the WordCount benchmark of Dhalion [18]. For Spark, we use YSB and, for TensorFlow, we use the AlexNet [23] program on ImageNet [29]. To evaluate SnailTrail performance we use Flink (configured with 48 parallel tasks) running a real-world *sessionization* program on a 10min window of operational logs from a large industrial datacenter. This generates a trace with a median number of 30K events per second (around 7.5M events for a 256s snapshot, the largest we used). We also show the instrumentation overhead in Flink, with the same sessionization experiment, and Timely, using a PageRank computation with 16 parallel workers on a random graph.

7.2 Instrumentation Overhead

SnailTrail relies on tracing functionality in the reference system, and this incurs performance overhead. To evaluate the overhead of the instrumentation we added, we implemented a streaming analytic job, *sessionization*, in Flink and an iterative graph computation, PageRank, in Timely, and measured performance with tracing enabled and disabled. For TensorFlow and Spark we use their existing, and somewhat incomplete, tracing facilities.

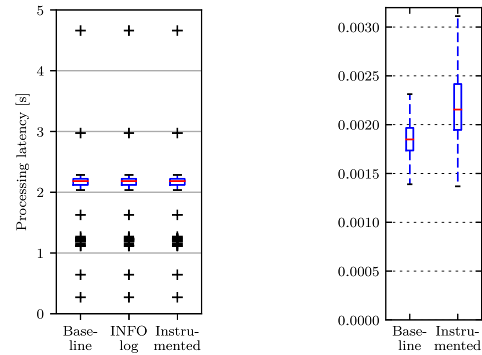


Figure 6: Latency with and without instrumentation for Flink (left) and Timely (right)

Figure 6 shows box-and-whisker plots of processing latency for Flink and Timely implementations. Individual bars correspond to the cases where logging is completely turned off (*baseline*), the default logging level (*info*), and our detailed tracing (*instrumented*).

Flink shows a statistically significant difference of 9.7 % (±1.43 %) additional mean latency, or 203ms (±29.9 µs) in absolute terms, at 95% confidence. This overhead is negligible, given that Flink typically runs with logging enabled in production deployments.

For Timely, there is a statistically significant difference of 13.9 % (±5.5 %) increase in the mean latency, or 319 µs (±126.2 µs) in absolute terms, at 95% confidence.

Experiments with Spark and TensorFlow showed no discernible overhead for collecting the traces required by SnailTrail. Overall, we argue that performance penalties around 10% are an acceptable tradeoff for greater insight, and could be additionally amortized in some cases.

7.3 SnailTrail Performance

We evaluate SnailTrail's performance to demonstrate that (i) it always operates online and thus provides feedback to the running reference applications in real-time and (ii) its analysis scales to large deployments of reference applications without violating this online requirement.

Latency. We require SnailTrail to be capable of constructing the PAG and computing the CP metric for a snapshot of size *x secs* in less than *x secs*. The number of events in a snapshot depends on (i) the snapshot duration and (ii) the instrumentation granularity of the reference system. For this experiment, we vary the number of events

snapshot size	1	2	4	8	16	32	64	128	256
latency	0.06	0.14	0.29	0.62	1.40	2.93	5.91	13.16	24.84
#events	0.03	0.06	0.12	0.24	0.48	0.94	1.91	3.76	7.5

Table 2: SnailTrail's median latency per snapshot (s) for the online analysis of different snapshot intervals (s). The last row shows the median number of events (millions) per snapshot.

snapshot size	1	2	4	8	16	32	64	128	256
throughput	1.2	1.2	1.2	1.1	1.1	1.0	0.8	0.5	0.4
latency	0.7	1.4	3.2	7.1	10.1	10.2	16.8	24.9	30.8

Table 3: SnailTrail's maximum achieved throughput (millions of processed events per second) and corresponding latency per snapshot (s) for the online analysis of different snapshot intervals (s).

in the snapshot by increasing its duration from $1s$ to $256s$ (in powers of 2) and we run SnailTrail on the Flink sessionization job trace, which is the densest one we have. Note that the public Spark traces from real-world cloud deployments [27] are not as dense as the ones generated by the Flink streaming computations we run.

We show median latency and number of events per snapshot in Table 2; SnailTrail is always capable of operating online and its latency increases almost linearly with the snapshot duration. Specifically, it can process 1s of input logs in 6ms and 256s of input logs in under 25s.

Throughput. To evaluate SnailTrail's throughput, we interleave the processing of multiple snapshots to increase the number of events sent to the system. Table 3 shows the maximum achieved throughput (number of processed events per second) while respecting the online requirement and the corresponding latency for processing an input snapshot, including PAG construction and CP computation. For 1s snapshots, SnailTrail can process 1.2 million events per second; a throughput *two orders* of magnitude larger than the event rate we observed in all log files we have, including the Spark traces from [27]. SnailTrail comfortably keeps up with all tested workloads: the time to process a snapshot is always smaller than the snapshot's duration. Throughput decreases when increasing the snapshot size since the PAG gets bigger.

7.4 Comparison with existing methods

We examine how useful the CP-based summaries produced by SnailTrail are in practice, as compared to the weight-based summaries produced by conventional profiling, where activities are simply ranked by their total

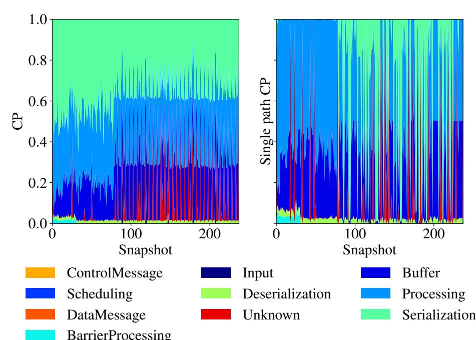


Figure 7: CP-based (left) and single-path (right) summaries for Flink on YSB (1s snapshots).

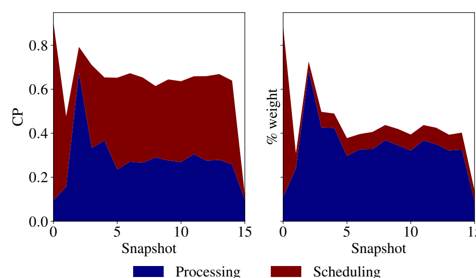


Figure 8: CP-based (left) and conventional profiling (right) summaries for Spark on YSB [12] (8s snapshots).

duration, and the single-path summaries, where CP is computed on a single transient critical path (in this experiment selected at random). We show examples of such summaries in Figures, 7, 8, and 9 for Flink, Spark, and TensorFlow, along with the configuration of each system.

First note that single-path summaries correspond to a straight-forward application of traditional CPA on trace snapshots where only a single path is chosen at random. The plot on the right of Figure 7 exhibits high variation because different transient critical paths may consist of completely different activities, even within the same graph snapshot. In contrast, CP is a *fairer* metric that avoids this misleading critical activity "switching" by aggregating information from all transient critical paths in a snapshot.

Conventional profiling summaries are different from CP-based summaries in that they do not account for overlapping activities, thus, they overestimate the participation of activities in the critical path (e.g., the processing activity in the right plot of Figure 8), resulting in activity durations that may even exceed the total duration of the snapshot. The CP-based summary of Figure 8 overcomes this problem and highlights the overhead of global coordination in micro-batch systems (driver's scheduling activity), a known result also pointed out in Drizzle [34].

SnailTrail is also different to traditional profiling in its ability to focus on different parts of a long-running

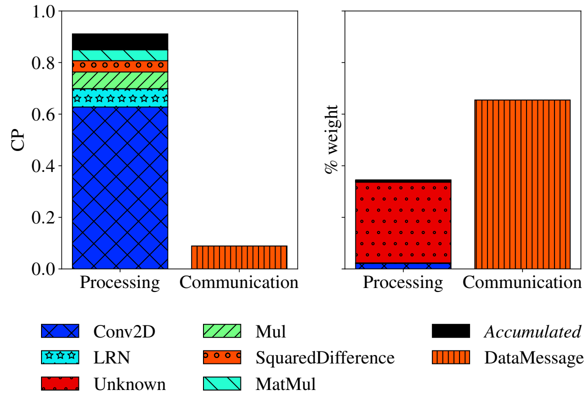


Figure 9: *CP*-based (left) and conventional profiling (right) summaries for the accuracy phase of AlexNet on TensorFlow (16 threads).

computation. This feature is particularly useful in machine learning, where program phases have diverse performance characteristics. As an example, Figure 9 shows *CP*-based and conventional summaries for the accuracy phase of the AlexNet image processing application on TensorFlow with 16 workers. We plot processing and communication as separate bars for convenience and we further break down processing into the different operators appearing in this computation phase. The conventional summary overestimates the participation of communication and underestimates the importance of the Conv2D operator, which is the most critical one according to the *CP*-based summary. Processing in the conventional summary is dominated by the unknown activity type due to limited instrumentation in TensorFlow (see [21]).

7.5 `SnailTrail` in practice

We select Apache Flink as the representative streaming system and demonstrate `SnailTrail` in action. We describe two use-cases and give examples of how the CP-based summaries can be used to understand and improve application performance of long-running computations.

Detecting skew. To demonstrate straggler summaries in action, we use the benchmark of [18]. The benchmark contains a WordCount application and a data generator. The data generator can be configured with a skewness percentage. We experiment with 30%, 50%, and 80% skewness. We configure the parallelism to be equal to 4 for all operators and we generate straggler summaries and conventional summaries shown in Figure 10. For small skew percentage, the conventional summaries fail to detect any imbalance and essentially indicate uniform load across workers. For higher skew percentages (50-80%) they indeed reveal a skew problem, yet they are unable to indicate a single worker as the straggler. Instead, they attribute the imbalance problem to several workers. On the other hand, the CP-based straggler summaries consistently and accurately detect the straggler worker, even for low skew percentage.

Optimizing operator parallelism. We now demonsrate how `SnailTrail` can guide scaling decisions for streaming applications. We use Dhalion's [18] benchmark again and initially under-provision the flatmap stage. We configure four parallel workers for the source, two parallel workers for the flatmap, and four parallel workers for the count operator. Figure 11 (left) shows the operator and conventional profiling summaries for this configuration. We see that the operator summary detects that the flatmap workers are bottlenecks. On the other hand, the conventional summary shows a negligible difference between the parallel workers' processing. In addition, we gather metrics from Flink's web interface. Using those, we can observe backpressure, yet we have no indication of the cause. We next decrease the source's input rate, by changing its parallelism to one worker. Note that slowing down the source is a common system reaction to backpressure. Figure 11 (middle) shows the operator and conventional profiling summaries after this change. Notice how slowing down the source does not solve the problem and how the operator summary still provides more accurate information than the conventional one. The operator summary essentially indicates that the flatmap operator has a high CP value and needs to be re-scaled. Figure 11 (right) shows the summaries after applying a parallelism of four to all operators. Checking Flink's web interface again we see that backpressure disappears.

8 Related Work

There exists abundant literature on performance analysis, characterization, and debugging of distributed systems, although we know of no prior work to perform online critical path analysis for long running computations, or applicable across a broad range of execution models. We distinguish three main areas of related work:

Critical Path Analysis: Yang *et al.* [36] first applied CPA to distributed and parallel applications, defined the PAG, gave a distributed algorithm for CPA, and showed its benefits over traditional profiling. CPA and related techniques have since been used to analyze distributed programs like MPI applications [32, 8] and web services [13], in all cases using offline traces. Algorithms to compute the k longest (near-critical) paths in a computation are given in [6].

The first *online* method for computing critical path profiles seems to be [22], where performance traces are piggybacked on data messages exchanged by processes at

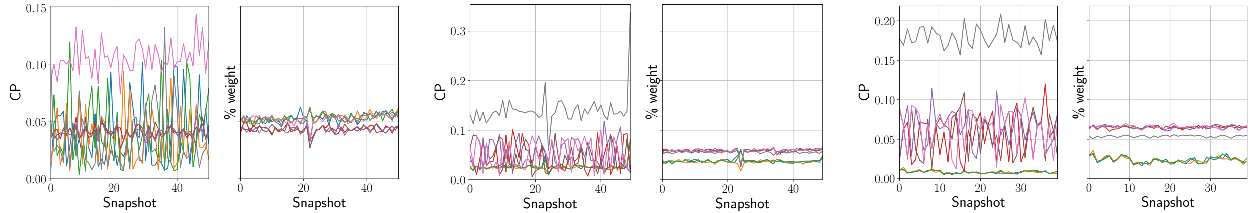


Figure 10: Straggler and conventional profiling summaries for the benchmark of [18] on Flink and different skewness percentage. The data generator has been configured with 30% (left), 50% (middle), and 80% (right) skeweness.

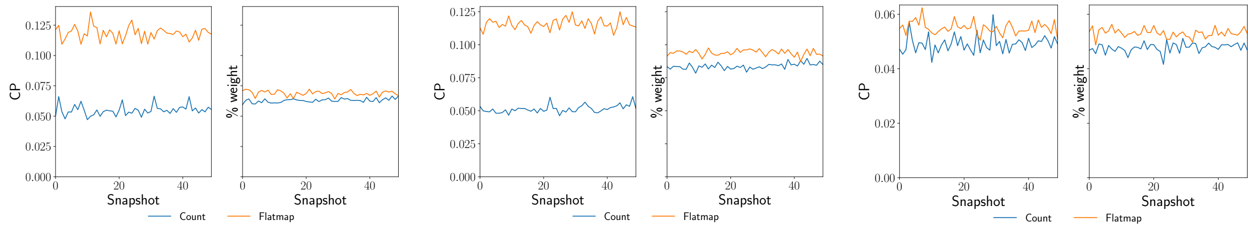


Figure 11: Operator and conventional profiling summaries for the benchmark of [18] on Flink and different configurations of operator parallelism. The source, flatmap, and count operators are configured with parallelism 4-2-4 (left), 1-2-4 (middle), and 4-4-4 (right).

runtime. However, the proposed algorithm is too expensive to construct the full PAG and is thus limited to a small number of user-selected activities. A nice feature of [22] is combining online CPA with dynamic instrumentation to selectively enable trace points on demand. [31] extends the analysis of [22] to the full software stack, and [17] uses this information for adaptive scheduling. Sonata [20] pinpoints critical activities in the spirit of CPA. It supports offline analysis of MapReduce jobs through identifying correlations between tasks, resources and job phases.

Dataflow Performance Analysis: [28] employs *blocked time* analysis to dataflow, a 'what-if' approach quantifying performance improvement assuming a resource is infinitely fast. Blocked time analysis is performed offline and assums staged batch execution. It can only identify bottlenecks due to network and disk and does not provide insights into the interdependence of parallel tasks and operators. An alternative approach in Storm [33]) is based on the *Actor Model* [7] rather than CPA. HiTune [16] and Theia [19] focus on Hadoop profiling; in particular, on cluster resource utilzation and task progress monitoring.

Distributed Systems Profiling: A comprehensive overview of prior work in distributed profiling is [39], which also introduces Stitch, a tool for profiling multi-level software stacks using traces. Like `SnailTrail`, Stitch requires no domain knowledge of the reference system, but its *Flow Reconstruction Principle* assumes logged events are sufficient to reconstruct the execution flow. `SnailTrail` in contrast does not assume this, and indeed yields insights for the better instrumentation of dataflow systems. VScope [35] targets online anomaly detection and root-cause analysis in large clusters. Finally, we note that capturing dependencies between activities in dataflows is similar to *causal profiling* in `Coz` [15]. `Coz` does not focus on distributed dataflows, but does work non-intrusively without instrumentation, and may be applicable to `SnailTrail`.

9 Conclusion

Online critical path analysis represents a new level of sophistication for performance analysis of distributed systems, and `SnailTrail` shows its applicability to a range of different engines and applications. Looking forward, `SnailTrail`'s online operation suggests uses beyond providing real-time information to system administrators: `SnailTrail`'s performance summaries could serve as immediate feedback for applications to perform automatic reconfiguration, dynamic scaling, or adaptive scheduling.

The code in `SnailTrail` has been released as open source[2].

Acknowledgments

We thank Ralf Sager for working on some intitial ideas of this paper, Frank McSherry and the anonymous NSDI reviewers for their comments, and Raluca Ada Popa for shepherding the paper. Vasiliki Kalavri is supported by an ETH Postdoctoral Fellowship.

[2] https://github.com/strymon-system/snailtrail

References

[1] BigData benchmark. https://amplab.cs.berkeley.edu/benchmark/. (accessed: September 2017).

[2] Nagios. https://www.nagios.org. (accessed: September 2017).

[3] VMware LogInsight. http://www.vmware.com/products/vrealize-log-insight.html. (accessed: September 2017).

[4] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA* (2016).

[5] Alexander, C., Reese, D., and Harden, J. C. Near-critical path analysis of program activity graphs. In *International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems* (1994).

[6] Alexander, C. A., Reese, D. S., Harden, J. C., and Brightwell, R. B. Near-critical path analysis: A tool for parallel program optimization. In *Southern Symposium on Computing* (1998).

[7] Bedini, I., Sakr, S., Theeten, B., Sala, A., and Cogan, P. Modeling performance of a parallel streaming engine: Bridging theory and costs. In *ICPE* (2013).

[8] Böhme, D., de Supinski, B. R., Geimer, M., Schulz, M., and Wolf, F. Scalable critical-path based performance analysis. In *IEEE International Parallel and Distributed Processing Symposium* (2012).

[9] Carasso, D. *Exploring Splunk*. Evolved Technologist Press, 2012.

[10] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. Apache Flink: Stream and batch processing in a single engine. *Data Engineering 38*, 4 (2015).

[11] Chen, J., and Clapp, R. M. Critical-path candidates: scalable performance modeling for MPI workloads. In *IEEE International Symposium on Performance Analysis of Systems and Software* (2015).

[12] Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B., and Poulosky, P. Benchmarking streaming computation engines: Storm, Flink and Spark Streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016* (2016), pp. 1789–1792.

[13] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 217–231.

[14] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[15] Curtsinger, C., and Berger, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 184–197.

[16] Dai, J., Huang, J., Huang, S., Huang, B., and Liu, Y. Hitune: Dataflow-based performance analysis for big data cloud. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC'11, USENIX Association, pp. 7–7.

[17] Dooley, I., and Kalé, L. V. Detecting and using critical paths at runtime in message driven parallel programs. In *IEEE International Symposium on Parallel and Distributed Processing* (2010).

[18] Floratou, A., Agrawal, A., Graham, B., Rao, S., and Ramasamy, K. Dhalion: Self-regulating stream processing in heron. *Proc. VLDB Endow. 10*, 12 (Aug. 2017), 1825–1836.

[19] Garduno, E., Kavulya, S. P., Tan, J., Gandhi, R., and Narasimhan, P. Theia: Visual signatures for problem diagnosis in large hadoop clusters. In *Proceedings of the 26th International Conference on Large Installation System Administration: Strategies, Tools, and Techniques* (Berkeley, CA, USA, 2012), lisa'12, USENIX Association, pp. 33–42.

[20] Guo, Q., Li, Y., Liu, T., Wang, K., Chen, G., Bao, X., and Tang, W. Correlation-based performance analysis for full-system mapreduce optimization. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA* (2013), pp. 753–761.

[21] Hoffmann, M., Lattuada, A., Liagouris, J., Kalavri, V., Dimitrova, D., Wicki, S., Chothia, Z., and Roscoe, T. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. Tech. rep., ETH Zurich, 2018.

[22] Hollingsworth, J. K. An online computation of critical path profiling. In *SIGMETRICS Symposium on Parallel and Distributed Tools* (1996).

[23] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[24] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

[25] McSherry, F. A modular implementation of timely dataflow in Rust (accessed: April 2017). https://github.com/frankmcsherry/timely-dataflow.

[26] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 439–455.

[27] Ousterhout, K. Spark performance analysis (accessed: April 2017). https://kayousterhout.github.io/trace-analysis/.

[28] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., and Chun, B.-G. Making sense of performance in data analytics frameworks. In *NSDI* (2015).

[29] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV) 115*, 3 (2015), 211–252.

[30] Sacerdoti, F. D., Katz, M. J., Massie, M. L., and Culler, D. E. Wide area cluster monitoring with ganglia. In *2003 IEEE International Conference on Cluster Computing (CLUSTER 2003), 1-4 December 2003, Kowloon, Hong Kong, China* (2003), p. 289.

[31] Saidi, A. G., Binkert, N. L., Reinhardt, S. K., and Mudge, T. N. Full-system critical path analysis. In *IEEE International Symposium on Performance Analysis of Systems and Software* (2008).

[32] Schulz, M. Extracting critical path graphs from MPI applications. *IEEE International Conference on Cluster Computing* (2005).

[33] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 147–156.

[34] Venkataraman, S., Panda, A., Ousterhout, K., Ghodsi, A., Franklin, M. J., Recht, B., and Stoica, I. Drizzle: Fast and Adaptable Stream Processing at Scale. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[35] Wang, C., Rayan, I. A., Eisenhauer, G., Schwan, K., Talwar, V., Wolf, M., and Huneycutt, C. Vscope: Middleware for troubleshooting time-sensitive data center applications. In *Proceedings of the 13th International Middleware Conference* (New York, NY, USA, 2012), Middleware '12, Springer-Verlag New York, Inc., pp. 121–141.

[36] Yang, C.-Q., and Miller, B. P. Critical path analysis for the execution of parallel and distributed programs. In *IEEE International Conference on Distributed Computing Systems* (1988).

[37] Yen, S. H., Du, D. H., and Ghanta, S. Efficient algorithms for extracting the k most critical paths in timing analysis. In *Proceedings of the 26th ACM/IEEE Design Automation Conference* (1989), DAC '89, pp. 649–654.

[38] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 2–2.

[39] Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., and Stumm, M. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proceedings of the 12th USENIX*

Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 603–618.

A Appendix

A.1 Model assumptions

We support both synchronous and asynchronous execution in shared-nothing and shared-memory architectures. Most dataflow systems use asynchronous computations on shared-nothing clusters, but sometimes synchronous computation is supported (e.g. in TensorFlow), and system workers can share state (e.g. in Timely). Specifically, our model is consistent with respect to critical path analysis under two assumptions:

Assumption 1 (Message-based Interaction). *Every interaction between operators in the dataflow must occur via message exchange, even if executed by the same worker.*

Note this assumption does *not* preclude shared-memory systems. Operators in the reference dataflow may share state as long as any modification to this state is appropriately instrumented to trigger a 'virtual' message exchange between the workers sharing that state. We use this approach in instrumenting shared state in Timely Dataflow, for example.

Assumption 2 (Waiting State Termination). *Every waiting activity in a worker's timeline is terminated by an incoming message, either from the same or a different worker.*

In other words, a worker in a waiting state cannot start performing activities unprompted without receiving a message. In the activity graph, a waiting edge's end node must correspond to that of a communication activity, i.e., a receive.

A.2 Instrumentation requirements

An activity may consist of sub-operations spanning multiple levels of the stack from user code to OS and network protocols. A given system can be instrumented at different levels of granularity, depending on the use-case: a multi-layered activity tracking approach enables more detailed performance analysis but introduces higher overhead. We allow this choice, but require that any instrumentation of the reference system satisfy two properties, without which the transient critical paths are ill-defined. The first states that any event having prior events must be caused by an activity earlier in time, i.e. any "out-of-the-blue" events in $(t_s, t_e]$ indicate insufficient instrumentation:

Property 1 (Minimum in-degree) *Let $G_{[t_s,t_e]} = (V, E)$ be the snapshot of activity graph G in time interval $[t_s, t_e]$. Let also $V_s \equiv \{v \in V \mid \nexists v' \in V : v'[t] < v[t]\}$ be a set of vertices in $G_{[t_s,t_e]}$. A vertex $v \in V \setminus V_s$ has in-degree at least one.*

The second states that at no point do all system workers perform waiting activities while no communication activity is occurring. Such behavior would imply deadlock, and so any such points in the activity graph of a non-blocked computation indicates insufficient instrumentation:

Property 2 (Communication Existence) *Let $G_{[t_s,t_e]} = (V, E)$ be the snapshot of an activity graph G in $[t_s, t_e]$, and $\tau \in [t_s, t_e]$ be a point in time. Let $S \equiv \{e = (v_i, v_j) \in E_w \subseteq E \mid e[p] = Waiting,\ v_i[t] \le \tau \le v_j[t]\}$. If $|S| = N_\tau$, where N_τ is the number of active workers of the reference system at time τ, then $\exists e' = (v_k, v_m) \in E_c \subseteq E$ for which $v_k[t] \le \tau \le v_m[t]$.*

These two properties can also checked efficiently online to inform users when the ingested activity logs are incomplete. For example, instrumentation (or associated log preprocessing) can guarantee that no waiting activities are created as long as the corresponding communication activity, which caused the waiting activity to end, has not been observed.

A.3 Proofs for Equations of Section 3.2

First, we provide a proof for Eq. 3:

$$CP_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} = \frac{c(e) \cdot e[w]}{N(t_e - t_s)} \in [0,1]$$

Recall that e is an activity edge in the PAG snapshot, N is the total number of transient critical paths in the snapshot, q_e^i is ratio of the activity's duration to the total duration of the i-th transient critical path (the ratio is 0 if the activity edge is not part of the i-th path), $0 \le c(e) \le N$ is the number of transient critical paths the activity e belongs to, $e[w]$ is the weight of the activity e, i.e., its duration, and $[t_s, t_e]$ is the snapshot window size.

Without loss of generality, we assume that the transient critical paths $\vec{p_i}$ the activity edge e belongs to are numbered from $i = 1$ to $i = c(e)$. Then:

$$CP_e = \frac{\sum_{i=1}^{i=N} q_e^i}{N} = \frac{\sum_{i=1}^{i=N} \frac{e[w]}{\|\vec{p_i}\|}}{N} = \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\vec{p_i}\|}}{N} + 0 = \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\vec{p_i}\|}}{N}$$

All transient critical paths in the snapshot have the same length $\|\vec{p_i}\|$ (in time units), which is equal to the duration of the snapshot $t_e - t_s$. Hence:

$$CP_e = \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{t_e - t_s}}{N} = \frac{\sum_{i=1}^{i=c(e)} e[w]}{N \cdot (t_e - t_s)} = \frac{c(e) \cdot e[w]}{N \cdot (t_e - t_s)}$$

Now we provide the proof for Eq. 5:

$$\sum_{\forall c \in G} \sum_{\forall e : e[p] = c} CP_e = 1$$

Recall that c denotes an activity type, e.g., serialization, and $e[p]$ is the type of the activity edge e in the snapshot $G_{[t_s, t_e]}$. We have:

$$\sum_{\forall c \in G} \sum_{\forall e : e[p] = c} CP_e = \sum_{\forall e \in G} CP_e = \sum_{\forall e \in G} \frac{\sum_{i=1}^{i=N} q_e^i}{N} =$$

$$= \sum_{\forall e \in G} \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\vec{p_i}\|} + 0}{N} = \sum_{\forall e \in G} \frac{\sum_{i=1}^{i=c(e)} \frac{e[w]}{\|\vec{p_i}\|}}{N} =$$

$$= \frac{\sum_{\forall e \in G} \sum_{i=1}^{i=c(e)} \frac{e[w]}{t_e - t_s}}{N} = \frac{\sum_{\forall e \in G} \sum_{i=1}^{i=c(e)} e[w]}{N \cdot (t_e - t_s)} = \frac{N \cdot (t_e - t_s)}{N \cdot (t_e - t_s)} = 1$$

since $\sum_{\forall e \in G} \sum_{i=1}^{i=c(e)} e[w]$ denotes the sum of the weights (durations) of all activity edges that comprise all N transient critical paths in the snapshot, which is equal to $N \cdot (t_e - t_s)$.

A.4 Clock alignment

Computing critical paths only needs logical time, i.e. the happens-before relationship between events. In practice we are using wall-clock time as a stand-in for Lamport timestamps [24] to establish partial ordering of events. Performance statistics such as summaries, however, do require real time.

A practical system for critical path analysis must therefore address issues of *clock drift* (where clocks on different nodes run at different rates) and *clock skew* (where two clocks differ in their values at a particular time).

Clock *drift* only affects activities running on the same thread with durations greater than the drift. Even a drift of 10 seconds/day translates to 0.1ms inaccuracy for activities taking around a second, which is probably tolerable. Clock *skew* is not an issue for activities timestamped by the same thread, but might be for communication activities.

In `SnailTrail`, we assume that the trend toward strong clock synchronization in datacenters [14] means that clock skew is not, in practice, a significant problem for our analysis. If it were to become an issue, we would have to consider adding Lamport clocks and other mechanisms for detecting and correcting for clock skew.

Balancing on the edge: transport affinity without network state

João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek and Raul Landa
Fastly

Abstract

Content delivery networks and edge peering facilities have unique operating constraints which require novel approaches to load balancing. Contrary to traditional, centralized datacenter networks, physical space is heavily constrained. This limitation drives both the need for greater efficiency, maximizing the ability to absorb denial of service attacks and flash crowds at the edge, and seamless failover, minimizing the impact of maintenance on service availability.

This paper introduces *Faild*, a distributed load balancer which runs on commodity hardware and achieves graceful failover without relying on network state, providing a cost-effective and scalable alternative to existing proposals. Faild allows any individual component of the edge network to be removed from service without breaking existing connections, a property which has proved instrumental in sustaining the growth of a large global edge network over the past four years. As a consequence of this operational experience, we further document unexpected protocol interactions stemming from misconfigured devices in the wild which have significant ramifications for transport protocol design.

1 Introduction

While economies of scale have increasingly centralized compute and storage, user expectations dictate that content and even logic be pushed further towards the edge of the network. This centrifugal relationship has increased the relevance of Points of Presence (POPs) as an intermediary between cloud hosted services and end-users.

As a design pattern, POPs were pioneered by Content Delivery Networks (CDNs) intent on improving performance for traditional HTTP caching and DNS resolution. They have since evolved to encompass a much wider set of application layer services such as media processing (*e.g.* video and image optimization), security (*e.g.* application layer firewalls) and business logic (*e.g.* authentication and authorization; paywalls; request routing). The common thread uniting these *edge cloud* services is that they are latency sensitive and must therefore be geographically distributed across POPs rather than merely centralized within availability regions.

Today, edge cloud providers deploy hundreds of POPs [31, 39], with individual POPs able to deliver upwards of a terabit per second of bandwidth whilst handling millions of requests per second. How traffic is distributed across available hosts within a POP has a significant impact on the performance and availability of a large number of Internet services. Load balancing in this context differs significantly from traditional datacenter environments, and as a result existing solutions [15, 16, 17, 22, 27] are not readily applicable.

The defining constraint in the architecture of a POP is that physical space is at a premium [39]. POPs are typically set up in colocation facilities, often in regions where few alternatives exist, if any. The resulting inelastic price dynamics impose a strong economic incentive to minimize POP hardware in an effort to reduce capital expenditure. Load balancing under such constraints exacerbates the following concerns:

Efficiency. The physical build of POPs must be designed to maximize the number of service requests that a given number of hosts can process. Proposals that rely on dedicated hardware appliances or VM instances [15, 16, 17, 27] are not cost-efficient, since they consume scarce resources without increasing the number of requests that a POP can service.

Resilience. POPs provide a critical service at a fixed capacity, and are therefore attractive targets for denial-of-service attacks. Load balancing proposals which rely on flow state or incur significant per-flow overhead [15, 16, 17, 22, 27] are vulnerable to exhaustion attacks, and pose a threat to business continuity.

Gracefulness. Owing to the higher processing density of POPs, individual components within a POP represent a much larger proportion of total system capacity

when compared with traditional cloud environments. It is therefore vital that all system components can be brought in and out of service gracefully, without disrupting active flows. To our knowledge, no existing load balancer ensures seamless addition and removal of every element of its architecture, including hosts, switches and peers.

The primary contribution of this paper is to describe the design and implementation of a load-balancer which achieves all of the above goals. *Faild* provides *transport affinity* (seamless transport-layer failover) with minimal processing overhead. It synthesizes two distinct approaches leveraging hardware processing on commodity switches where possible, and pushing out flow handling to efficient software implementations when necessary.

A load balancing solution operating further down the network stack would be oblivious to how packets relate to ongoing transport connections, and therefore unable to ensure *graceful draining*, *i.e.* bringing a system component out of production without affecting service traffic. On the other hand, operating above the transport layer requires per-session state tracking, which is at odds with our requirement for robustness against resource exhaustion attacks. A key insight in this paper is that since end hosts track flows themselves, this trade-off is not binding: by performing host mapping on switches and re-purposing the connection tracking functions that end-hosts already perform, a load-balancer can retain transport affinity without the burden of maintaining network state. Although many of the specific techniques used by Faild have been used in isolation in other contexts (see Sec. 7), Faild brings them together in a novel way that directly addresses the needs of edge clouds and CDNs.

Our final contribution is to provide insight into the trials and tribulations of operating Faild over the past four years at a large[1] edge cloud provider. We document where reality fell short of our original design assumptions - from network equipment shortcomings to unexpected middlebox behavior - and how these quirks may affect future protocols to come.

The remainder of this paper is organized as follows. Sec. 2 expands upon the engineering requirements of edge load balancing. Sec. 3 describes the design of Faild, and Sec. 4 details its current implementation. Sec. 5 evaluates the system in practice, and Sec. 6 describes some of the lessons we learned by developing it and deploying it on production. We present relevant related work in Sec. 7, and conclude with Sec. 8.

2 Background and motivation

While the architecture of a POP bears superficial resemblance to traditional datacenter environments, their goals differ, and as a result the set of constraints they impose diverge. This section expands upon the requirements presented in Sec. 1 and highlights the idiosyncrasies of load balancing within POPs.

High request processing density. Ideally, all available power and space in the POP would be devoted exclusively to *hosts*. This proves unfeasible except for the very smallest POPs. As the number of hosts in the POP increases, directly connecting them to upstream providers becomes 1) logistically impractical for providers; 2) prohibitively expensive for intra-POP traffic; and eventually 3) physically impossible due to lack of ports. Network devices are therefore required to both reduce the number of interconnects required towards providers and ensure connectivity between hosts[2]. Our preferred *POP topology* maximizes power and space cost-benefit for hosts by collapsing all load balancing functions into just two layers: one for switches and one for hosts. A POP hence consists of a minimal number of switches, each one connected to all hosts and providing them with consolidated BGP sessions and Internet connectivity. This deviates from a common design pattern in datacenters, which relies on Clos-like topologies [6, 33].

Traditional hardware solutions *e.g.* [1, 3] are undesirable both due to their low power/space efficiency and poor horizontal scalability. Software-based solutions [15, 27] on the other hand perform poorly given the lower throughput and higher latency of packet processing on general purpose hardware. For example, Maglev [15] claims a 10 Gbps limit per load balancing host. Our most common POP build has 4 switches fully meshed with 32 hosts using 25 Gbps NICs. Since each host has a rated capacity of 40 Gbps in order to ensure operational stability, this results in a reference POP throughput of 1.28 Tbps. This alone would require over 100 Maglev hosts to load balance, which greatly exceeds the number of target hosts. Even if future advancements made it possible to substantially reduce the number of software load balancers required per target host, deploying load balancing functions in hosts would prevent using the full bisection bandwidth offered by the physical POP topology, further impacting request processing density.

Traffic surges. POP load balancers must be designed to gracefully withstand traffic surges of hundreds of times their usual loads, as well as DDoS attacks.

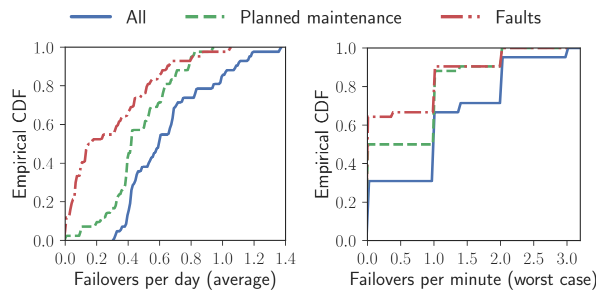
Highly optimized stateful solutions such as SilkRoad [22] can scale to ten million simultaneous connections by using hashing to scale per-connection state management and storing this state in the ASIC SRAM memory of switches with a programmable data plane. In our experience however POPs are routinely subjected to SYN floods largely exceeding this number, and the intensity and frequency of these attacks is suppressed only

[1] 40+ POPs globally; 7+ million RPS and 4+ Tbps of client traffic.

[2] Facebook EdgeFabric [31] and Google Espresso [39] rely on similar architectures to provide a capacity-aware, SDN-based *edge peering*.

when they fail to inflict economic harm. While software-based solutions like Maglev [15] and Duet [17] are less memory-constrained than SilkRoad, they are still subject to degraded performance as the connection count rises.

A commonly deployed alternative to stateful load balancing is to rely on Equal Cost Multipath (ECMP) [18, 38] to simply hash inbound connections to active hosts. While ECMP results in connection resets when re-hashing, many operators tolerate this given they more often operate under high load than high churn. Flashcrowds and DDoS in particular make statelessness an engineering necessity for edge load balancing, rather than a design choice.



(a) Average host drain event rate per POP per day

(b) Worst case host drain event rate per POP per minute

Figure 1: Drain events for one month, for all POPs

Host churn. While horizontally scaled services in a datacenter may comprise tens or hundreds of thousands of instances, POPs will typically have tens of nodes. As a result, the relative service impact of removing and adding hosts is much greater. Similarly, POPs are directly exposed to a much more diverse set of peers (via IXPs, Internet exchange points).

Without support for graceful failover, churn across the set of hosts and providers can disrupt traffic in a manner which is observable by customers, consequently increasing the operational expense for support and deployment. The need to remove hosts for software upgrades in particular is further exacerbated in edge networks due to their pivotal role in securing cloud services. Often edge networks will provide TLS termination and interface with a much wider set of clients, which increases the churn due to important software upgrades. Fig. 1 shows the average and worst case rate for hosts being drained across POPs over the course of a month. The average daily drain rate is not neglible given the size of a POP - this will include both planned, automated maintenances and software or hardware faults. Multiple concurrent drain events, however are relatively rare, since automated upgrades will be halted in the presence of unplanned events. In our environment switches are almost as frequently upgraded as hosts, given we run a custom control plane stack and there are frequent adjustments to the set of BGP peers.

Software load balancers [15, 16, 17, 27] can gracefully drain service hosts, but not the load balancers themselves since they maintain per-flow state. Hence, it is normally not possible to remove a load balancer instance from service without disrupting ongoing connections unless state is synchronized across all instances, which in itself is non-trivial and hence rarely supported. Furthermore, routing changes within a POP or peer churn may cause flows to be hashed to different load balancer instances, thereby disrupting existing flows.

3 Design

Building on the stated goals from previous sections, we now turn our attention to designing an efficient, stateless and graceful load balancing system. *Faild* attains these goals by relying on the socket information that hosts are required to maintain, allowing network devices to remain oblivious to TCP connection state. Similarly to other load balancing architectures [15, 17, 19, 27], Faild uses ECMP to hash flow tuples and load balance *service* traffic. Correspondingly, services are application layer abstractions advertised to the Internet using sets of *virtual IP addresses (VIP sets)*. Sec. 3.1 explores how Faild couples ECMP and MAC address rewriting to approximate *consistent hashing*. We then describe how this can be leveraged to enable graceful host failover in Sec. 3.2 and discuss host-side packet processing in Sec. 3.3.

3.1 Consistent hashing

We define *consistent hashing* as the ability to change from load balancing over a *baseline* host set B to balancing over a *failover* host set F with only traffic destined to hosts in a *removal* host set R of hosts in B but not in F being affected. Only flows terminating on R (the set of hosts being *drained*) are affected; ongoing flows terminating on hosts in the *common* set C of hosts found in both B and F remain unaffected.

In Faild, switches implement platform agnostic consistent hashing by maintaining a *fixed* set of virtual nexthops, forcing the switch to perform an ARP lookup instead. By not manipulating the routing table, we avoid rehashing events which would otherwise reset existing connections. Faild maps each service (set of VIPs) to a set of ECMP nexthops, and each ECMP nexthop to a MAC address. It is this transitive association between services and MAC addresses that determines the load balancing configuration.

Draining a switch can be achieved by instructing it to withdraw route advertisements for all services from all BGP sessions with its upstream providers. This will redirect traffic flowing through it towards neighboring switches, which would still advertise the withdrawn prefixes. The only requirement for this process to be grace-

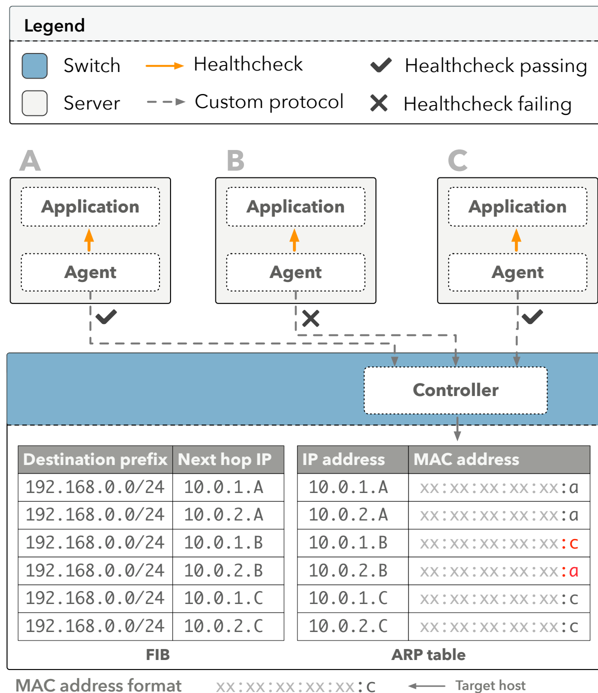


Figure 2: Custom routing protocol based on ARP table manipulation. The routing table remains static while the ARP table is adjusted to point at healthy hosts. Additional nexthops ensure even distribution of traffic when some hosts are unavailable.

ful is that all switches are configured to hash identically, so that all switches allocate flows to the same hosts.

A constraint of implementing consistent hashing in this manner is that the granularity with which traffic can be re-balanced is now directly tied to the number of nexthops used. If we only allocate one nexthop per host, removing a host from service would potentially double the amount of traffic towards a healthy neighbor. To avoid this, we can use more nexthops to provide finer-grained control, as illustrated in Fig. 2. In this example having two virtual nexthops per host is enough ensure that healthy hosts in the POP have an equal number of ARP entries directed at them when withdrawing B from production. In our implementation, we greedily reassign ARP entries to the hosts with the fewest mapped entries. Faild can approximate uniform load balancing by allocating a large number of ECMP nexthops to a smaller number of hosts. In practice we are constrained by the maximum number of ECMP nexthops supported by the underlying hardware (see Sec. 6.2).

Although switch vendors have recently started to provide consistent hashing natively [2, 4], Faild provides its own implementation for two important benefits. Firstly, by not relying on vendor implementations we lower the entry-level cost of eligible switches for use in our POP designs. More importantly, controlling the consistent hashing logic allows us to signal to target hosts whether the connections hashed onto them may have previously been hashed onto different hosts. This is crucial to use host TCP connection state to achieve transport affinity, as we show in Sec. 3.2.

In the example in Fig. 2, if the ECMP nexthops associated with B are re-allocated to other hosts in the pool, all ongoing connections towards host B will be terminated. Within large datacenters this problem is usually dealt with by using application-aware load balancers that track flow state and map new connections to healthy hosts, maintaining this mapping until completion. This approach runs counter to our goal of maximizing overall system capacity. In order to retain efficiency, we must push the equivalent functionality down towards the hosts.

3.2 Encoding failover decisions

Host draining cannot be implemented on switches alone, since they are layer 2/3 entities with no visibility of what flows are in progress towards each host at any given time. Instead, Faild distributes the responsibility for host draining across both the switch controller and hosts.

On the switch, a controller is responsible for using consistent hashing to steer traffic towards hosts in the failover set F. Hosts in F redirect traffic for ongoing connections back to their original destination in R, so that they continue to be served until their natural conclusion. Rather than relying on proprietary mechanisms, Faild implements detour routing when draining by extending the semantics of MAC addresses to encode load balancing state and its associated routing decisions, in addition to network interface identification.

Any host f in F can forward ongoing connections to their original host r in R if they ascertain the identity of this host. Conceptually, this can be done by annotating all drained traffic sent to f with the host r responsible for handling connections which were active before the drain episode started.

On baseline conditions all traffic for host r will be sent to MAC addresses with `r:r` suffix, and all hosts will receive the load assigned to them by the baseline distribution. Hence, in the baseline state the last two octets for all MAC addresses installed in the switch ARP table will be identical, signaling that all flows should be processed by their baseline host. In practice, switches forward frames to hosts based on the last identifier only.

Now consider host r being drained. Fig. 3 revisits the ARP table for the example in Fig. 2, this time using MAC addresses which reflect a host being drained from service, rather than simply removed. Faild will update all MAC addresses in the switch forwarding table that have suffixes of the form `r:X` by re-writing the penultimate octet to denote the *failover host* f. Hence, ECMP

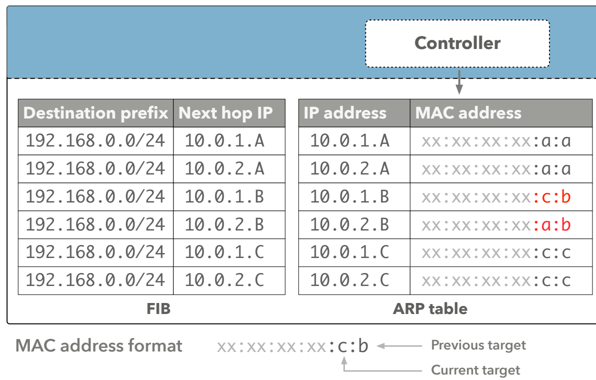


Figure 3: ARP table during draining of host B. The destination MAC address now encodes the drained host alongside the failover host.

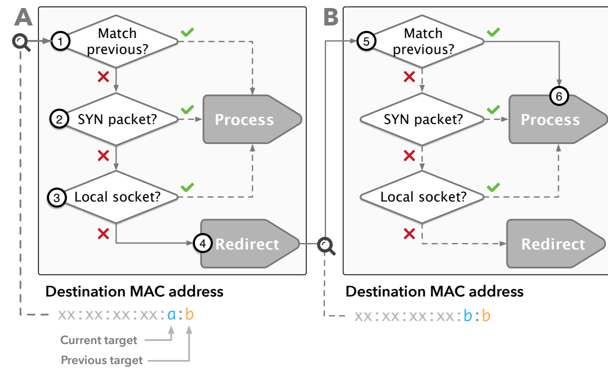


Figure 4: Receive-side packet processing example for traffic draining from host B towards host A. Packets filtered through host A are only accepted if they belong to a new connection, or if they match a local socket.

nexthops previously mapped to MAC addresses with an `r:r` suffix will now have an `f:r` suffix and traffic will be failed over to the correct host f in F.

By rewriting the ARP table in the switch, Faild can specify, for each IP nexthop mapped to a host r in R, the failover host f in F that will process new incoming requests. This improves upon alternative techniques such as switch-native consistent hashing: by appropriately selecting MAC addresses, Faild can inform a given host b which flows correspond to its baseline ECMP nexthops (if $b \in C$ for a given nexthop, the MAC address used will have a `b:b` suffix) and which ones correspond to failed-over ECMP nexthops (if $r \in R$ and $f \in F$ for a given nexthop, the MAC address used will have a `f:r` suffix). Such an explicit signaling is not achievable with switch-based implementations of consistent hashing.

3.3 Host-side processing

The method by which Faild steers traffic means we can no longer rely on routing protocols such as BGP and OSPF. Faild removes this responsibility from the routing layer, instead pushing it down to the data link layer. This is done using a controller that exchanges information with agents running on hosts and manipulates the ARP table on the switch. A host must therefore run an agent which is responsible for health checking local services, and is able to drain hosts if their associated service becomes degraded, non-responsive or placed under maintenance. Since this agent provides intelligent control over MAC-to-IP bindings, Faild switches do not use either ARP or IPv6 ND protocols.

Having received failover state information down from the switch, hosts can decide whether to process traffic locally or whether to forward it to a drained host. This not only removes the need for maintaining flow state within the network, but also distributes the computational cost of load balancing across a set of nodes which are more numerous and possess greater processing capabilities than switches. This computational cost is further reduced by implementing all of the receive-side processing as a single purpose kernel module, which efficiently processes inbound packets according to the destination MAC address (see Fig. 4).

When processing a frame, the kernel module receive handler at host h first determines whether the MAC address used is of the form `h:h` where h matches its own identifier. If so, packet processing is handed over to the local network stack, since we are operating as a baseline host ($h \in B$). Otherwise, the MAC address in the frame will be of the form `h:r`, where $h \in F$ and $r \in R$ (h is the failover host for r for this ECMP nexthop). The module must then verify whether the packet inside the frame belongs to a new connection, as signaled by the SYN flag in the TCP header (step 2), or to an existing connection, which can be verified by performing a lookup against the local socket table (step 3). If none of these conditions are met, the packet is redirected to the drain target r of the request by rewriting its destination MAC address to have a `r:r` suffix and returning the packet to a Faild switch. The same processing logic is applied at r (step 5). In this case, a `r:r` MAC address suffix indicates a locally maintained connection, and the frame will be immediately accepted. Outbound packets always follow the direct path towards their destination, resulting in asymmetrical packet forwarding during draining.

4 Implementation

This section details implementation specific nuances of Faild on both the switch and host.

4.1 Switch controller

The switch controller is implemented in approximately 3500 LOC of Python code and runs as a userspace dae-

mon on the control plane of a commodity switch built on merchant silicon. While we use a proprietary vendor API to access and configure data plane lookup tables, most of the functionality could be implemented in a portable way using OpenFlow [21], P4 [9] or SAI [5]. The controller operates on three tables:

- the **routing table**, by configuring VIP prefixes for each of the services to be load balanced and mapping them to the corresponding group of ECMP next-hops;

- the **ARP table**, by mapping each nexthop to the appropriate virtual MAC address as hosts are added or removed from the service pool; and

- the **bridging table**, by mapping each virtual MAC address to an outbound interface. Given the MAC addresses are virtual, there is no opportunity for learning what egress ports they map to. Instead, the controller must statically construct this mapping in order to avoid flooding traffic over all ports. This mapping can be provided via configuration, or autoconfigured using a custom protocol (*e.g.* LLDP discovery).

The switch controller also has the responsibility of keeping track of agent health checks, as well as ensuring that load is evenly distributed across available hosts. Despite its statelessness, a controller can use its logically centralized view of network health to implement a variety of load balancing strategies. Currently, agents encode three possible states for a health checked service: *up*, *down*, *disabled*. Operationally, there is an important distinction to be made between the *disabled* and *down* service states, since the former denotes an intentional withdrawal from service. While the underlying mechanics used to drain the host are the same in either case, a rapid sequence of downed hosts may be a symptom of a cascading failure, at which point a controller may decide to lock itself and fall back to standard ECMP.

Routing a packet of a load balanced connection requires up to two sequential lookups. The first is the routing table lookup, normally performed in TCAM, to resolve the destination IP address to a group of ECMP nexthops. The second involves the computation of a hash on the packet five tuples and the resolution to an entry of the nexthop group, normally stored on on-chip SRAM.

ECMP lookup table size does not adversely affect available space in the TCAM given that exact match lookups are executed on on-chip SRAM [10]. Hence, whereas supporting more services or fragmenting the VIP prefixes used will lead to an increase in TCAM footprint, adding more hosts or assigning more MAC addresses to each host will only require an increase in SRAM footprint. Faild leverages the lower cost and greater abundance of SRAM over TCAM to achieve improved horizontal scalability and decouple it from memory limitations.

4.2 Host agent

The host runs both a kernel module (1250 LOC in C) and a userspace daemon (2000 LOC in Python). The userspace daemon is responsible for configuring VIPs locally, executing healthchecks and relaying service health upstream to switch controllers. The kernel module is responsible for processing incoming packets with a Faild virtual MAC address as a destination. Depending on the address and the local socket table, the module will either deliver the packet locally or redirect the packet towards the alternate host encoded in the destination MAC.

In addition to the processing described in Fig. 4, the kernel module must add each of the secondary MAC addresses to the NIC unicast address filter using a standard kernel network driver API function. The NIC driver implements this function by adding the MAC address to the NIC's unicast perfect match filter. If this filter table is full, the NIC will resort to either using a hash-based filter, or by enabling unicast promiscuous mode, depending on the particular NIC model in use.

A further implementation nuance of the kernel module is in ensuring correct SYN cookie support. SYN cookies [8, 14] are vital in defending against large scale SYN floods. The difficulty when we are sending SYN cookies is that returning ACK packets will not match a local socket. As a result, we cannot determine whether they belong to a 3-way handshake for a connection that we ourselves sent a SYN cookie for and hence should be delivered locally, or whether they are regular ACKs for a drained host connection and hence should be forwarded.

Fortunately, there is a solution for this that is both simple and elegant. The Linux kernel enables SYN cookies automatically upon listen queue overflow. We check whether the listening socket on the local host has recently seen a listen queue overflow, and if so, we execute a SYN cookie validation on the ACK field. If the validation succeeds we deliver the packet locally and we forward the packet otherwise. Since a SYN cookie is a hash of a set of connection-related fields as well as some secret data and is designed to be nontrivial to forge, we are unlikely to accept an arbitrary ACK value as a valid SYN cookie. We are also unlikely to accept a SYN cookie generated by another host in the POP as a valid SYN cookie since each host uses a distinct secret hash seed.

5 Evaluation

In this section we evaluate the efficiency, resilience and gracefulness of Faild (see Sec. 1). In particular, we show that Faild can drain any system component without impacting end-to-end traffic (Sec. 5.1); can drain hosts in a timely manner (Sec. 5.2); does not induce a significant latency increase when detouring drained connections (Sec. 5.3); does not impose significant CPU over-

head on hosts (Sec. 5.4); and can achieve good load distributions across available hosts (Sec. 5.5).

All results presented were collected by means of active and passive measurements on our smallest production POP deployments, composed of only two switches and eight hosts. Despite occupying only half a rack, this particular instantiation of our POP design can scale up to 400 Gbps of throughput and handle up to 320k requests per second (RPS). Larger POPs can span up to four switches and 64 hosts.

5.1 Graceful failover

Sec. 3.1 claimed Faild maintains transport affinity when draining switches, allowing switches to be removed from operation without impacting established TCP connections. This is demonstrated in Fig. 5, that shows off-peak draining and refill of a Faild switch in a production POP. This POP only comprises two switches, both multihomed to the same set of providers. The drain and refill events are visible in the graph at the top, in which all inbound traffic from one switch is initially shifted to the other and then shifted back. The other graphs highlight that both the number of requests served by POP hosts and the rate of reset packets (RST) and retransmissions they generate remain unchanged. The graceful removal and addition of one switch confirms that both switches must be applying the same hashing function.

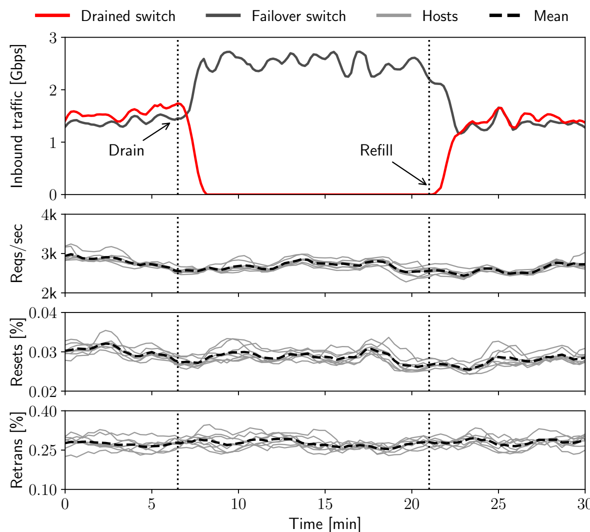


Figure 5: Graceful draining and refill of a switch

We proceed to demonstrate host draining. With all other components in steady state, the withdrawal of a single host from service should trigger Faild to redistribute traffic equally across all seven neighbors. This is observable in the top graph of Fig. 6, which shows the rate of requests handled by each host of a POP as a host is drained and refilled. We can make two important observations. First, upon disabling a host, the rate of VIP-related requests decreases rapidly and eventually converges to zero, on a timespan depending on the distribution of flow sizes active on the host when drained. Similarly, upon the host being re-enabled, the rate of requests rapidly converge to pre-draining values. Second, enabling and disabling a host does not cause any increase in RST and retransmission rates on any host. This validates that both events did not cause packets being delivered to incorrect hosts or dropped.

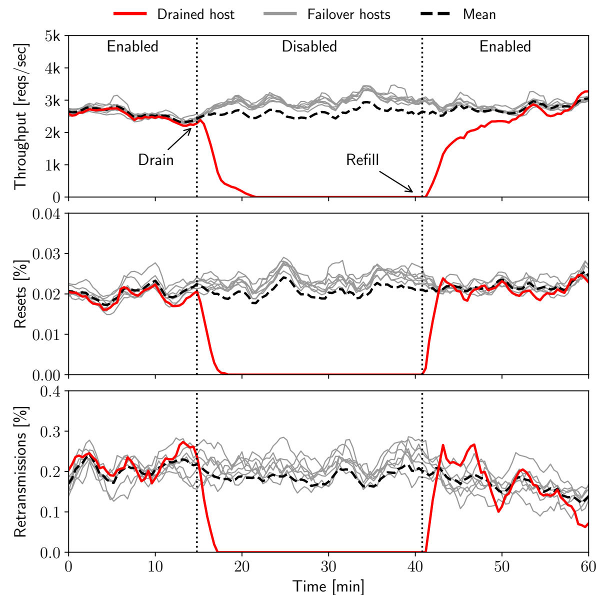
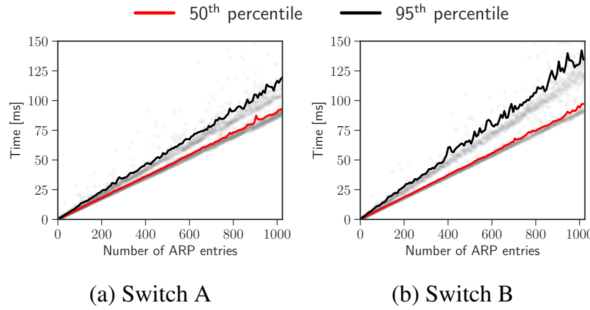


Figure 6: Host failover. Host d, in red, is drained and then refilled; traffic is shifted towards failover hosts $f \in F$, in grey, and then shifted back to d.

5.2 Switch reconfiguration time

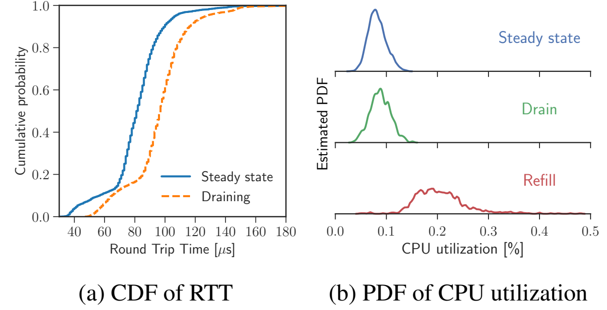
As discussed in Sec. 3.1, Faild implements host draining by means of a user space application running on the switch that updates the ARP table. For benchmarking purposes, we repurposed this application to perform batch ARP table updates of varying sizes. This was carried out multiple times on two models of production switches, referred to as switch A and B, equipped with different ASICs. Fig. 7 plots the measured time as a function of the number of ARP entries to be updated.

Roughly, we observe that overall reconfiguration time scales linearly with the number of updates. Even in the worst case scenario, requiring 1024 ARP entries to be updated, the 95th percentile is 119ms for switch A and 134ms for switch B. These values are low enough to ensure that, for the foreseeable future, Faild does not hinder our ability to react to host liveness in a timely manner. This is particularly true given ARP updates are atomic, and therefore service traffic is not disrupted during reconfiguration.



(a) Switch A (b) Switch B

Figure 7: ARP table reconfiguration time



(a) CDF of RTT (b) PDF of CPU utilization

Figure 8: Draining microbenchmarks

5.3 Detour-induced latency

Unfortunately, common debugging tools (*e.g. ping*, *traceroute* and *nc*) are not useful to measure the latency induced by detour routing during draining. This is because TCP SYN packets are never detour routed, and ICMP echo request packets do not trigger a lookup in the host socket table, which is crucial to obtain a realistic approximation of system behavior.

To measure detour routing latency we built a simple active measurement tool that runs on a non-Faild server q. The tool generates a stream of non-SYN TCP packets having a destination MAC address suffix `f:r` that specifies a node r as the drained host and f as its failover host. Since these are not TCP SYN packets, they trigger a lookup in the socket table when received by f. Since these packets do not correspond to an ongoing TCP connection, the lookup will not return a match, resulting in a detour through host r. When r attempts to process the packet it will not find a matching socket either, and will subsequently generate an RST packet and send it to q. By measuring the time between the generation of the non-SYN and RST packets q can measure the round trip latency. The direct round trip latency between q and f is measured using the same tool, but in this case the destination MAC address is set with two identical last octets, which causes f to reply directly with a RST packet.

The subsequent results are plotted in Fig. 8a, that shows the empirical CDF of RTT in steady-state and draining phases. Introducing a single hop detour induces a very low increase in end-to-end latency: the 50th percentile of the latency differential is 14 μs, the 95th percentile is 14.6 μs, and the 99th percentile is 19.52 μs. It should be noted that this small additional latency is observed only during a host draining phase and only by flows terminated at hosts being drained. In contrast, software load balancers add a latency between 50μs and 1ms [17, 15] to all packets they process.

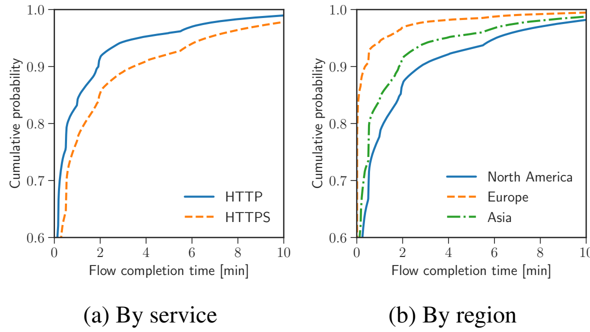
5.4 Host overhead

In order to measure the overhead incurred by the kernel component of Faild, we instrumented hosts to periodically collect stack traces on each CPU and count how many of those traces included function calls attributed to Faild. These counts are then normalized using the total number of traces collected. Since hosts run on GNU/Linux, we collected traces at 999 Hz to avoid synchronization with the 1 kHz timer interrupt frequency.

Fig. 8b shows the estimated probability density for the CPU utilization of the Faild kernel module calculated using Gaussian kernel density estimation [32]. We model the distribution for each one of three phases: steady-state, draining and refill. Data referring to host draining and host refill was collected over the first two minutes after the transition was triggered.

In steady-state we can observe that CPU utilization is very low, averaging approximately 0.1%. The utilization remains approximately constant as draining begins, and during the drain event itself. A noticeable increase occurs when we revert the draining operation and shift traffic back to the drained host, which we denote as *refilling*. At this point the kernel module receives a large number of packets in which the last two octets of the destination MAC address differ, which requires a lookup in the socket table to be able to trigger detour forwarding in case of a miss. In spite of the increased workload, the average CPU utilization remains remarkably low, peaking at 0.5% for a very small subset of samples. In general, the expected utilization during refill is 0.22%.

Our two minute measurement cut-off is justified because flows usually terminate quickly after a drain event starts. To verify this, we measured the distribution of flow completion times from different vantage points on our network. The resulting distributions for three of our POPs in distinct geographic regions are plotted in Fig. 9. While these results are biased towards our customer base and the configuration settings for their applications, they provide us with a feel for the underlying properties of flows in flight. Our analysis shows that most flows are short: between 60% and 70% of flows last less than 10 seconds and between 78% and 85% last less than 1 minute. Host overhead is therefore not only small to begin with, but also decays rapidly over time.

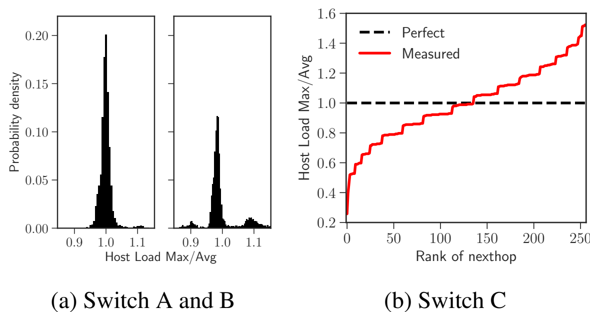


(a) By service (b) By region

Figure 9: Distribution of flow completion time

5.5 Load balancing accuracy

In order to achieve good load distributions, 1) switches must provide a good enough hardware implementation of ECMP hashing; and 2) the mix of inbound traffic must be such that ECMP will provide sufficiently homogeneous traffic balancing. We validate these two requirements.



(a) Switch A and B (b) Switch C

Figure 10: Granularity of ECMP load balancing

To measure load balancing quality we sample two hosts, each one from a POP equipped with a different switch model, and assign the same number of nexthops to each. By measuring the traffic volume served by either host at 5 minute intervals over a 24h period, we can calculate the actual fraction of traffic volume served by each host normalized by the average amount of traffic served by all hosts for each interval. Fig. 10a shows the distribution of this fraction for both switch models; the density concentration around unity shows that load balancing quality is good for both switches used in production. As shown in Sec. 6.3, this does not always hold (*e.g. Switch C*, Fig. 10b).

6 Operational experience

Faild derives much of its elegance from a core set of simplifying assumptions. This section revisits our assumptions in light of our operational experience and highlights their practical implications.

6.1 Recursive draining and POP upgrades

A limitation of the stateless architecture of Faild is that hosts that are actively forwarding drained traffic cannot be drained themselves. If any nexthops have MAC addresses with a `f:r` suffix (indicating that traffic for r is already being failed over to f), failing over host f to host g would imply updating these suffixes to have the form `g:f`, and the previous draining indirection from r to f would be lost. If f were still forwarding ongoing connections for r at this point, they would be reset by g as they would not have an associated open socket. This type of *recursive draining* fails because it cannot be effectively represented with our current MAC address encoding, and this means that Faild is unable to recover when a host that is already forwarding drained traffic fails.

While trivial to address by further overloading MAC address semantics, in practice this shortcoming is not a concern. For one, it is mitigated trivially by waiting for drained traffic to decay naturally. As long as the average time between failover reconfigurations is greater than the time needed for ongoing connections to end naturally once a draining episode is started, no customer traffic will be affected. Since edge traffic is preponderantly composed of small objects, flow completion time is low and drained traffic decay is fast, as observed in Fig. 9. Additionally, the probability for overlapping draining episodes is correspondingly low. As shown in Fig. 1, every one of our POPs had a worst-case draining rate of 5 events per minute or lower at the 99.9^{th} percentile, irrespective of underlying event cause. In our experience, the benefits of recursive draining do not justify the resulting increase in operational complexity.

Faild can implement seamless addition of hosts simply by draining an appropriate number of ECMP nexthops to the new hosts being deployed. However, instead of updating all MAC addresses with suffixes `r:r` associated with a drained host r, the ECMP nexthops to be reallocated are chosen from the entire forwarding table in such a way that the resulting configuration is balanced. After a host f is inserted, and once draining traffic has subsided, addresses with `f:X` suffixes can be re-labeled as `f:f` to distinguish the change as permanent. In order to simplify the addition of hosts to a POP, Faild switches are configured to use the maximum number of nexthops supported by the hardware, which are then allocated to the initial number of hosts in a balanced manner.

6.2 Scalability challenges

When scaling Faild to larger POPs, the main bottleneck in a single-layer topology will be the port density of switches; this can be mitigated by using alternative topologies with multiple switch layers. However, a challenge arises when performing ARP table updates on switches running Faild but not directly connected to

hosts: bridging table changes need to be synchronized. This can be achieved with either custom agents running on switches or using *e.g.* LLDP discovery.

Although our MAC address encoding limits POP size to 256 hosts, we have yet to come close to this limit. The tight latency bounds crucial for edge clouds push for geographically diverse POPs with a small number of high-end servers, counterbalancing the economies of scale driving conventional datacenter deployments. It would however be simple to extend the host encoding in MAC addresses to support larger POPs.

A further challenge arises when the maximum number of ECMP nexthops supported by the underlying switch hardware is insufficient to provide adequate load balancing. Recent commodity switches provide 2048+ nexthops, which we have found to be sufficient for our needs.

The current layer 2 architecture is attractive because it is efficient and widely supported and tested. Furthermore, IPv6 is trivially supported and kernel packet processing on the hosts is simpler than alternative encapsulation methods. If necessary however Faild can operate in the network layer without significant modifications. By simply using IP-in-IP ECMP nexthops bound with host IP addresses and with statically configured ARP tables, all the benefits that we have described become available in a layer 3 architecture.

6.3 ECMP hashing assumptions

While ECMP hashing plays an important role in most load balancing solutions [17, 27], it is central to Faild. Our original design and subsequent analysis make numerous assumptions which were empirically validated in Sec. 5. This, however, is a reflection of our own selection bias rather than representative for all switch models. While evaluating various hardware options, we came across numerous implementation flaws.

Uneven hashing. Some switches under evaluation were incapable of hashing evenly. The worst example was a switch that, when configured with 256 ECMP nexthops for a given destination, hashes traffic according to the ranked load distribution in Fig. 10b. For this particular switch model, the most and least heavily loaded of the 256 ECMP nexthops differ in allocated traffic share by a factor of approximately six. The impact of this cannot be understated, given that it is uncommon for customers to rigorously evaluate hashing, and instead rely on vendor claims to drive capacity planning. As a result, it is likely that many commercial networks reliant on ECMP are far more prone to overloading capacity than they realize.

Unusable nexthops. Some switches also have odd restrictions on the number of usable ECMP nexthops for any given destination. In particular, one model we tested appears to only support ECMP nexthop set sizes that are of the form $\{1,2,\ldots,15\} \times 2^n$, presumably because of hardware limitations. Configuring an ECMP route with a nexthop count not of this form will result on the next lower number of nexthops of this form to be used. For example, configuring this switch with an ECMP route with 63 nexthops will cause only 60 (*i.e.* 15×2^2) of the nexthops to be used, and 3 of the nexthops will thus receive no traffic at all. As a result, a switch may hash evenly amongst internal buckets, and yet still lead to a skew in load distribution because of a mismatch in how these buckets map to nexthops.

Hash polarization. For a Faild instance with multiple ingress switches, each switch should be configured to hash in the same manner. If this is not the case, an external routing change may divert a flow through a different switch, which in turn may hash the flow onto a different host leading to TCP connection resets.

Vendors however can make such hash polarization impossible to achieve in practice. ECMP hash polarization is often configured unintentionally by omission, and can lead to poor performance in networks employing multiple levels of ECMP routing, as it leads to correlation between ECMP nexthop decisions made at different levels of the network hierarchy. Some vendors have addressed this potential misconfiguration by introducing additional sources of entropy into their ECMP hashing functions.

Unfortunately we found that many switch models include the index of the ingress interface of a packet in that packet's ECMP nexthop hash computation, and in one particular case, we found that line-card boot order was used to seed the ECMP hashing function. If the hardware vendor does not provide a knob to disable such behavior, hash polarization is rendered impossible, which has dire consequences for our use case.

6.4 Protocol assumptions

One might expect that, in the absence of routing changes, packets belonging to the same flow follow a single network path. This however assumes that any load balancing along the path, including that applied by Faild itself, is consistent across all packets within a flow. This section reviews cases where this does not hold.

Inbound fragmentation. If we receive a TCP segment that has been fragmented by either the originating client or an intermediate router, its additional fragments will not contain the TCP port numbers for the connection, which will cause the receiving switch to hash the initial fragment and the set of additional fragments to different hosts in the POP.

IPv4-speaking clients that use IPv4 Path MTU Discovery [23] transmit TCP segments that have the IPv4 Don't-Fragment bit set, and will therefore not be fragmented by intermediate routers. Anecdotally, virtually all IPv4 clients we see traffic from implement Path MTU Discovery and transmit TCP segments that are unfragmented

and have the IPv4 Don't-Fragment bit set. In IPv6 packets are never fragmented by intermediate routers [12], and fragmented IPv6 TCP segments belonging to established connections are exceedingly rare. Considering this, we continue to allow switches to use port numbers for hashing, as this does not appear to cause operational problems. Nevertheless, a possible mitigation for this problem is to configure Faild switches to exclude the received segment's TCP port numbers in its packet hash computation. This would allow successful defragmentation and processing of received TCP segments, but could harm load balancing evenness.

Outbound fragmentation. Inevitably we will receive ICMPv4 Fragmentation Needed or ICMPv6 Packet Too Big messages in response to outgoing TCP segments destined for a network (or link) that uses an MTU lower than the segment size. As noted in [11], such ICMP messages, and ICMP errors in general, are not guaranteed to ECMP-hash back to the same host as the one that was handling the corresponding TCP session. Faild handles this by having hosts that receive certain ICMP messages from upstream switches rebroadcast these messages to all hosts in the local POP, and by processing received ICMP messages that were rebroadcast in this manner as if they were unicast to the local host. This ensures that the host that was handling the TCP session corresponding to an inbound ICMP message will receive and process a copy of that ICMP message. Given this is a potential denial-of-service vector, we also implemented a mechanism for rate limiting broadcasts.

ECN. In 2015 we experienced an increase in reports of connection resets coinciding with Apple enabling Explicit Congestion Notification [28] by default on iOS and OS X. Further investigation revealed that this did not affect all users; it was dependent on network path.

Prior to the deprecation of the IPv4 Type of Service field and its redefinition as the Differentiated Services field by RFC 2474 [24], and the reservation of the last two bits of the DS octet for ECN by RFC 3168 [28], it was explicitly permitted by the IPv4 router requirements RFC [7] to involve the second-to-last bit of the TOS/DS octet in routing decisions, and at least one major operator in the US and several operators outside the US appeared to have deployed network devices which take this bit into account in ECMP nexthop selection. Given that this can cause packets for a single ECN-using flow to follow different paths, there is the potential for trouble if such a flow hits a POP where Faild is operating across multiple switches which do not support hash polarization.

Given we are still phasing out switches which do not support hash polarization from our network, we decided instead to disable ECN negotiation entirely. The expectation that all packets within a flow follow the same path however is likely to be broken for ECN for a much longer period of time, as devices hashing on ECN bits are embedded in networks with long hardware refresh cycles.

SYN proxies. More recently, we uncovered a similar lack of packet affinity between our POPs and a major cloud provider which had deployed a SYN proxy for their enterprise platform. In this case, a software proxy undertakes the responsibility of establishing outbound connections, and then passes the established flow to the proxied host. In practice, this results in separate route lookups - one for the SYN handshake, and another for the subsequent data. While this corner case appears similar on the surface to ECN, it is much harder to work around. After extended discussions with the cloud provider in question, it was decided that it would be simpler to use BGP to pin an ingress path until we could upgrade our contingent switches to support hash polarization.

7 Related work

We now review how existing proposals fall short of the load balancing requirements for POP deployments. A comparative summary is provided in Table 1.

Consistent hashing ECMP provided by recent switches (*e.g.* [2, 4]) is a typical solution adopted by CDNs to achieve stateless load balancing but does not make it possible to gracefully add and remove hosts.

Ananta [27] and Maglev [15] propose the use of software load balancers, with the latter improving packet throughput through batch processing, poll mode NIC drivers and zero copy operations (also adopted by [13, 29]). Despite these improvements, both require per-flow state to provide connection persistence.

Duet [17] and Rubik [16] combine the ECMP capability of commodity switches with a software load balancer to address the performance bottleneck of general purpose hardware. They configure routing for heavy hitting VIPs directly on switches, and relegate less popular VIPs towards software load balancers. Although both add cursory support for migrating flow state between switches and software load balancers, this proves impractical to implement. For one, the approach assumes that the ECMP hash function and seed used on the switches can be replicated on load balancer instances. This however is typically proprietary knowledge which equipment manufacturers are unwilling or unable to disclose [30]. Even if this were not the case, the resulting implementation can still lead to flow disruption, as demonstrated in motivating the design decisions of SilkRoad [22].

SilkRoad [22] implements connection tracking in programmable switches by storing compressed flow state in SRAM memory, which can be potentially overrun. It also supports graceful draining of hosts without the high latency and low throughput of software load balancers, at the cost of requiring the switch control plane to execute a

Table 1: L4 load balancers: suitability comparison for POP deployment

System	Low latency	No dedicated HW	Stateless	Host draining	Switch draining	In production
Consistent ECMP [2, 4]	✓	✓	✓	✗	✓	✓
Ananta [27], Maglev [15]	✗	✗	✗	✓	✗	✓
Duet [17], Rubik [16]	✓	✗	partial	✓	partial	✗
SilkRoad [22]	✓	✓	✗	✓	✗	✗
Beamer [25]	✗	✗	✓	✓	✓	✗
Faild	✓	✓	✓	✓	✓	✓

table insertion for each received SYN. As a result of both these features, SilkRoad is vulnerable to resource exhaustion attacks and therefore not an appropriate match for load balancing within a POP. SilkRoad eliminates the need for load balancer instances to be deployed on hosts, but in doing so hinders the ability to drain switches.

The closest approach to Faild is Beamer [26, 25]. Beamer also implements consistent hashing by mapping a fixed number of buckets to hosts and uses detouring to drain hosts without keeping per-flow state, but is predated by Faild on both counts [34, 35, 36, 37]. However, whereas Faild does not require any additional hardware, Beamer requires dedicated hosts to run controllers and load balancer instances, making it unsuitable for deployment within a POP. Finally, the design of Faild is informed by years of operational experience. In addition to the benefits provided by Beamer, it provides explicit support for SYN cookies and maintains correct operation when facing anomalous cross-layer protocol interactions (*e.g.* inbound/outbound packet fragmentation).

A number of techniques used in Faild have also been used singularly albeit in different contexts and for different purposes. Extending MAC address semantics to implement switch-to-host signaling was used in [20] to allow switch forwarding based on application layer information. Mapping a large number of virtual nexthops to a much smaller number of physical nexthops was used in [40] to implement weighted multipath forwarding. Faild is unique in using these mechanisms to achieve graceful, stateless load balancing and having been validated extensively in production.

8 Conclusion

This paper revisits load balancing in the context of edge clouds, which are orders of magnitude more dense than datacenter-based cloud computing environments. Stripped of virtually unbounded physical space, even individual failures can have a noticeable impact on availability and cost becomes primarily driven by efficiency.

In light of these issues, this paper proposed Faild, a stateless, distributed load balancing system which supports *transport affinity*, hence allowing the graceful failover of any individual component. Constrained to commodity hardware, Faild redefines the semantics of existing network primitives and rethinks the allocation of load balancing functions across network components. We demonstrate that commodity switches can be leveraged to perform fast, stateless load balancing, while retaining the ability to signal failover forwarding information toward the hosts.

In optimizing our design for cost, we inadvertently implemented all functions necessary to achieve graceful failover. A key insight in this paper is that all state needed for graceful failover is readily accessible within the host kernel itself. Previous load balancers rely on additional per-flow state to track the allocation of flows to hosts: a costly, unscalable design pattern which is prone to resource exhaustion attacks. Faild instead inspects existing socket state to determine whether to failover traffic to neighboring hosts. Our results show that this is seamless to clients while incurring negligible CPU overhead and minimal delay. By keeping a tight focus on solving the engineering challenges of edge clouds, Faild combines well-known techniques in a novel way to achieve graceful addition and removal of any component without requiring per-flow state beyond that already present in the service hosts themselves.

This paper reflects our collective experience in scaling Faild over the past four years to handle in excess of seven million requests per second for some of the most popular content on the Internet. While the core design principles of Faild have fared remarkably well, we have strived to document cases where our assumptions proved overly optimistic. In exposing hardware limitations and unintuitive protocol interactions, we hope that with time these issues may begin to be addressed by a wider community.

Acknowledgments

We would like to thank our shepherd KyoungSoo Park and the anonymous reviewers for their feedback.

Faild was originally conceived by Artur Bergman and Tyler McMullen, and owes much to the many engineers at Fastly who contributed to its implementation and operation over the past four years.

References

[1] A10 Networks. https://www.a10networks.com/.

[2] Broadcom Smart-Hash. https://docs.broadcom.com/docs/12358326.

[3] F5 Networks. https://f5.com/.

[4] Juniper Networks - Understanding the Use of Resilient Hashing to Minimize Flow Remapping in Trunk/ECMP Groups. http://www.juniper.net/techpubs/en_US/junos15.1/topics/concept/resilient-hashing-qfx-series.html.

[5] Switch Abstraction Interface. https://github.com/opencomputeproject/SAI/.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '08)*.

[7] F. Baker. Requirements for IP Version 4 Routers. IETF RFC 1812 (Proposed Standard), June 1995.

[8] D. J. Bernstein. SYN cookies. http://cr.yp.to/syncookies.html.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[10] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*.

[11] M. Byerly, M. Hite, and J. Jaeggli. Close Encounters of the ICMP Type 2 Kind (Near Misses with ICMPv6 Packet Too Big (PTB)). IETF RFC 7690 (Informational), Jan. 2016.

[12] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. IETF RFC 2460 (Draft Standard), Dec. 1998.

[13] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*.

[14] W. Eddy. TCP SYN Flooding Attacks and Common Mitigations. IETF RFC 4987 (Informational), Aug. 2007.

[15] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI '16)*.

[16] R. Gandhi, Y. C. Hu, C. kok Koh, H. Liu, and M. Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *USENIX Annual Technical Conference (USENIX ATC '15)*.

[17] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '14)*.

[18] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. IETF RFC 2992 (Informational), Nov. 2000.

[19] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient traffic splitting on commodity switches. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '15)*.

[20] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman. Be fast, cheap and in control with SwitchKV. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, (NSDI '16)*.

[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 Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[22] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful Layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[23] J. Mogul and S. Deering. Path MTU discovery. IETF RFC 1191 (Draft Standard), Nov. 1990.

[24] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. IETF RFC 2474 (Proposed Standard), Dec. 1998.

[25] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu. Stateless datacenter load-balancing with Beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*.

[26] V. Olteanu and C. Raiciu. Datacenter scale load balancing for multipath transport. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox '16)*.

[27] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '13)*.

[28] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. IETF RFC 3168 (Proposed Standard), Sept. 2001.

[29] L. Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC '12)*.

[30] L. Saino. Hashing on broken assumptions. In NANOG '70. https://www.nanog.org/sites/default/files/1_Saino_Hashing_On_Broken_Assumptions.pdf, June 6th, 2017.

[31] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[32] B. W. Silverman. *Density estimation for statistics and data analysis*. Chapman and Hall, London, 1986.

[33] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '15)*.

[34] J. Taveira Araújo. Communication continuation during content node failover. US Patent 9,678,841. Filed May 30th, 2014, Issued Jun. 13th, 2017.

[35] J. Taveira Araújo. Failover handling in a content node of a content delivery network. US Patent 9,569,318. Filed May 30th, 2014, Issued Feb. 14th, 2017.

[36] J. Taveira Araújo. Scaling networks through software. In *USENIX SREcon '15*.

[37] J. Taveira Araújo, L. Saino, and L. Buytenhek. Building and scaling the Fastly network, part 2: balancing requests. `https://www.fastly.com/blog/building-and-scaling-fastly-network-part-2-balancing-requests/`, Dec 2016.

[38] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. IETF RFC 2991 (Informational), Nov. 2000.

[39] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with Espresso: Scale, reliability and programmability for global Internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*.

[40] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: Weighted Cost Multipathing for improved fairness in data centers. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*.

Stateless Datacenter Load-balancing with Beamer

Vladimir Olteanu, Alexandru Agache, Andrei Voinescu and Costin Raiciu
University Politehnica of Bucharest

Abstract

Datacenter load balancers (or muxes) steer traffic destined to a given service across a dynamic set of backend machines. To ensure consistent load balancing decisions when backends come or leave, existing solutions make a load balancing decision per connection and then store it as per-connection state to be used for future packets. While simple to implement, per-connection state is brittle: SYN-flood attacks easily fill state memory, preventing muxes from keeping state for good connections.

We present Beamer, a datacenter load-balancer that is designed to ensure stateless mux operation. The key idea is to leverage the connection state already stored in backend servers to ensure that connections are never dropped under churn: when a server receives a mid-connection packet for which it doesn't have state, it forwards it to another server that should have state for the packet.

Stateless load balancing brings many benefits: our software implementation of Beamer is twice faster than Google's Maglev, the state of the art software load balancer, and can process 40Gbps of HTTP uplink traffic on 7 cores. Beamer is simple to deploy both in software and in hardware as our P4 implementation shows. Finally, Beamer allows arbitrary scale-out and scale-in events without dropping any connections.

1 Introduction

Load balancing is an indispensable tool in modern datacenters: Internet traffic must be evenly spread across the servers that deal with client requests, and even internal datacenter traffic between different services is load balanced to ensure independent scaling and management of the different services in the datacenter.

Existing load balancer solutions can load balance TCP and UDP traffic at datacenter scale at different price points [26, 13, 9, 22, 15, 31, 12, 18]. However, they all keep per-flow state: after a load balancer decides which server should handle a connection, that decision is "remembered" locally and used to handle future packets of the same connection. Keeping per-flow state should ensure that ongoing connections do not break when servers and muxes come or go, but has fundamental limits:

- Standard scaling events that include both muxes and servers break many ongoing connections.
- SYN flood attacks prevent muxes from keeping "good" connection state, negating its benefits.
- Running stateful load-balancers in software with many flows reduces throughput by 40% (§6.1).

In this paper we design, implement and test Beamer, a **stateless and scalable datacenter load balancer** that supports not only TCP, but also Multipath TCP [27]. The key idea behind Beamer is *daisy chaining* that uses the per-connection state already held by servers to forward occasional stray connections to their respective owners.

Our prototype implementation can forward 33 million minimum-sized packets per second on a ten core server, twice as fast as Maglev [9], the state of the art load balancer for TCP traffic. Our stateless design allows us to cheaply run Beamer in hardware too, as shown by our P4 implementation (§5). Beamer can scale almost arbitrarily because each load balancer acts completely independently and holds no per-connection state. Our experiments show that Beamer is not only fast, but also extremely robust to mux and server addition, removal or failures as well as heavy SYN flood attacks.

2 Background

Services in datacenters are assigned public IP addresses called VIPs (virtual IP). For each VIP, the administrator configures a list of private addresses called DIPs (direct IPs) of the destination servers. The job of the load balancer is to load balance connections destined to the VIPs across all the DIPs. Hardware load balancing appliances have long been around and are still in use in many locations; however they are difficult to upgrade or modify and rather expensive. Traditional app-level proxies such as HAProxy or Squid that terminate the client's TCP connection and open a new one to the server are also not desirable because their performance is quite low.

A raft of load balancers based on commodity hardware have been proposed that seek to address the shortcomings of existing solutions [26, 9, 13, 12, 18, 22, 15, 31]. Their goal is to process packets as ***cheaply*** as possible, while ***balancing load evenly*** across a dynamically changing population of backend servers and ensuring ***connection affinity***: all packets of a connection should reach the same server.

Almost all existing load balancers follow the same architecture introduced by Ananta [26] and we provide a brief description in Figure 1. In Ananta, load balancing is performed using a combination of routing (Equal Cost Multipath) and software muxes running on commodity x86 boxes. All muxes speak BGP to the border datacenter router and announce the VIPs they are in charge of as accessible in one hop. The border router then uses equal-cost multipath routing (ECMP) to split the traffic equally

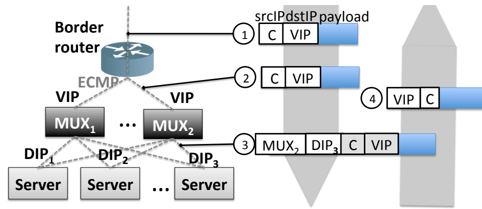


Figure 1: Load balancing: traffic to the VIP address is load-balanced across a pool of servers, each with a DIP address. Return traffic bypasses the muxes.

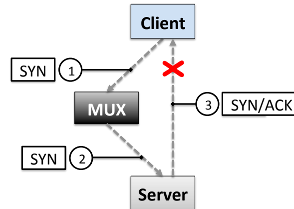


Figure 2: Mux and server disagree over the status of a connection.

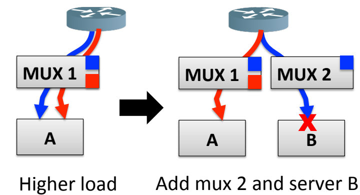


Figure 3: Scale out: stateful load balancers break TCP connections.

to these muxes. When a connection starts (i.e. a SYN packet arrives at a mux), a hash function is applied on the five-tuple and a server is chosen based on this hash. If a single server is added to the DIP pool, the assignment of some existing connections to servers will change; at the very least, the new server must receive an equal fraction of all ongoing connections. That is why, once a mapping of connection to DIP is chosen, it is stored locally by the mux to ensure all future packets will go to the same DIP.

Upon leaving the mux, the original packet is encapsulated and sent to the DIP. The receiving server first decapsulates the packet, changes the destination address from VIP to DIP, and then processes it in the regular TCP stack. When the reply packet is generated, the source address is changed from DIP to VIP and the packet is sent directly to the client, bypassing the mux to reduce its load (this is Direct Source Return, or DSR).

3 Limits of stateful load balancing

A key design decision of all existing load balancers is to keep a small amount of per-connection state to ensure *connection affinity*: once a connection is assigned to a backend, the mux will remember this decision until the connection finishes or a timer fires.

While per-connection state works well in the average case, it has a number of fundamental limitations which reduce its effectiveness in practice. First, because the mux only sees one direction of traffic, state kept by the mux can differ from the server's state for the same connection; the worst case here is the muxes' inability to cope with SYN flood attacks. Secondly, even in the absence of SYN floods, connections will be broken in scale-out or scale-in (or failure) events where both the mux set and the DIPs change simultaneously; such events happen naturally. Finally, software muxes' forwarding performance decreases with many active connections (see §6). We discuss these issues next.

State mismatch between mux and server. Consider the simple example in Figure 2: a client starts a TCP connection by sending a SYN packet, which is seen by a mux and then redirected to a server, and the mux saves the chosen mapping locally. The server replies with a SYN/ACK packet which never reaches its destination because the client is now disconnected. The server will send this packet a few times until it terminates the connection; the mux however is not aware of the reverse path unreachability and will maintain the state for minutes.

SYN flood attacks, where attackers send many SYN packets with spoofed IP source addresses [8], cause similar problems. During a SYN flood, the SYN/ACKs sent by the server never reach their destination, but both the server and the mux install connection state. SYN-cookies [8] are the standard protection against SYN flood attacks: when the number of half-open connections reaches a threshold, the server stops keeping state upon receiving a SYN, encoding the state in the SYN/ACK packet it sends to the client. When legitimate customers reply with the third ACK to finalize the connection handshake, the server uses the information from the ACK number (a reflection of its initial sequence number) and the timestamp (the echo reply field) in conjunction with local information to check if this is a valid connection; if so, it creates an established connection directly.

SYN cookies help the server shed unwanted state, but have no positive effect at the mux: the mux is forced to allocate state for every SYN it sees. Under a SYN flood attack, the servers will function normally but the muxes' connection memory will be overloaded to the point where they will behave as if they have no connection state, and thus DIP churn will break connections.

Ensuring state synchronization and defending against SYN floods at muxes is far from trivial: at the very least it requires muxes to keep more state (i.e. is the server sending SYN cookies or not?) and examine both directions of traffic; another cleaner solution is for the mux to terminate TCP. All solutions limit scalability.

Connection failures during scaling events. Even without SYN floods, keeping mux state does not guarantee connection affinity. Figure 3 shows a datacenter service that is running with one mux and one server. As load increases, one more mux and server are added. The border

router now routes half the connections it sent to mux 1 to mux 2, as the blue flow in our example. Mux 2 does not have state for the blue flow, and it simply hashes it, assigns it to B and remembers the mapping for future packets. B receives packets from an unknown connection so it resets it. Such failures happen even when the border router uses resilient hashing and when all muxes use the same hash function. The necessary condition, though, is that both the mux set and the server set changes in quick succession, but such sequences of events occur naturally in datacenters during scale out and scale in events.

4 Beamer: stateless load-balancing

Using per-flow state at muxes fails to provide connection affinity in many cases. Can we do better without keeping flow state in the muxes? This is our goal here.

To achieve it, we leverage the per-flow state servers already maintain for their active connections. As an example, consider server B in Fig. 3: it receives a packet belonging to the blue connection, for which it does not have an entry in the open connections table; the default behaviour is to reset the blue connection. If B knew that server A might have state for this connection, it could simply forward all packets it doesn't have state for, including the blue connection, to A, where they could be processed normally. We call such forwarding between servers *daisy chaining* and it is the core of Beamer.

The architecture of Beamer mirrors that in Figure 1: our muxes run BGP (Quagga) and announce the same VIP to border routers. ECMP at the routers spreads packets across the muxes, which direct traffic to servers; finally the servers respond directly to clients. To build a scalable distributed system around daisy chaining, Beamer uses three key ingredients:

- *Stable hashing* (§4.1), a novel hashing algorithm that reduces the amount of churn in DIP pool changes to the bare minimum, while ensuring near-perfect load balancing and ease of deployment.
- A *fault-tolerant control plane*(§4.5) that scalably disseminates data plane configurations to all muxes.
- An in-band signaling mechanism that gives servers enough information for daisy chaining, without requiring synchronization (§4.2).

4.1 Stable hashing

Beamer muxes hash packets independently to decide the server that should process them. A good hashing algorithm must satisfy the following properties: it should load balance traffic well, it should ensure connection affinity under DIPs churn, and it should be fast.

A strawman hashing algorithm is to chose the target server by computing `hash(5tuple)%N`, where N is the number of DIPs; this is what routers use for ECMP. This mechanism spreads load fairly evenly and as long as the set of DIPs doesn't change, and mux failures or additions do not impact the flow-to-DIP allocations. Unfortunately, when a single server fails (or is added), most connections will break because the modulus N changes.

Consistent hashing [19], rendezvous hashing [30] and Maglev hashing [9] all offer both good load balancing and minimize or at least reduce disruption under churn. On the downside, in all these algorithms each server is in charge of *many* discontiguous parts of the hash space; this means the mux must match five-tuple hashes against many rules, reducing performance (for software deployments) or increasing hardware cost (for hardware ones). These algorithms target wide-area distributed systems and thus strive to reduce (mux) coordination. In datacenters, however, we can easily add lightweight coordination which enables a simple and near-optimal hash algorithm.

Beamer implements *stable hashing*, an extensible hashing approach that can be used to implement all the algorithms above. *Stable hashing* adds a level of indirection: connections are hashed against a fixed number of buckets, and each bucket can be mapped by the operator to any server. Before the load balancing service starts for a certain VIP, the operator chooses a fixed number of buckets B that is strictly larger than N, the maximum number of DIPs that will serve that VIP (e.g. `B=100N`). Each bucket is assigned to a single server at any time, and each server may have multiple buckets assigned to it. The number of buckets B and the bucket to server assignments are known by all muxes, and they are disseminated via a separate control plane mechanism (see §4.5). When a packet arrives, muxes hash it to a bucket by computing `b=hash(5tuple)%B`, and then forward the packet to the server currently assigned bucket *b*. As B is constant by construction, server churn does not affect the hashing result: a connection always hashes to the same bucket, regardless of the number of active DIPs.

Bucket-to-server mappings are changed on server failure or explicitly by the administrator for load-balancing and maintenance purposes. These mappings are stored in reliable distributed storage (Apache ZooKeeper [16] in our implementation); muxes retrieve the latest version before they start handling traffic. As changes to the bucket-to-DIP mappings are rare, this mechanism has low communication overhead and scales to datacenter-sizes (§6.4).

We show an example of stable hashing in Figure 4. The administrator has configured four buckets; muxes first hash flows into these buckets to find the destination

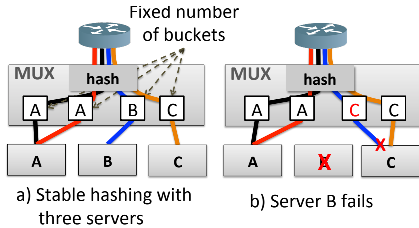


Figure 4: Stable hashing is resilient to server failures.

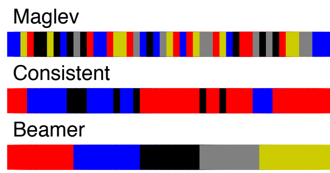


Figure 5: Hashing algorithms comparison

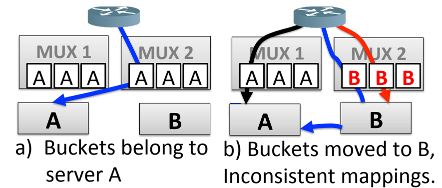


Figure 6: Daisy chaining allows server addition or removal without disrupting ongoing connections.

server. When server B fails, the controller will move the third bucket from B to C; the mapping is then stored in ZooKeeper and disseminated to all muxes. After the mapping is updated, flows initially going to A or C are unaffected, and flows destined for B are now sent to C. Only the blue connection, handled by B, is affected.

Our bucket-to-server mappings are managed centrally by the controller. The controller has the freedom to implement any bucket-to-server mapping strategy to mimick consistent hashing, rendezvous hashing or Maglev. In our implementation we chose a greedy assignment algorithm that aims to maximize the contiguous bucket ranges assigned to muxes; this is very useful especially when Beamer is deployed in hardware, because it can use fewer TCAM rules to implement its dataplane functionality. To provide intuition about why this is the case, in Fig. 5 we show how 47 buckets are assigned to 5 servers using the three algorithms, where each server is shown in a different colour: the bigger the fragmentation, the higher the cost to match packets against packets in the dataplane. Beamer is the least fragmented, followed by Consistent and Maglev. When servers come and go, bucket assignments to servers will also become fragmented even with Beamer; Beamer runs a periodic defragmentation process to avoid this issue (see §4.5).

Our stateless design ensures that mux churn has no impact on connections: as soon as BGP reconverges to the new configuration, load will be spread equally across all muxes and no connections will be broken.

4.2 Daisy chaining

There is a natural amount of churn of servers behind a VIP, be it for load balancing purposes or for planned maintenance. To implement such a handover, all our controller has to do is to map to the new server the buckets belonging to the old server and store the new mappings in ZooKeeper. The muxes will then learn the new mapping and start sending the bucket traffic to the new server. For a truly smooth migration, however, there are two complications that need to be taken in account: existing connections will be broken and there might be inconsistencies when some muxes use the new mappings while others are using the old ones.

To solve both issues we use *daisy chaining*, a transitory period where both the new server and the old one are active and servicing flows that hit the migrated bucket(s). We aim to move all new connections to the new server, and process ongoing connections by forwarding them to the old server even if they arrive at the new server.

We give an example in Fig. 6 where we migrate three buckets between servers A and B. Initially, both muxes have the same mappings for all buckets. Daisy chaining starts when the controller migrates the buckets from A to B by storing the new mapping in ZooKeeper and marking A as the previous DIP, along with the timestamp of the update. Note that the muxes save the previous DIP for each bucket, as well as the time the reallocation took place. To see how daisy chaining comes into play, let us consider the way packets are processed upon reception by the server. If the incoming packet is a SYN (TCP or MPTCP), a valid SYN-cookie ACK, or it belongs to a local connection, then we can process it locally. Otherwise, the packet could belong to a connection that has been previously established on another server. In this case, we want to daisy-chain packets back to the appropriate server, but only for a limited time.

To enable this, packets destined to ports lower than 1024 (higher numbers are used for MPTCP load balancing, see §4.4) also carry the previous DIP and the timestamp of the change (or 0 when there is none). We always save locally the highest timestamp seen for the bucket the packet is hashed to, and enable daisy chaining to the previous DIP when current time is smaller then the timestamp plus the daisy chaining interval. Thus, packets are redirected to the appropriate destination as long as daisy chaining is active. Otherwise, they are dropped and a RST is sent back to the source.

Daisy chaining adds robustness to our whole design. Consider what happens if the two muxes in Fig. 6 temporarily disagree on the server now in charge of the three buckets. Flows that hit mux 1 are load balanced according to the old mapping and will be directed to A, who

will simply process them locally (the black connection). Meanwhile, B will locally service the red connection, but will daisy chain the blue connection to A (the previous DIP) because it doesn't have state for it. When mux 1 finally updates its state, the black connection will be sent to B, and daisy chained back to A (assuming the rule is still active). After all the muxes have updated their state, A will only receive packets related to ongoing connections, which will quickly drop in number. While in principle daisy chaining can be left running forever, we try to avoid migrating buckets that are being daisy chained. That is why our Linux kernel implementation uses a hard timeout of four minutes for daisy chaining.

There is one subtle corner case where daisy chaining still doesn't protect against broken connections, and we exemplify in figure 6. Consider the red connection that is being serviced by B after mux 2 updates its configuration; if this connection is sent to mux 1 (e.g. via ECMP churn in BGP), mux 1 will send it to server A which will reset it. To avoid this problem, packets also carry the generation number for the dataplane information, and all servers remember the highest generation number they have seen. In this example, server A will receive packets from B (and possibly from other muxes) with generation 2 and will remember 2 as the latest generation. When A receives a mid-connection packet that can not be daisy chained and for which it has no state, it will check if the generation number from the mux equals the highest generation number seen; if yes, the connection will be reset. If not, the server silently discards the packet. This will force the client to retransmit the packet, and in the meantime the stale mux mappings will be updated to the latest generation, solving the issue. Note that, if the border router uses resilient hashing, the mechanism above becomes nearly superfluous.

4.3 Mux data plane algorithm

The mux data plane algorithm pseudocode is shown in Fig.7. Lines 3-9 handle regular TCP traffic: first the bucket b is found together with the current and previous DIPs for bucket b. After that, the packet is encapsulated and sent to the current DIP. The algorithm is very simple, requiring a hash and one memory lookup in the buckets matrix (all three columns easily fit in one cache line). The remaining code in the mux performs Multipath TCP (MPTCP) [27] traffic load balancing equally cheaply: a single lookup is needed and the packet is encapsulated and sent to the appropriate DIP (see §4.4).

The simplicity of the mux is key to good performance: on one core our prototype can handle around 5-6Mpps, and around 33Mpps on an ten core Xeon box.

```
 1 packet* mux(packet* p){
 2   if (p->dst_port<1024){
 3     gen = buckets.version
 4     b = hash(5-tuple)%B;
 5     dip = buckets[b][0];
 6     pdip = buckets[b][1];
 7     ts = buckets[b][2];
 8
 9     return encapsulate(mux,dip,pdip,ts,gen,p);
10   }
11   else {
12     dip = id[p->dst_port];
13     return encapsulate(mux,dip,p);
14   }
15 }
```

Figure 7: Mux data plane pseudocode

4.4 Handling Multipath TCP

MPTCP deployment on mobiles is spreading: all IOS-based phones have it, as do top-end Android devices such as Galaxy S7 / S8. None of the existing datacenter load balancers support MPTCP, unfortunately, and this is a barrier to server-side deployment. This is because load balancing MPTCP is more difficult than regular TCP.

An MPTCP connection contains one or more subflows, and it starts when its first subflow is created. Each subflow looks very much like an independent TCP connection to the network, with the exception that its segments carry MPTCP-specific options. When load balancing MPTCP, all subflows of the same connection must be sent to the same DIP. Existing datacenter load balancers (e.g. Ananta [26], Maglev [9], SilkRoad [22], Duet [13]) treat MPTCP subflows as independent TCP connections, thus the DIP for each subflow will be decided independently, sending them to different servers most times, and breaking secondary subflows.

In MPTCP, after the initial subflow is setup, each endpoint computes the token—a unique identifier its peer has assigned to this connection. This token is embedded in the handshake of additional subflows within the same MPTCP connection and helps the remote end find the appropriate connection to bind the subflow to.

If one follows the mux state design approach, implementing MPTCP support requires storing the server-token T_B to DIP mapping in some shared memory all muxes can access, but this poses two problems: first, since only the DIP knows the token, it should update the shared memory when a new connection is created; secondly, having a shared memory access for each additional subflow would be prohibitive from a performance point of view.

We propose a stateless solution that leverages the mobility support available in MPTCP to ensure that secondary subflows can be forwarded to the correct server. We use the destination port in SYN JOIN packets to encode the server identifier. This is shown in Fig. 8: when

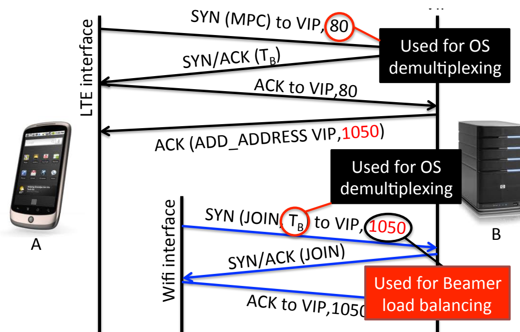


Figure 8: Load balancing MPTCP statelessly. Beamer uses address advertisement to embed the server identifier in the destination port of secondary subflows.

receiving TCP SYN or MPTCP initial subflow SYN packets, the port number is used to find the listening socket. However, SYN JOIN packets (handshake of secondary subflow) contain a token (T_B) that servers use to find the existing MPTCP connection [11]; the destination port is not used by the stack and we use it for Beamer.

Before deployment, Beamer assigns each server a unique identifier in the 1025-65535 range. We reserve port numbers (1-1024) for actual services, and utilize the remaining port numbers to encode server identifiers for secondary subflows. MPTCP allows endpoints to send `add address` options that specify another address/port combination of the endpoint to be used in future subflows. We use this functionality as shown in Fig. 8: whenever a new MPTCP connection is established (i.e. the third ACK of the first subflow is received), servers send an ACK with `add address` option to the client with the VIP address and the server identifier as port number. The client remembers this new address/port combination and will send subsequent subflows to it.

To handle MPTCP secondary subflows correctly, our mux (Fig. 7) treats traffic differently depending on the packet's destination port: traffic to ports greater than 1024 are treated as secondary subflows and directed to the appropriate servers. As each server has exactly one port associated to it, our solution can support at most 64K servers for each VIP. The muxes use another indirection table called `id`, that simply maps port numbers to DIP addresses (identified with D_i here, see Fig. 9).

Note that we only need daisy chaining to redirect initial subflows of MPTCP connections or plain TCP connections. Secondary subflows are sent directly to the appropriate server (uniquely identified by the port).

4.5 Beamer control plane

We have designed our control plane to be scalable and reliable and built it on top of ZooKeeper. ZooKeeper ensures reliability by maintaining multiple copies of the data and using a version of two-phase commit to keep the copies in sync. Users can create hierarchies of nodes, where each node has a unique name and can have data associated to it, as well as a number of child nodes.

We show the operation of our control plane by detailing how the most important operations are implemented. The controller is the only machine that writes information into ZooKeeper and muxes only read ZooKeeper information. Servers do not interact with ZooKeeper at all. The node hierarchy used by Beamer is shown in Fig. 11.

When a new Beamer instance is created, the controller creates a high level node (called in this example "beamer") and a "config" child node holding the basic configuration information including the VIP and the total number of buckets. Next, the operator can add DIPs to the load balancer instance by specifying the DIP address, an identifier (unique within an instance) and a weight.

The bucket-to-server assignments are stored in the "mux_ring", while the "dips" node contains DIP-related metadata, which is not read by the muxes.

Creating a DIP. To add a DIP, the controller will add an entry for the DIP in the "dips" node, and then in the "id" node. ZooKeeper guarantees that all individual operations are atomic. If the controller or its connection to ZooKeeper crashes at any point, it checks the "dips" node for any in-progress DIP additions. If a DIP is not represented in the "id" node, it is added there as well.

Load balancing is run after one or more DIPs are added, before they are removed or after their weight is changed. The assignment algorithm runs in a loop, aiming to balance load properly while reducing daisy chaining:

- Select the most overloaded server `A` and underloaded server `B`, where load is the ratio between assigned buckets and weight.
- Find the maximum number of buckets `n` such that, if transferred from `A` to `B`, `A`'s load would not fall under the average, and `B`'s load would not rise above.
- Select the `n` buckets that have been in `A`'s pool for the longest time, and move them from `A` to `B`.

To move buckets between two servers, the controller simply updates the mux ring (see below). Our greedy bucket-to-DIP assignment algorithm will cause fragmentation when the DIP set is altered and buckets are reassigned to ensure good balancing. Beamer includes a defragmentation algorithm (see Appendix) that runs when fragmentation exceeds a threshold.

Removing a DIP begins by setting its weight to zero. After running the load balancing algorithm, the "dips" entry is removed.

Updating mux dataplane configuration safely. The muxes load the dataplane configuration from the

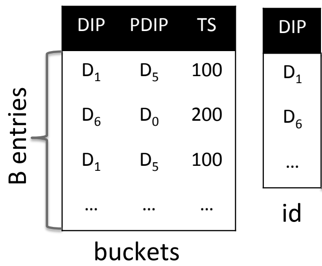


Figure 9: Mux data structures

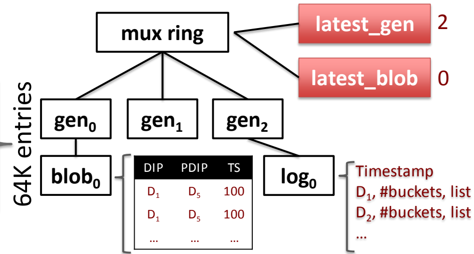


Figure 10: Mux configuration information stored in ZooKeeper

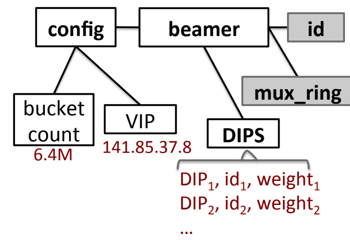


Figure 11: Controller information stored in ZooKeeper

"mux_ring" node in ZooKeeper, as shown in Fig. 10. The dataplane information is stored in several generation nodes, that have *logs*, which capture incremental updates to bucket ownership. Optionally, a generation node may also have a *blob*, which is an entire snapshot of the dataplane. The logs and blobs are compressed using zlib [1], and may span multiple nodes[1].

The blob contains the same data structure used by the muxes to forward packets. When it starts up, the mux first reads the values of "latest_blob" (the newest generation that contains a blob, in this case "gen0"). The mux reads the blob from "gen0" and obtains a functional forwarding table. If the "latest_gen" node has a value greater than the latest blob, the mux reads all the generations in ascending generation number order and applies the deltas contained therein. The mux now has an up-to-date forwarding table and can process packets.

ZooKeeper allows clients to register watches on nodes and it delivers notifications when the nodes' data is updated. We leverage this functionality to inform muxes that the forwarding information has changed: all muxes register watches for the "latest_gen" node; when it changes, the muxes will fetch and apply the new deltas.

Finally, the controller updates the mux ring information with the following algorithm: a) create a new generation node and store the updates to the bucket-to-server assignments, and b) update the "latest_gen" node to inform the muxes of the new version. The controller also creates blobs by applying the deltas in the same way the muxes do, creating the blob nodes under the current generation and then updating the "latest_blob" entry.

Safety. The controller algorithm above does not require any synchronization between muxes or the controller beyond ZooKeeper interactions. To ensure correctness, it maintains the following invariants: a) Muxes only read ZooKeeper information; they never update it. Configuration information is only written by the fault-tolerant controller; b) State updates are atomic from the muxes' point of view: they occur when the "latest_gen" node is changed (an atomic ZooKeeper operation), which only occurs after the controller has finished writing the data pertaining to the newest generation; and c) Generations with an identifier smaller than "latest_blob" can be safely deleted by the controller since muxes do not need them to have an up-to-date version of the dataplane.

[1] ZooKeeper nodes have a maximum size of 1MB.

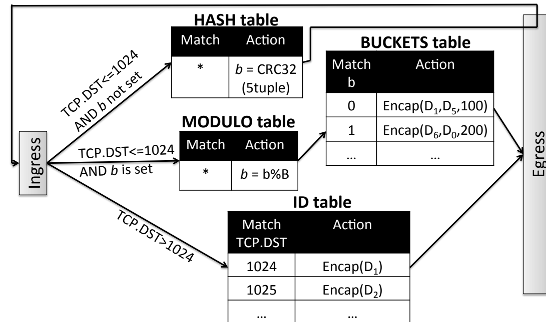


Figure 12: P4 implementation of a Beamer mux.

5 Implementation

Beamer servers run a kernel module (1300LOC) that handles decapsulation, address mangling and daisy-chaining. We have also patched the MPTCP Linux kernel implementation (version 0.90) to advertise the server ID for subsequent subflows (a few tens of lines of code). Our controller is implemented in 2100 lines of Java.

We have implemented Beamer muxes both in software and hardware (P4). The software mux runs a Click configuration atop the FastClick suite [4]. FastClick enables scaling to multiple cores, sets thread to core affinities and directly assigns NIC queue interrupts to cores.

Software mux. The core of the software mux is a Click element we have developed that implements our mux algorithm and acts as a ZooKeeper client to receive state updates. To improve performance, our design is completely lock free, which we achieve by carefully ordering the way we update the buckets matrix during updates.

Our hardware mux implementation is based on P4 [5] and is shown in Fig. 12. It contains two match-action

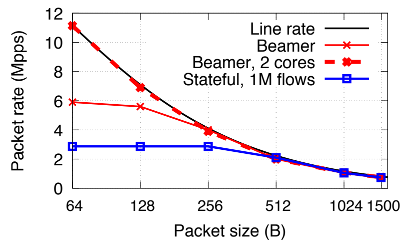


Figure 13: Forwarding performance vs. packet size (one core). Beamer outperforms stateful design 2-3x.

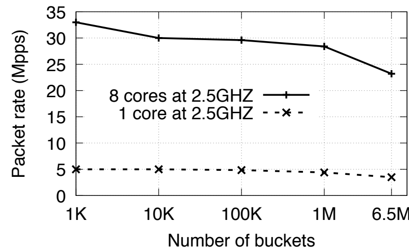


Figure 14: Forwarding performance (ten-core Xeon, 4 x 10Gbps). Beamer forwards 40Gbps with 128B packets.

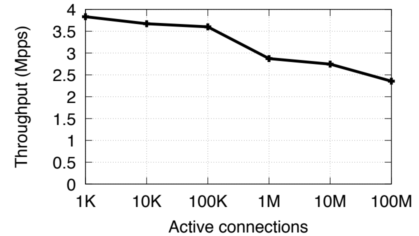


Figure 15: Software mux performance decreases with more active connections.

tables, one for the bucket-to-server mappings and one for the id-to-server mappings. The control part of the mux simply directs packets to one of these tables based on their destination port. The tables are populated by a software control plane that speaks to ZooKeeper.

The biggest challenge is computing the hash of the 5-tuple that is needed for lookup in the buckets table in the ingress stage of the pipeline: we can use the stock CRC32 function to compute it, but the checksum is calculated only in the egress stage. Since we cannot compute the CRC manually, we "recirculate" the packet instead: when the packet first enters the pipeline, its hash is calculated and stored as metadata "b" and the packet is resubmitted to the ingress port. To compute the modulus we add one more table with a single default entry where the modulus in computed in the action. Finally, the packet hits the buckets table and is encapsulated.

6 Evaluation

The purpose of our evaluation is to test the performance, correctness, fault tolerance and deployability of our prototype implementation. We used our local testbed containing 20 Xeon-class servers connected directly to an 48-port 10Gbps BGP router to test dataplane performance and perform microbenchmarks of our control plane. We ran experiments on Amazon EC2 to show that our controller can scale to a large Beamer instance with one hundred muxes, 64K DIPs and 6.4 million buckets. In the appendix we also evaluate stable hashing.

Our results show that Beamer is simultaneously fast and robust: no connections are ever dropped, in contrast to stateful approaches, Beamer's dataplane performance is twice that of the best existing software solution, and our mux introduces negligible latency when underloaded ($100\mu s$). The control plane experiments highlight the robustness and scalability of our design.

6.1 Micro-benchmarks

We first tested our software mux in isolation handling 1000 buckets. The server used for testing has a ten core Intel Xeon processor running at 2.7GHz, 16GB of RAM and a ten gigabit NIC using the 82599 Intel chipset. Our traffic generator is based on the *pkt-gen* utility from the netmap [29] suite. The generator can saturate a 10Gbps link with minimum sized packets. In each experiment we generate packets of a single size and we measure performance at the receiver using pkt-gen.

The source code of previous datacenter load balancers including Ananta and Maglev is not publicly available. To compare against such solutions, we implemented *Stateful*, a version of our mux that uses a hash table to store per flow load balancing decisions. We use *Stateful* to understand the performance of stateful load balancers.

The results are shown in Fig. 13. First, we note that the stateful design, running with 1 million active flows (a typical load seen in production [22]), is significantly slower than Beamer, because flow table lookups and insertions are comparatively expensive and result in cache-thrashing. Fig. 15 shows performance as a function of the number of active flows. Throughput drops from 3.9Mpps with one thousand active flows to 2.3Mpps with 100 million active flows. The performance results presented in the Maglev paper ([9], Fig. 8) are comparable to those of *Stateful*: 2.8Mpps per core and 12Mpps for six cores. Beamer forwards 6Mpps per core, twice faster.

To see how Beamer scales, we also increased the number of cores it uses to service the single NIC while spreading the NIC queues across the cores. With at least two cores, the Beamer software mux achieves line rate for all packet sizes. Note that the maximum throughput with 64B packets is lower than the expected 14.88Mpps because of the overhead of the encapsulation we use: our mux adds an IP-in-IP encapsulation header (20B) to all packets, and an IP option (16B) to packets to ports smaller than 1024.

Finally, we installed four ten gigabit NICs into a Xeon server with ten cores at 2.5GHZ per socket. The per-core forwarding performance on this machine is 10% slower

than in the above experiments because the CPU is 10% slower. This setup allows us to test just how much traffic a software mux can handle if it uses all its resources.

We used four clients and four servers each with one 10Gbps NIC to saturate our MUX with 64B packets. We also varied the number of buckets to see how our design copes with larger server populations. Per core throughput with 64B packets drops to 5.6Mpps when the mux handles 100K buckets, and to 5.1Mpps when there are 1M buckets. The results are due to decreased cache locality when the memory needed to store the bucket information increases. A mux implementation could coalesce neighbouring buckets that point to the same server to reduce the number of effective buckets, thus increasing performance.

The total throughput per mux is shown in Fig.14: our mux can forward 23 to 33Mpps per server, or 20 to 30Gbps depending on the number of buckets. With 128B packets the mux saturates all interfaces (40Gbps).

Performance with real traffic. We used MAWI [21] traces to estimate the throughput of our mux in realistic traffic conditions, and to estimate how many web servers could be handled by a single mux. We built a replay tool that takes packet sizes from MAWI HTTP uplink traffic and generates such packets as quickly as possible.

We measured the performance of a mux with four 10Gbps NICs installed: our mux can forward 36Gbps of HTTP uplink traffic, saturating all links (considering our encapsulation overheads at the mux) while using 7 of the 10 cores of the machine.

In the MAWI traces, server-to-client traffic is 15 times larger than client to server traffic, so one mux can load balance a pool of servers that together serve 540Gbps of downlink traffic. HTTP servers running custom made stacks can serve static content at 60Gbps [20]; however most servers will serve much less than that because content is dynamic. We expect one server to source around 1-10Gbps of traffic, and expect that a single software mux could cater for 50-500 servers.

Implementation overheads. We measure the server overhead introduced by our kernel module that decapsulates packets and implements daisy chaining. To this end we ran a 10Gbps iperf connection between a client and a Beamer server and measured its CPU usage with and without our kernel module. The vanilla server has an average CPU utilization of 7%, and of 9% with our module installed; this overhead is negligible in practice.

Latency. Our software mux achieves high throughput, but have we sacrificed packet latency in our pursuit of speed? We setup an experiment where our mux is running on a single core and processing 64B packets sent at different rates. In parallel, we run a ping with high frequency between two idle machines. The echo request packet passes through the mux, and the reply is sent directly to the source. We show a CDF of ping latency measurements for different packet rates in Figure 16. As long as the CPU is not fully utilized, both median and worst-case packet latencies stay below 0.2ms. When we overload the mux with 6.6Mpps (600Kpps more than its achievable throughput), the ping latency jumps to 1.5ms and 14% of packets are dropped. This latency is a worst case and is explained by the time it takes one core to process all the packets stored in the 10 receive queues used by netmap (one queue per core, 256 packets per queue).

P4 dataplane. We do not have access to a Tofino switch yet, so we resort to both software deployment and NetFPGA deployment to test our P4 prototype.

We first ran our P4 mux in the behavioural model switch on one of our Linux machines and measured its performance: the switch can only sustain 55Mbps of iperf throughput with 1500B packets, and around 4.5Kpps with minimum-sized packets. Any performance measurements with this switch are therefore irrelevant; we do, however, use it to check the correctness of our implementation and interoperability with our controller and the Click-based software mux.

Our NetFPGA implementation of the P4 switch uses P4-NetFPGA [2]. To enable our prototype to compile we had to make a number of modifications. First, we upgraded our code to P4-16 which simplified our code because actions can compute checksums, so we don't need to recirculate packets anymore. Next, running on hardware imposes constraints on table actions, limiting the bitsize of action parameters. To avoid these problems we broke up bigger tables into cascading smaller tables which satisfy the constraints. The decomposition is done such that we maintain consistency even if concurrent tables are not modified simultaneously.

We tested our implementation with Vivado's xsim 2016.4 simulator, injecting a batch of packets, verifying they are processed correctly, and measuring the time needed by the switch. The simulator reports that it takes 154 μs to process 10000 packets with 100B packets; this means the P4 mux can handle around 60Mpps. Deploying this prototype on actual hardware is ongoing work.

6.2 Scalability and robustness

Handling mux churn. One of the major benefits of software load balancing is the ability to add capacity when demand increases. This means setting up a new mux and sending a BGP announcement for the VIPs it serves. As soon as the announcement propagates to the border routers, they start hashing traffic to the new mux.

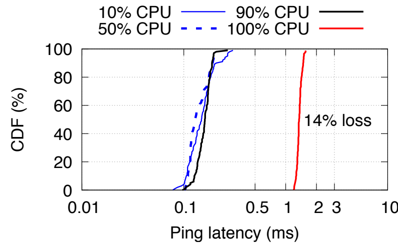


Figure 16: Mux latency is less than 0.3ms: when not fully loaded.

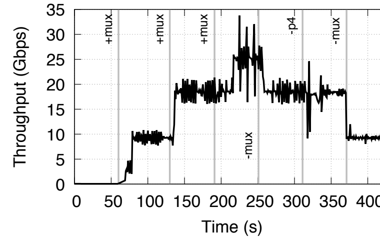


Figure 17: Beamer handles failures and mux churn smoothly.

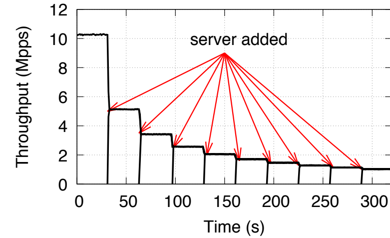


Figure 18: Beamer spreads traffic evenly across all active servers.

To emulate a real datacenter setup, we use an IBM 8264 RackSwitch as the border router and setup five clients and six servers, each with one 10Gbps interface. Clients open several `iperf` connections each to the VIP of the server, and we plot their added throughput in Figure 17. We begin the experiment with a P4 switch running alone and load balancing all traffic; the performance is terrible, just 55Mbps. Next, we add three software muxes a minute apart: the graph shows the few seconds it takes the muxes to setup a BGP session, announce the VIP, and to the router to install the route and start using it. Traffic scales organically as we add more muxes. Connections change muxes, yet they are not affected since our muxes are stateless. The P4 and Click muxes behave the same way, and are interchangeable.

We start simulating mux failures: first we kill a Click mux at 240s by bringing its network interface down, and throughput drops to 20Gbps. Next we kill the P4 mux: total throughput suffers until BGP discovers the failure and reconverges, then it recovers to 20Gbps. During the experiment not a single iperf connection is broken.

We conclude that Beamer handles mux churn smoothly, and that it is trivial to create a heterogeneous deployment with both software and hardware muxes.

Handling server churn. We want to see how Beamer balances server traffic when servers are added or removed. We generate 64B packets from a single machine and send them directly to one mux (using two cores). The experiments start with a single server receiving all traffic, and then we keep adding servers every 30 seconds using the controller's command-line interface. The controller moves buckets between servers to evenly balance the traffic across all servers by updating ZooKeeper data, as detailed in §4.5. When all the changes are ready, the controller "commits" the change by updating the current generation node, and the mux updates its state. Figure 18 shows the throughput received by each server as a function of time: changes are almost instantaneous, and traffic is evenly balanced across all active servers.

Connection affinity. We now use TCP clients to estimate the ability of Beamer and Stateful to provide connection affinity in different scenarios. In all experiments, we have 7 clients each open 100 persistent HTTP connections and continuously downloading 1MB files over each of them, in a loop, from servers in a Beamer cluster.

In our first experiment, we begin with two muxes and 8 servers and then perform a scale-down event: we first remove some servers, wait for 30s and then remove one mux. The number of broken connections is shown below.

DIPs removed	0	1	2	4
Stateful	0	54 ± 7	103 ± 14	214 ± 48
Beamer	0	0	0	0

As expected *Stateful* behaves poorly: it drops 7%-30% of the active connections when 1 to 4 servers are removed. After the DIP is removed, traffic is still sent correctly because there is state in both muxes. However, when one mux is removed, its state (for half of the connections hitting the removed DIP) is lost, and these will be hashed by the remaining mux to the other servers, which will reply with RST messages. In contrast, Beamer does not drop any connection because daisy chaining keeps forwarding packets to the removed DIP.

SYN flood. In our second experiment we use a similar setup, but only remove servers, keeping the mux set constant. In the absence of a SYN flood attack both Stateful and Beamer provide perfect connection affinity, as expected. We then started a SYN flood attack (1Mpps) running in parallel to our 7 clients; we show the number of dropped connections below:

DIPs removed	0	1	2	4
Stateful	0	87 ± 2	184 ± 8	351 ± 21
Beamer	0	0	0	0

Stateful performs rather poorly in this SYN flood attack, because the state its muxes keep for its real clients is flushed out by the aggressive attack. In contrast, Beamer's performance is not affected. Finally, we measured the effect of the SYN flood on the servers themselves, finding there was little impact: the average server utilization increased by 1% during the attack, and the flow completion times increased from 1ms to 1.3ms in the median and from 1.4ms to 1.7ms at the 99%.

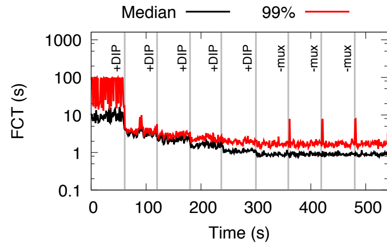


Figure 19: Flow completion times for MPTCP clients using Beamer

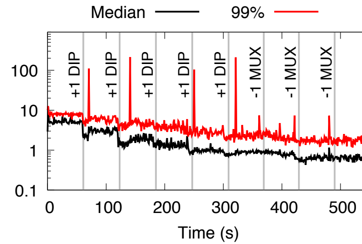


Figure 20: Flow completion times for MPTCP clients using Stateful.

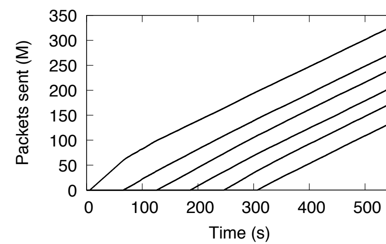


Figure 21: Outgoing traffic per server with Beamer

6.3 Load balancing HTTP over MPTCP

In our next experiment we emulate over-the-air mobile app updates. We setup four clients repeatedly downloading the same 100MB file using the `siege` tool from the VIP handled by Beamer. Each client opens 10 parallel downloads, and runs in a closed loop: as soon as one transfer finishes, it starts a new one. The clients run MPTCP and have two virtual interfaces on the same 10Gbps physical interface.

We start the experiment with four muxes and one server processing all the traffic. After every minute a new server is added until we reach six servers. Next, we start killing one mux every minute until there is a single mux running.

We plot the median and 99th percentile of client-measured flow completion as a function of time in figure 19. The graph shows that adding servers drastically reduces both the median flow completion time and especially the tail. The median drops from 10s when a single server is used, to 1s when six servers are used. The 99th percentile also drops from 50-100s with one server to a couple of seconds with six servers. Notice the ten second spikes in 99% FCT when muxes are killed: these are subflows that were handled by the failed mux, and they stall the entire MPTCP connection until they timeout and their packets are reinjected on good subflows.

Figure 21 shows the cumulative number of packets sent by each server and we can clearly see how initially only one server is active, another quickly follows and then more servers join one minute apart. The graph also shows that Beamer does a good job spreading connections equally across all active servers.

Finally, we wanted to check that daisy chaining works as described. We plot the number of daisy chained packets by each server in Figure 22. Note the different unit on the y axis (thousands of packets instead of millions): daisy chaining not only works, but it is also quite cheap. Daisy chaining forwards a total of 30 thousand packets, most of which are ACKs (total size is around 200KB).

Siege did not report any failed connections, but this could be masked by MPTCP's robustness to failures. We looked at individual subflow statistics and found that no subflows were reset; we did see numerous subflow timeouts triggered by mux failures, however these are well masked by MPTCP: when one subflow crossed a failed mux, its packets get resent on other working subflows.

We ran the same experiment with *Stateful* to test its behaviour when handling MPTCP connections and under mux failures. Packet traces show that, overall, less than 20% of MPTCP secondary subflows are created; this is expected, since *Stateful* is oblivious to MPTCP and randomly sends subflows to servers. With *Stateful*, MPTCP connections have a single subflow most times and behave like regular TCP. Without failures, FCTs should be similar to MPTCP/Beamer since the total achievable network capacity is the same. The FCT results in Figure 20 confirm our expectations: median FCTs are similar, however the 99th percentile is much higher. This is because MPTCP does a much better job of pooling the available network capacity than TCP does, thus reducing the outlier FCTs. Also note the huge spikes when DIPs are added or muxes are removed: this is Siege timing out on a connection after many retransmission attempts.

6.4 Controller scalability

Stable hashing works great as long as the centralized allocation of buckets to servers scales to large deployments. In this section we evaluate whether our controller can scale to many muxes and by extension to a large datacenter. To stress the controller we generate the maximum number of DIPs for a single VIP supported by our solution (64K), and create 100 buckets for each DIP, resulting in a total of 6.4 million buckets. We deploy our system in Amazon EC2 as follows: ZooKeeper runs on three VMs, one VM runs our controller, and one hundred VMs run our mux. According to our benchmarks in §6.1, one hundred muxes should easily be able to load balance traffic for 64K typical HTTP servers.

First, we want to see how long it takes to perform control plane operations on the maximum load balancer instance we can support. We have stress-tested the controller and run each operation multiple times with different numbers of pre-existing DIPs, randomized bucket-to-

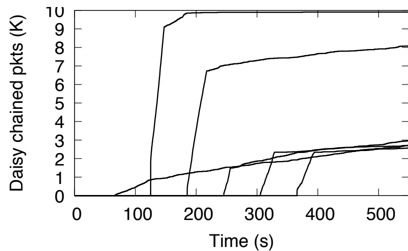


Figure 22: Packets daisy chained per server.

# DIPs	Add (sec)	Rm (sec)
1	0.63	0.58
10	0.57	0.57
100	0.69	0.67
640	0.87	1.58
6400	6.9	2.25
16000	8.1	3.2
32000	9.8	9.7

Figure 23: Duration of control plane ops on the largest Beamer instance

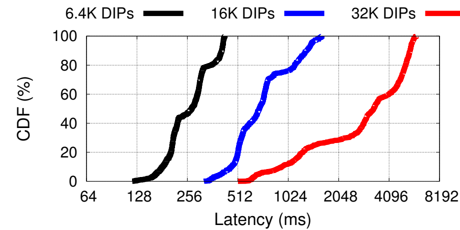


Figure 24: Time to propagate a controller update from ZooKeeper to all the muxes.

DIP assignments [2], and recorded the maximum completion time. The results are provided in table 23, showing that the Beamer controller performs large config changes in a few seconds.

Next, we measured the time it takes since the controller commits a new generation until all muxes download and start using it. Fig. 24 shows a CDF of the propagation time as measured at the muxes for three large config changes (adding 6.4K, 16K or 32K servers); the results show that even for 32K servers, all muxes use the new dataplane rules in a few seconds. Smaller updates are installed in tens to hundreds of milliseconds.

Finally, we note that all operations generated negligible amounts of control traffic; the largest ones (the addition or removal of 32K DIPs) incurred less than 1GB of traffic (10MB per mux).

7 Related work

Almost all existing solutions for datacenter load balancing keep per-flow state at muxes. Software solutions include Ananta [26], Maglev[9], IPVS[31] and GLB[15] while hardware ones include Duet [13] and SilkRoad [22]. Resilient hashing [6] takes a similar approach on switches and routers to avoid the pitfalls of ECMP. To allow scale in/out without affecting client traffic, stateful designs could use flow state migration, which is very expensive: OpenNF [14] or Split Merge [28] offer migration guarantees and strong consistency but at a steep performance cost (Kpps speeds).

A parallel effort to ours is Faild[3], a commercial stateless load balancer that works within a single L2 domain using ARP rewriting; this reduces its applicability to small clusters. Kablan et al.[17] propose to store per-flow state in a distributed key-value storage solution such as RAMCloud [25] instead of keeping it in memory; however its performance is limited to 4.6Mpps per box, eight times slower than Beamer.

[2] Updates to the dataplane are stored in a compressed format and having randomized bucket assignments yields near-worst-case compression ratios.

Load balancing within a single OpenFlow switch has been examined in [24, 32]. Orthogonal to existing load balancing solutions, Rubik[12] uses locality to reduce bandwidth overhead of load balancing while Niagara [18] offers an SDN-based solution to improve network-wide traffic splitting using few OpenFlow rules.

Paasch et. al [7] discuss the problems posed by MPTCP traffic to datacenter load balancers. Their analysis focuses on ensuring SYN(MPC) and SYN(JOIN) packets reach the same server, and it assumes muxes keep per flow state after the initial decision has been made. Duchene et. al [10] propose to load balance secondary MPTCP subflows by using IPv6 addresses; Beamer could easily implement this solution if both the client and the datacenter have IPv6 enabled and a working IPv6 path between them. Finally, Olteanu et al. propose [23] to load balance MPTCP traffic by encoding the server identifier in the TCP timestamp option; unfortunately this solution does not work if the client does not support or enable timestamps, and supports a smaller number of servers (8192) per VIP.

8 Conclusions

We have presented Beamer, the first stateless datacenter-scale load balancer solution that can handle both TCP and Multipath TCP traffic. Beamer muxes treat TCP and MPTCP traffic uniformly, allowing them to reach speeds of 6Mpps per core and 33Mpps per box, twice faster than the fastest existing TCP load balancer, Maglev [9]. A Beamer mux can saturate four 10Gbps NICs with real HTTP uplink traffic using just 7 cores.

Daisy chaining enables Beamer to provide connection affinity despite DIP and mux failure, removal and addition. In contrast to stateful designs, Beamer handles SYN flood attacks seamlessly.

Beamer is available as open-source software at `https://github.com/Beamer-LB`.

Acknowledgements

This work was partly funded by the SSICLOPS and SUPERFLUIDITY H2020 projects.

References

[1] zlib. https://www.zlib.net/.

[2] P4 to NetFPGA project. https://github.com/NetFPGA/P4-NetFPGA-public/wiki, February 2018.

[3] J. T. Araujo, L. Saino, L. Buytenhek, and R. Landa. Balancing on the edge: Transport affinity without network state. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA, 2018. USENIX Association.

[4] T. Barbette, C. Soldani, and L. Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, 2015. IEEE Computer Society.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, Jul 2014.

[6] Brad Matthews and Puneet Agarwal. Resilient Hashing for Load Balancing of Traffic Flows. US Patent Application: US20130003549 A1, Jan 2013.

[7] Christoph Paasch, Christoph and Greenway, G. and Ford, Alan. Multipath TCP behind Layer-4 loadbalancers (internet draft). https://tools.ietf.org/html/draft-paasch-mptcp-loadbalancer-00, Sep 2015.

[8] W. M. Eddy. Defenses Against TCP SYN Flooding Attacks. *The Internet Protocol Journal*, 9(4), Dec 2006.

[9] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, Mar 2016. USENIX Association.

[10] Fabien Duchene and Vladimir Olteanu and Olivier Bonaventure and Costin Raiciu and Alan Ford. Multipath TCP Load Balancing. https://tools.ietf.org/html/draft-duchene-mptcp-load-balancing-01, July 2017.

[11] Ford, Alan and Raiciu, Costin and Handley, Mark and Bonaventure, Olivier. RFC6824: TCP Extensions for Multipath Operation with Multiple Addresses. https://tools.ietf.org/html/rfc6824.

[12] R. Gandhi, Y. C. Hu, C.-K. Koh, H. Liu, and M. Zhang. Rubik: Unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *Usenix Annual Technical Conference*, 2015.

[13] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM*, 2014.

[14] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014.

[15] GitHub Engineering. Introducing the GitHub Load Balancer. https://githubengineering.com/introducing-glb/, September 2016.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[17] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, 2017. USENIX Association.

[18] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford. Efficient traffic splitting on commodity switches. In *CONEXT*, 2015.

[19] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[20] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, New York, NY, USA, 2014. ACM.

[21] MAWI Working Group Traffic Archive. http://mawi.wide.ad.jp/mawi/.

[22] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. ACM.

[23] V. Olteanu and C. Raiciu. Datacenter scale load balancing for multipath transport. In *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, pages 20–25, New York, NY, USA, 2016. ACM.

[24] V. A. Olteanu, F. Huici, and C. Raiciu. Lost in network address translation: Lessons from scaling the world's simplest middlebox. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '15, pages 19–24, New York, NY, USA, 2015. ACM.

[25] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[26] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[27] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *NSDI*, 2012.

[28] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, 2013.

[29] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. USENIX Annual Technical Conference*, 2012.

[30] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. Netw.*, 6(1):1–14, Feb 1998.

[31] The Linux Foundation. The IP Virtual Server Netfilter module for kernel 2.6. http://www.linuxvirtualserver.org/software/ipvs.html, February 2011.

[32] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *HotICE*, 2011.

Appendix

A1. Defragmentation

There is a three-way trade-off between load balancing, fragmentation and churn. Beamer prioritizes load balancing, ensuring near-perfect balancing at all times. This means that we can either end up with fragmented bucket ranges assigned to servers (which will increase dataplane matching costs, especially for hardware dataplanes such as P4) or move buckets to reduce fragmentation but create daisy chaining traffic in the meantime.

Defragmentation is therefore necessary to ensure servers get a contiguous range of buckets, as this will reduce the number of rules needed in the mux dataplane. Beamer implements an algorithm that reduces fragmentation progressively, while keeping daisy-chaining costs small. The algorithm has two parameters: a target fragmentation rate ($fr \geq 1$, target average number of rules per DIP), and $bmax$, the maximum number of buckets that can be moved per server, per iteration.

The defragmentation algorithm has two phases: it first selects a *target mapping* and then it iteratively moves towards this target. The target mapping is computed whenever the DIP set changes, and this triggers a second implementation phase that includes a number of iterations where at most $bmax$ buckets per server are moved in each iteration; iterations are spread in time (one iteration every 4 minutes). The second phase stops whenever the target defragmentation rate is reached. While heuristic, the algorithm performs really well in practice.

The target mapping is computed greedily: starting at bucket offset 0, the controller selects the server which can take over a contiguous range while causing the least amount of churn. After every step, the offset is then incremented past the newly-allocated range.

```
start = 0
while (start < #buckets) {
  select DIP A s.t. churn of assigning A
         at position start is minimized.
  target(A) = {start, start+#buckets(A)}
  start += #buckets(A)
}
```

During each iteration of the second stage (pseudocode below), the controller performs a subset of the reallocations prescribed by the target mapping. It performs no

Hashing algo.	Imbalance	Min. data plane rules	Server churn 1%	5%
Consistent[19]	2.27	9K	0	0
Maglev[9]	1.01	65K	2.3%	3.3%
Stable Hashing	1.01	1K	0	0

Table 1: Hashing comparison, N=1000 and B=65537

more than $bmax$ reallocations per DIP, while keeping the number of buckets constant for each DIP. (I.e. the number of buckets allocated to each DIP is the same as the number of buckets allocated away from it.)

```
let G = (V, E) be a directed multigraph, where
       vertices are DIPs and edges are bucket
       reassignments
for each DIP in V
  DIP.budget = bmax
while (G.has_cycles) {
  select cycle C
  for each realloc in C.edges {
    perform(realloc)
    E.remove(realloc)
  }
  for each DIP in C.vertices {
    DIP.budget -= 1
    if (DIP.budget == 0)
      V.remove(DIP)
  }
}
```

A.2 Stable hashing evaluation

Table 1 shows the performance of Stable hashing against our implementations of the classical consistent hashing algorithm [19] and Maglev [9]. We used 1000 servers and 65537 buckets (the params are from the Maglev paper, §5.3, for fair comparison). We first measured the *imbalance*—the ratio between the maximum and average load—finding that Maglev and Stable have near-perfect load balancing, while Consistent hashing places twice more load on one unlucky server.

We next computed the minimum number of range rules we need to use in a hardware data plane to perform matching: algorithms that assign consecutive buckets to the same server will utilize fewer rules and are better. This is the case for Stable, where we only need one rule per server; Maglev, in comparison, needs as many rules as buckets, two orders of magnitude more than Stable. Consistent falls somewhere in the middle. Finally, we looked at the number of "innocent" connections that are disrupted when we remove a number of nodes; both Consistent and Stable have no collateral damage, while Maglev breaks 2.3%-3.3% of ongoing connections.

A.3 Defragmentation evaluation

Stable hashing can enable very cheap hardware implementation with one rule per server, but this is only the case when all the buckets assigned to one DIP are contiguous. As servers come and go, even a perfect distribution can end up fragmenting the buckets, with each server in charge of many small ranges; this would require proportionally more rules to implement in hardware.

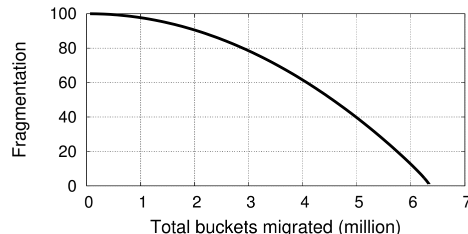


Figure 25: Defragmentating bucket-to-DIP assignments to reduce the number of data plane rules.

To avoid this effect, especially for hardware deployments, we can use the defragmentation described above: when the average number of rules per server increases beyond a given threshold, the defragmentation algorithm is invoked, reassigning buckets to remove fragmentation as described in §4.5.

We show a run of our defragmentation algorithm in Figure 25 starting from a worst case scenario where all buckets assigned to a DIP are scattered, and each DIP needs 100 rules to match its buckets. The figure shows how the fragmentation (number of rules needed per DIP) decreases as the algorithm migrates more buckets to reduce fragmentation.

The cost of defragmentation is daisy chaining, which is proportional to the number of buckets "moved" between servers. In the worst case when we move all buckets, Beamer will duplicate the incoming traffic for a brief period of time. To avoid creating congestion, the defragmentation algorithm moves slowly, migrating a few buckets at a time and then waiting for daisy chaining to end; this ensures that overall load increases only marginally.

We also note that defragmentation is only needed infrequently. We start with a fresh cluster containing 10K DIPs (1M buckets) and perform a number of control plane operations and show the resulting fragmentation in the table below. Even after large control plane operations, Fragmentation only increases slightly, and in some cases (like doubling or halving the cluster) it stays perfect. So in normal operation, we expect a cluster to slowly become more fragmented, reducing the need for defragmentation.

# servers	Added	Removed
0.1%	1.01	1.01
10%	2	2.1
33%	1.5	2
50%	1.33	1
100%	1	N/A

Larry: Practical Network Reconfigurability in the Data Center

Andromachi Chatzieleftheriou, Sergey Legtchenko, Hugh Williams, Antony Rowstron
Microsoft Research

Abstract

Modern data center (DC) applications require high cross-rack network bandwidth and ultra-low, predictable end-to-end latency. It is hard to meet these requirements in traditional DC networks where the bandwidth between a Top-of-Rack (ToR) switch and the rest of the DC is typically oversubscribed.

Larry is a network design that allows racks to dynamically adapt their bandwidth to the aggregation switches as a function of the traffic demand. Larry reconfigures the network topology to enable racks with high demand to use underutilized uplinks from their neighbors. Operating at the physical layer, Larry has a predictably low traffic forwarding overhead that is adapted to latency sensitive applications. Larry is effective even when deployed on a small set of racks (e.g., 4) because rack traffic demand is not correlated in many DC workloads. It can be deployed incrementally and transparently co-exist with existing non-reconfigurable racks. Our prototype uses a 40 Gbps electrical circuit switch we have built, with a simply local control plane. Using multiple workloads, we show that Larry improves tail latency by to 2.3x for the same network cost.

1 Introduction

There is a rapid adoption of high bandwidth networking in the DC. It is now common to deploy 40 Gbps to the server [59], and 50-100 Gbps is becoming popular [1]. An increasing number of applications are capable of consuming that bandwidth [22, 24, 34, 36, 42, 51] and require low and predictable latency [24, 41, 59]. Emerging techniques such as disaggregation of DRAM and non-volatile memory are also sensitive to latency and packet queuing [26]. This is a challenge because a large fraction of the traffic for these applications is not rack local [44, 59], and rack uplink bandwidth is typically oversubscribed [28, 47] which leads to rack-level congestion.

This paper presents Larry, a network design that addresses rack uplink congestion by dynamically adapting the aggregate uplink bandwidth of the rack to its traffic demand. For that, racks with congested uplinks use spare uplink bandwidth from physically adjacent racks. Larry targets workloads in which rack traffic is bursty and loosely correlated across racks, and we observe these properties in traces from our DCs.

Using local resources for traffic offloading has been first proposed by GRIN [18]. However, GRIN offloads traffic through multiple hops at layers 2 or 3. This typically adds 200-500 ns *per hop* even in cut-through mode and without queuing [6, 26, 60]. For some applications that require round trip times of a few microseconds [24, 26], this could represent a non-negligible latency overhead. In contrast, Larry is reconfigured at the physical layer, and only adds a predictable end-to-end forwarding overhead of a few nanoseconds. This *local reconfigurability* differentiates Larry from prior work on fully reconfigurable networks [23,25,27,29,30,39,40,43,50,54–56]. These systems typically redesign the *entire DC network*, making them efficient, but hard to deploy in practice, especially in existing data centers.

Larry uses a custom electrical circuit switch and exploits unused ports on the ToRs. Larry is designed for small-scale deployments of physically adjacent racks (e.g., 4 to 6) called *rack sets*. In a rack set, the ports used on *each* ToR to connect to the aggregation switches are instead connected to the circuit switch. The circuit switch is then connected to the set of aggregation switches that would have been connected to the ToRs. Any non-cabled ToR ports are also connected to the circuit switch. This does not change the number of uplinks to the aggregation switches, but creates a network reconfigurable at the physical layer and local to the rack set. Once configured, traffic between ToRs and aggregation switches is transparently forwarded by the circuit switches at the physical layer. Within the rack set, a rack with high traffic demand can hence forward its traffic

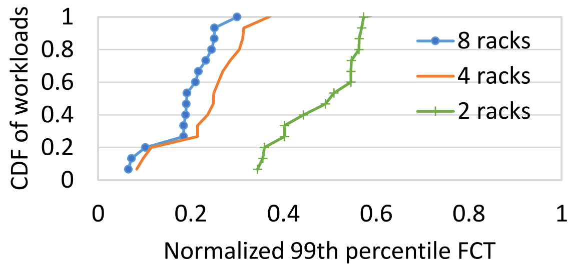


Figure 1: Impact of local reconfigurability on tail FCT.

through underutilized uplinks on other racks. The properties of our design are the following:

Incremental Deployment. The smallest unit of deployment is the rack set. Therefore, if a cloud provider suffers from congestion in a specific service (e.g., storage racks), local reconfiguration can be deployed only on the racks used for that service.

Transparency. Larry works with the DC network in place: any existing global controllers see links failing and being established, as reconfiguration occurs at the physical layer. No reconfiguration-specific routing state is needed outside the rack set and reconfiguration events are only visible to ToR and aggregation switches. We use off-the-shelf ToR and aggregation switches, and no software or hardware changes are required on the servers to use the extra uplink bandwidth.

Cost efficiency. All the racks in the rack set are physically close to each other, which allows, even at 100 Gbps, to use cheaper passive cables for all additional connectivity. No extra connectivity is required in the rest of the DC network. Further, our circuit switch design uses simple, commercially available crosspoint ASICs [7].

Low latency. The crosspoint ASIC forwards the incoming signal from the high-speed serial data link without processing any packets, hence the forwarding latency of the circuit switch is within a few nanoseconds [7].

Our evaluation shows that uplink reconfiguration can be done with low overhead and augmenting a traditional oversubscribed topology with Larry increases the performance per dollar by up to 2.3x.

The rest of the paper is organized as follows. Section 2 describes the benefits and challenges of local reconfigurability. Section 3 details the design of Larry that implements local reconfigurability at the physical layer. Section 4 discusses the practicality of our approach. Section 5 evaluates our design. Finally, Section 6 presents the related work and Section 7 concludes.

2 Local Reconfigurability

We now describe the benefits and challenges of local reconfigurability.

2.1 Is Local Reconfigurability Useful?

Local reconfigurability reduces rack uplink congestion by using underutilized uplink bandwidth from a small set of adjacent racks. We now show that some key DC workloads do exhibit low traffic correlation across racks and can benefit from local reconfigurability. For our analysis, we assume that any uplink can be used by any rack in a rack set composed of m physically adjacent racks. We also assume that the core network is fully provisioned. These optimistic assumptions allow to estimate an upper bound on the performance of local reconfigurability and will be refined in latter sections.

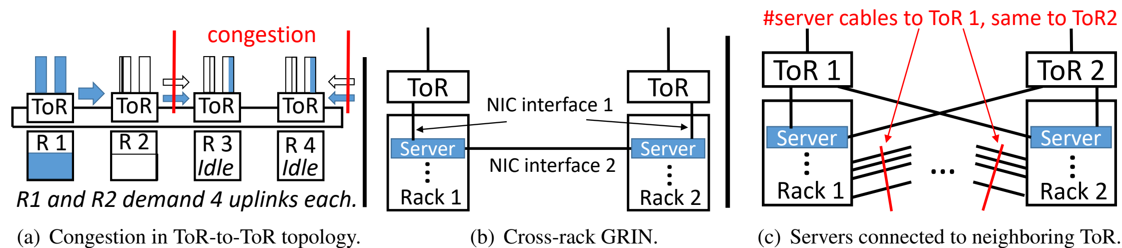
We use two DC traces and show that they exhibit low rack traffic correlation. The first trace contains all files accessed by a large-scale cloud service over a week in mid-2016. The traces do not differentiate between local accesses to disk and remote accesses over the network. To measure the impact of data transfers on the network, we simulate a small local write-back cache at the compute node to which all accesses are made. Files that have not been accessed for more than a day are de-staged to the DC storage tier over the network. On a cache miss, a file is read from the storage tier over the network. We group the network transfers by rack, then form groups of eight randomly selected racks[1]. For every group of racks, we replay all the network transfers between the storage tier and the cache in a flow-based simulator that computes the flow completion times (FCT) assuming max-min fairness bandwidth allocation and one flow per file access.

In the simulated topology, each rack is provisioned with u uplinks, such that a rack set with m racks has $u \times m$ uplinks. In the rack set, at any point in time, each rack gets a subset of the uplinks that is proportional to its demand. We compute the tail FCT for each sub-trace for $u = 4$ and $m = 1, 2, 4, 8$ and use $m = 1$ (all racks are independent) as a baseline. Figure 1 shows the CDF of the 99th percentile FCT for all the simulated sub-traces for different rack set sizes, normalized to the baseline. We can see that tail FCT is improved for *all* rack set sizes, showing that local reconfigurability reduces congestion. The results also show that for this workload, there is only a marginal benefit of having rack sets larger than 4 racks.

The second trace was obtained from the authors of [27], and covers four clusters from a large cloud provider. As described in [27], the clusters have between 100 and 2,500 racks and run a mix of workloads. The results qualitatively show a similar trend: when operating at a scale of 4 to 8 racks, local reconfigurability can reduce most of the uplink bandwidth congestion during peak demand.

Notably, in the first trace, demand is not correlated de-

[1] The trace lacks rack placement information.



(a) Congestion in ToR-to-ToR topology.
(b) Cross-rack GRIN.
(c) Servers connected to neighboring ToR.

Figure 2: Challenges of implementing local reconfigurability at layers 2 or 3.

spite the fact that all racks host the same workload (storage). This suggests that rack traffic decorrelation is a property of the workload. We hence expect the analysis to hold if racks were chosen according to their physical proximity. Other studies of workloads [28, 40, 44] show the skew at rack-level. In the public cloud, services and applications are used by a large number of clients with decorrelated demand, and services often use several generations of rack hardware. We expect that local reconfigurability can be beneficial to a number of key workloads in that context.

2.2 Design Challenges

Several network designs can enable local reconfigurability. This section discusses the challenges of implementing local reconfigurability at layer 2, thereby motivating the use of layer 1 circuit switching. The number of aggregations switch ports connected to the core network *per rack* is the same for all described designs.

Local reconfigurability can be implemented at layer 2 by enabling non-shortest path routing between ToR and aggregation switches. This can be done by either adding extra packet switches between ToRs and the aggregation layer, or by interconnecting spare ToR ports in a multi-hop topology. Figure 2(a) shows one simple example of the latter, with four ToRs interconnected into a ring topology[2]. A Software-Defined Network (SDN) controller monitors the uplink bandwidth usage and balances traffic in the rack set. However, this introduces extra forwarding latency as store-and-forward packet switching adds approximately 1 μs per hop. Cut-through packet forwarding reduces the latency down to 200 to 500 ns per hop [6, 26, 60] but is not supported by all switches. Gao et al [26] describe emergent applications with a 3μs round trip latency budget, for which extra packet forwarding could increase round trip latency by 30% or more. In addition, for ToR-to-ToR topologies, there will be worst case scenarios with fate sharing of the ToR-to-ToR links leading to congestion. In Figure 2(a), racks have two uplinks, but each of 1 and 2 demands 4 uplinks. These racks hence forward half of their demand to the idle racks 3 and 4. This causes congestion on the ToR-to-ToR links such that 1 and 2 cannot use all the available uplink bandwidth. Finally, this increases the unpredictability of the network, as the latency of a flow depends on the uplink through which it is forwarded, and its throughput is impacted by flows from other racks on the ToR-to-ToR links. In the control plane, this requires custom routing, fine-grain bandwidth management and consistent updates to sets of SDN-enabled switches. This increases the probability of software bugs in the controller [46] and reconfiguration latency [31].

GRIN [18] implements local reconfigurability *within* a rack by interconnecting spare ports on servers' NICs and allowing a busy server to forward traffic through idle servers in the rack. This approach can be extended to enable local reconfigurability across racks by interconnecting servers on neighboring racks as shown in Figure 2(b). In addition to the extra hop latency, the flexibility is limited by the number of spare ports per NIC. At 40 Gbps per port and beyond, NICs with more than two ports (or multiple NICs per server) are not common, which limits the approach to *pairs* of servers. Packet forwarding and traffic prioritization between NIC ports need to be done in hardware, which is not supported by all NICs.

Finally, avoiding extra latency is possible with extra cables and ports at ToR or aggregation switches. For example, spare NIC ports on the servers can be connected to neighboring ToRs, as proposed by Liu et al [40] (see Figure 2(c)). However, in that case, the number of ports required for server connectivity increases by at least a factor of p (if p-port NICs are used) and it is unlikely that ToRs have enough spare ports. Alternatively, all spare ports on the ToRs can be connected directly to aggregation switches, which either requires additional aggregation switches and extra optical cables, or increased oversubscription at the aggregation layer.

3 Overview of Larry

We enable low, predictable forwarding latency by ensuring that any extra forwarding is done at the physical

[2] Alternative topologies are also possible.

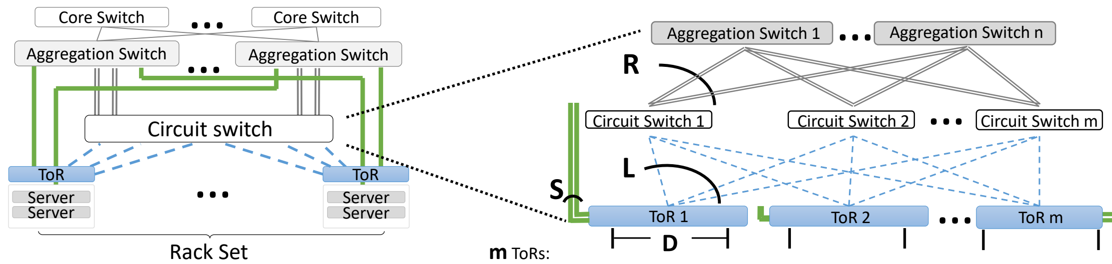


Figure 3: Architecture overview.

layer. We achieve this by using a *reconfigurable fabric* at the rack set that operates at the physical layer and allows uplinks to be *migrated* from one rack to another. The fabric enables racks to adapt their uplink bandwidth while appearing as a collection of independent racks to the rest of the DC. Uplinks are used to forward all the traffic from the ToRs to the aggregation switches. When a ToR is experiencing congestion to the core network, a local controller can physically migrate uplinks to that ToR from a non-congested ToR in the rack set. For example, the ToR could have 8 uplinks, and then have 2 additional uplinks migrated to it. The packet-switched network literally observes the two uplinks as disconnecting, and then connecting to the new ToR. We use Equal-Cost Multi-Path (ECMP) routing on the ToRs and aggregation switches to balance traffic across uplinks [33]. This is done at the flow granularity, such that for each packet, the switches hash the TCP five tuple to determine the port for the next hop. Uplink migration is enabled by deploying a custom low-cost low-radix electrical circuit switch in each rack to complement the existing ToR packet switches.

3.1 The circuit switch

An electrical circuit switch forwards the *electrical signal* received on one port through a circuit established to another port on the switch. The switching cannot be done *per packet*: once established, a circuit is expected to exist for hundreds of milliseconds or longer. The circuit switch does no packet header inspection or buffering, as a result the latency of transferring the signal from one port to another is on the order of 2 ns [7]. This is low enough to be transparent to the upper layers of the network. The packet switches connected to the circuit switch are unaware of its presence, and perceive the link as being a cable. A relatively simple hardware design using crosspoint ASICs [2, 7, 10] available today can have up to 48 100 Gbps ports. Section 5.1 describes the design of a prototype switch we have built.

Cost. The hardware architecture of crosspoint ASICs is typically simpler than merchant silicon. Crosspoint ASICs have no packet buffers or packet processing logic. In addition to that, because they do not need to process packets, the incoming signal from the high-speed serial data link does not need to be de-multiplexed to lower speed lanes, as done in packet switches. Crosspoint ASICs hence do not require SerDes at the inputs and outputs[3] which are typically challenging to design at high speed rates [15]. The switch core itself is hence simpler and switching is done at the line rate. In equivalent manufacturing processes, crosspoint ASICs should take less die area and have a lower power consumption for the same number of I/O lanes.

While the cost of the switch depends on the cost of its ASIC, it is harder to do a fair comparison of packet- and circuit- switches due to the cost of the other components, the difference of production volume, etc. Section 5.2.3 includes a sensitivity analysis on cost.

3.2 The reconfigurable fabric

Figure 3 shows the topology of a reconfigurable fabric that connects a set of racks to the aggregation switches. Each rack contains a set of servers connected to an unmodified ToR. ToRs and aggregation switches operate at layer 2 and above. For simplicity, in this section we assume that the reconfiguration is done by a single large circuit switch, as depicted on the left-hand side of Figure 3. In practice using a single circuit switch has scaling and deployment issues, and we distribute the circuit switch using one small-radix circuit switch per rack. The circuit switches are interconnected as shown on the right-hand side of Figure 3. We will describe this in more detail in the next section.

In a traditional network design, a ToR is connected to all the servers in the rack and has several (e.g., 8) uplinks connected to the aggregation switches. In our design, only a fraction of these uplinks is directly connected to the aggregation switches. We keep at least two (full green lines in Figure 3) to ensure the rack is still connected to the core if an aggregation switch fails while no uplinks are available through the circuit switch. The

[3] A SerDes is a pair of functional blocks that convert data between a fast serial link and slower parallel interfaces in each direction.

remaining ToR ports (e.g., 6), that would have been connected to aggregation switches are instead connected to the circuit switch. For each of these, a port on the circuit switch is then connected to the aggregation switch. Finally, any unused ports on each ToR are also connected to the circuit switch. Unused ports typically exist for topological reasons (e.g. due to oversubscription or mismatch between the number of servers per rack and the switch radix). The links from the circuit switch to the aggregation switches are active (optical) cables, and are shown as gray double lines in Figure 3. All the links from the ToR to the circuit switch use passive (copper) cables (dashed blue in Figure 3). At current price points, this reduces the cabling costs: at 100 Gbps, the transceivers account for a large part of the cable cost. While transceiver costs are currently dropping, passive cables are cheaper than the active ones [45]. At 100 Gbps, passive cables are available up to 5 m [16], which is long enough for rack sets of at least 4 racks. As the data rate increases, the loss of the passive cable is typically reduced by increasing the thickness of the cable. We have successfully used our 40 Gbps circuit switch prototype with long passive cables to interconnect a row of 4 racks. An important aspect of our design is that all the extra connectivity is done at lower cost using spare ToR ports, passive cables and circuit switch ports. The number of aggregation switch ports and active cables used in our design is the same as in a traditional topology with the same level of oversubscription.

The circuit switch can connect any of its uplinks to the aggregation switches to any ToR in the rack set. On the circuit switch, the number of uplinks is lower than the number of links to the ToRs. Therefore, a fraction of the ToR ports will not be connected by a circuit to an aggregation switch port at a given point in time. Such *disconnected* links have no PHY established and are seen as disconnected ports by the ToR. These links are used by the ToR to get extra bandwidth on demand. Consider a scenario where each rack has on average 8 uplinks to the aggregation switches, and this is not sufficient to satisfy the demand on one rack. With the reconfigurable fabric, any links that are underutilized on other racks in the rack set can be migrated to the rack experiencing high demand. The number of extra uplinks that can be migrated is a function of demand across the racks, and is at most the number of extra unused ports connected to the circuit switch from each ToR. More formally, if each rack has on average U uplinks to the aggregation switches, a ToR with S directly connected uplinks and L links to the circuit switch will be able to have between S and $S+L$ uplinks ($S \leq U \leq S+L$). The uplink bandwidth allocation on each ToR is managed by a local controller that monitors the set of ToRs in the rack set and reconfigures the circuit switches through a separate control plane. We will describe this in Section 3.4.

Physical layer reconfiguration enables low forwarding latency on all uplinks regardless of their physical location in the rack set. It also improves performance and predictability: there is no fate sharing of uplinks by multiple racks in the rack set. So far, we considered for simplicity that each rack set had a single large circuit switch. However, this is unpractical as even at small scales (e.g., 4 racks), the number of reconfigurable fabric ports exceeds the port count of the largest electrical circuit switch capable of 100 Gbps. This makes the cost prohibitive. Further, using a single circuit switch per rack set introduces a single point of failure.

3.3 Distributing the circuit switch

To simplify deployment and management, we use one circuit switch per rack. Each ToR needs to be attached to *all* the circuit switches in the rack set. The links to aggregation switches are distributed across the circuit switches, ideally with (at least) one link to each aggregation switch per circuit switch (see Figure 3 right hand side). This reduces the flexibility compared to a single large circuit switch: some pairs of ports cannot be connected by a circuit. For example, a port attached to one circuit switch cannot have a circuit to a port on another circuit switch. *Is this flexible enough to operate efficiently?*

To answer this, the key insight is that allowing any pairs of ports to be connected provides more flexibility than it is necessary for our design. The reconfigurable fabric can be represented as a bipartite graph in which vertices are ToR and aggregation switch ports and edges are circuits established between them. As the traffic demand changes, the circuit switch instantiates the bipartite graph that is the best adapted to the demand. Intuitively, since the graph is bipartite, there is no need to guarantee that *any* two ports can be connected, but only that each ToR has the required number of uplinks.

To show that the latter can always be achieved, we denote L the number of links from each ToR to the circuit switches and R the number of links from each circuit switch to the aggregation switches (see right hand side of Figure 3). We have $L \geq m$ because each ToR is connected to every circuit switch and there is one per rack. In addition to that, as described in the previous section, on each circuit switch the number of links to the ToRs is equal or higher to the number of links to the aggregation switches, i.e. $L \geq R$. Intuitively, these constraints over L mean that: *(i)* an uplink on a circuit switch can be connected to *any* ToR in the rack set and *(ii)* there are enough ports to ToRs on each circuit switch to allow all uplinks to be connected at the same time.

Therefore, for any uplink assignment in the rack set,

there exist a circuit configuration on the circuit switches that will instantiate the assignment. Furthermore, this configuration can be easily found. Intuitively, if each circuit switch is associated with a color, this problem can be expressed as an edge f-coloring of the bipartite graph [57]. The f-coloring problem is NP-complete in general, however f-coloring of *bipartite* graphs can be done in polynomial time [57]. It means that our set of circuit switches can instantiate any uplink assignment. In the evaluation, we show that our fabric achieves the performance of a single large circuit switch.

The number of ports required on each ToR and circuit switch is shown in Figure 3 (right hand side). On a ToR, our design needs $D+S+L$ ports: D to servers in the rack, S directly attached to aggregation switches and L to the circuit switches. On a circuit switch, $L+R$ ports are required: L to the ToRs in the rack set and R to aggregation switches. For example, assuming both 32-port ToR and circuit switches, $D=16$ ports to servers and $S=2$ static uplinks per ToR, our design can scale to $m=14$ racks per rack set. In this case, the limiting factor is the number of ports on the 32-port ToRs, because half of the ToR ports are used for in-rack connectivity. In practice, assuming a standard hot-/cold-aisle DC layout, we are limited to 4-6 racks by 5 m passive cables. For the rest of the paper, we conservatively consider rack sets with 4 racks only.

3.4 Controller

Each rack set has an independent lightweight controller that monitors the network load within the rack set, decides when to migrate the uplinks and manages the reconfiguration. The controller and circuit switches communicate through a control plane. In the prototype, we use a pre-existing management switch connected to board management controllers in the rack [14]. The controller can specify a new port mapping for each circuit switch, and read out the current port mapping. To ease deployment, the controller is designed as a soft-state process. When started, it is provided with a rack set configuration that describes the racks, circuit switches, ToRs and aggregation switches associated with the rack set. It then reads the current mapping from each circuit switch.

A key property of this controller is that it is *local*: it requires no global information from the core network. It only relies on the information that comes from the ToRs in the rack set, not even from the aggregation switches. We assume that ToRs enable a mechanism to query per port traffic statistics, e.g., OpenFlow [12]. This design simplifies deployment and reduces the impact on any existing global SDN controllers used in the core data center network. We assume that the core network is configured such that any flow-granularity traffic management is orthogonal to our design, but we do need the core network to use a mechanism, e.g., ECMP, to efficiently spread the network load across all possible paths to a rack[4].

When the controller migrates an uplink, there are two approaches to handling this from the perspective of the core network. The clean approach is to ensure no packet loss, which can be done by signaling to the global controller that the link will fail before it is remapped. The global controller then removes all routes that currently use the link. After that, the circuit switches are reconfigured and the PHYs are established over the new circuits. We assume that the ToRs can report physical layer changes to the rack set controller. When the new link is established, it is reported to the global controller that starts to route traffic over it. This approach can be for example easily implemented using off-the-shelf OpenFlow-enabled ToRs and aggregation switches [4,9].

The dirty approach is to allow rack set controllers to operate independently without communicating with the global controller. This relies on the existing core network monitoring and management mechanisms to see the reconfiguration as a link failure followed by a link repair. However, this can lead to a small number of packets being lost on the failed links. The reconfiguration typically occurs when the link is not highly utilized. In the evaluation, we show that reconfiguration delays are low, and links are reconfigured infrequently, suggesting that this approach will not cause significant traffic disruption.

The goal of the controller is to migrate uplinks to decongest ToRs with high demand. We do not modify the end-systems, and the controller has no global visibility, so it has no understanding of potential future demand for either egress or ingress traffic to the racks. We therefore use a greedy algorithm that periodically classifies each ToR in the rack set as being underutilized or potentially congested. To do this, the controller uses per-port byte count metrics obtained from the ToRs. This can be done at a fine-granularity: our prototype shows that if the controller is managing a rack set with 4 racks, polling all 4 ToRs every millisecond would generate only in the order of 10 Mbps of traffic.

The algorithm takes uplinks from ToRs which are not being utilized, and migrates them to ToRs that could potentially use more bandwidth. The greedy algorithm computes the aggregate uplink utilization for each ToR, in terms of ingress and egress bandwidth. It then determines the number of links from the ToR to the aggregation switches required to support that demand, considering a link fully utilized at 85% of its nominal capacity. If it determines that a ToR could support the workload with one or more links fewer than it currently has, the ToR is marked as underutilized. If all the links to the aggregation switches are fully utilized, it is marked as congested.

[4]For example by using groups of type `select` in OpenFlow 1.3 [12, 53].

The controller knows the full topology of the rack set and has a list of the circuit switches. For each circuit switch, it greedily assigns an uplink from the most underutilized ToR to the most overloaded ToR, records the corresponding circuit configuration and re-computes the link utilization. This happens until no more uplinks can be reassigned, because either there are no more underutilized uplinks, or all overloaded ToRs reached their maximum number of uplinks.

The controller is currently configured to be conservative to minimize the impact of reconfigurations on performance. If all ToRs in the rack set are underutilized no uplinks are migrated. If no ToRs are underutilized, no uplinks are migrated even if other ToRs are congested except if congestion is detected on a ToR that has less than its fair share of the aggregation links attached to the circuit switch. If so, we allocate the ToR the fair share of links. This ensures that all ToRs get at least their baseline uplink bandwidth under congestions across multiple racks. The output of the algorithm is a circuit configuration for each circuit switch that instantiates the new uplink assignment.

4 Discussion

This section discusses the feasibility of deploying Larry in production DCs. We first focus on the features that facilitate deployment:

Good failure resilience. A failure in one rack set does not lead to failures in other rack sets or elsewhere in the core network. The failure of the controller or any circuit switch does not disconnect racks from the core network as racks have direct connections to aggregation switches and controller failures do not lead to inconsistent network state. The rack set configuration state is stored in persistent storage and the internal local state is soft. We assume that a data center-wide service can monitor the liveness of the controller, and restart it upon failure.

Transparency. The reconfiguration is scoped within a rack set and is transparent to the end-systems. At the physical layer, PHY loss and establishment events only occur on the ToRs and aggregation switches. These events are managed by a local rack set controller that updates the mapping of links between the aggregation switches and the rack set ToRs. Larry can operate without modifications to the end-system operating system or applications, or the firmware or hardware design of the ToR or aggregation switches.

Support for incremental deployment. Larry does not require any changes to the core network management or operation. Existing cabling from the aggregation switches to the racks can be used. Within a data center, some sets of racks can have local reconfigurability provisioned (e.g., storage racks), while other racks can be deployed without it. It is even possible, to retrospectively fit this to deployed racks if needed.

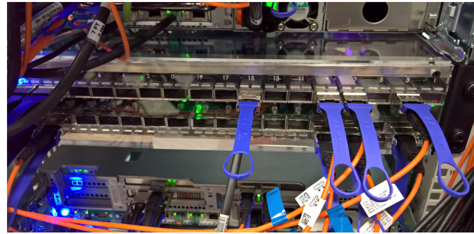


Figure 4: Prototype circuit switch.

Ease of deployment. State of the art reconfigurable topologies can be hard to deploy and operate [48]. In contrast, the sensitivity of Larry to environmental factors is negligible, and no specific operator experience is required. Each rack needs a 1U slot for a production version of the circuit switch.

Deployment Challenges. Larry requires extra cabling across racks, which complicates rack provisioning. We ensure that the additional cabling complexity is limited. The cabling is scoped to a rack set and is symmetric: all racks have the same number of extra cables to the other racks in the rack set. Additional cables originate on the ToRs and target the circuit switch. With a rack set size of m it is feasible to have just m extra cables per rack, where each cable carries multiple lanes similar to original 100 Gbps cables that bonded ten 10 Gbps lanes.

The reconfigurability of the network in the rack set can increase the link churn and create topological asymmetry. Link churn and asymmetry exist in DCs today, and prior work has already been done on efficient load balancing and neighbor discovery mechanisms to address the associated challenges [20, 47]. Larry also increases the routing state update rate on the switches. While reconfiguration in the rack set is expected to be infrequent (see Section 5.2.4), this could induce overhead on core network (T2) switches. Finally, the performance of Larry can be improved by rethinking existing DC components. In particular, the PHY negotiation mechanism on the ToR and aggregation switches should be tuned to minimize the link downtime.

5 Evaluation

Our evaluation aims to explore the benefits and overheads of deploying our design in a data center. We aim to answer the following questions: *(i)* What are the overheads of reconfiguring the physical link? *(ii)* How does Larry compare to static DC topologies with respect to performance metrics and cost efficiency? and *(iii)* What are the properties of the reconfigurable fabric?

For that, we first evaluate reconfiguration overheads

using a prototype circuit switch. Then, we explore the properties of our design with data center workload traces using discrete event simulation. Overall, our results show that local reconfigurability improves the performance per dollar of the network. The reconfiguration is infrequent, mainly adapting the network to macro changes in the workload and reconfiguration overheads are low.

5.1 Prototype

To demonstrate the viability of our design, we have built a prototype circuit switch (see Figure 4). The circuit switch has a 2U form factor with 40 front-facing QSFP+ ports. We use the M21605 asynchronous fully non-blocking crosspoint switch ASIC from MACOM [7]. The ASIC supports up to 160 lanes at 10 Gbps per lane and can connect any two lanes with an internal circuit. Each QSFP+ port is internally connected to four lanes on the crosspoint ASIC enabling 40 Gbps per port. The crosspoint ASIC is controlled by custom firmware on an ARM Cortex-M3 micro-controller on the switch. The micro-controller has an Ethernet link to an external control plane and a parallel interface to the ASIC.

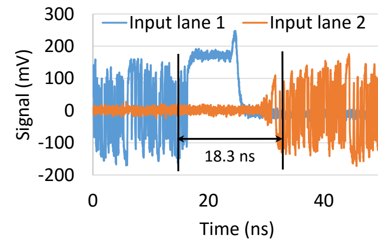
We emulate the egress/ingress traffic between a ToR and its aggregation using two Arista 7050X packet switches with 32 40 Gbps ports [3]. The packet switches are both attached to the circuit switch and to each other using passive copper cables. For traffic generation, we connect 2 servers per packet switch. Each server has a Mellanox ConnectX-3 40 Gbps NIC [5] and a dual Intel Xeon E5-2660 v3 CPU at 2.6 GHz running Windows Server 2016. All switches are connected to an Ethernet control plane and controlled by a separate server. The packet switches support OpenFlow and we use the Floodlight OpenFlow controller to reconfigure the routing state of the switches during reconfiguration [13].

When we designed the switch, 100 Gbps network components were not widely available. Today, this can be implemented using for example the MAXP-37161 crosspoint ASIC that supports 25 Gbps per lane [8]. A 100 Gbps zQSFP+ port [17] uses 4 lanes so it is possible to build a circuit switch with 16 ports at 100 Gbps by using four MAXP-37161 ASICs in parallel and routing each lane of a zQSFP+ connector to a separate crosspoint ASIC then back to another zQSFP+ connector. While 16 ports are enough to implement all the topologies described in the evaluation, there exist up to 48-port crosspoint ASICs operating at the same bandwidth per lane.

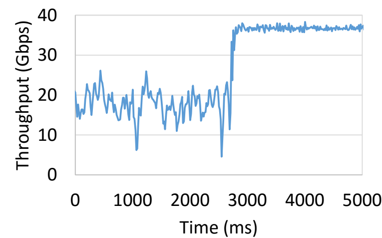
5.1.1 Micro-benchmarks

We evaluate the reconfiguration overheads by measuring the circuit switching time and its impact on throughput.

Switching time: We measure the time taken by the crosspoint ASIC to establish a circuit. For that, we connect a high frequency oscilloscope to two spare ports on the circuit switch. The oscilloscope measures the voltage output on lane 0 of each port. We use one of the ports attached to a packet switch as a traffic generator. We first set up a circuit between the packet switch and the first port of the oscilloscope. We then connect the packet switch to the second port and measure the time taken by the signal to appear on the second port. Figure 5(a) shows the voltage generated by the signal over time on both ports during reconfiguration. Initially, the signal appears on the first port while the second port is idle. After a transition period of 18.3 ns on average, the signal appears on port 2, while port 1 becomes idle. We ran the experiment 10 times and observed a reconfiguration delay of 19.5 ns in the worst case, with a median of 18.27 ns. This is lower by about an order of magnitude or more compared to other circuit switching proposals for DC networks [27, 50] because electrical circuit switching requires no physical movement in the switch.



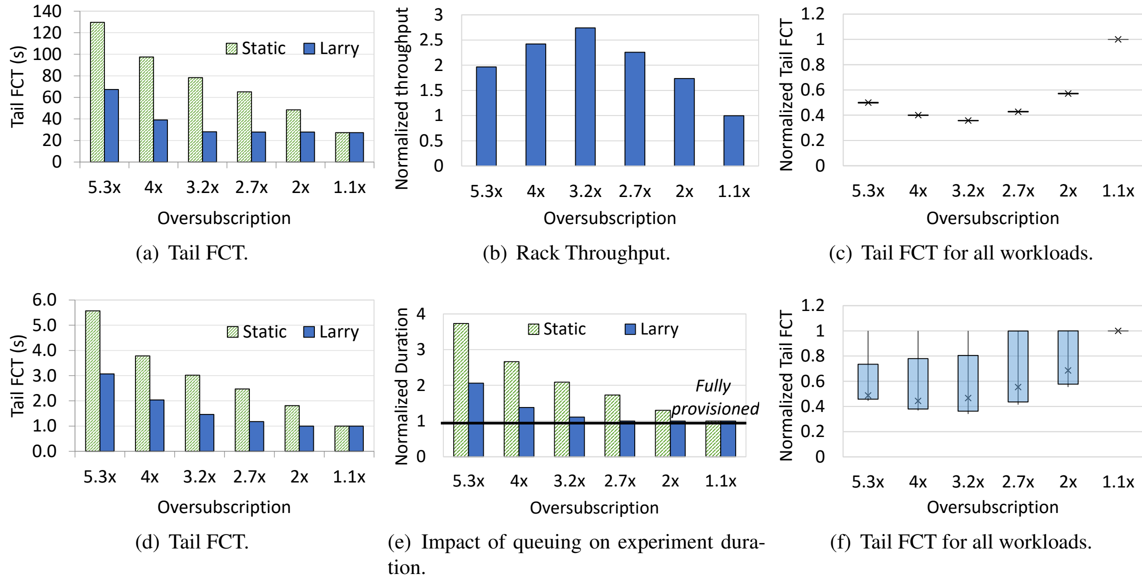
(a) Signal under reconfiguration.



(b) TCP throughput on reconfiguration.

Figure 5: Prototype micro-benchmarks.

Throughput: We now measure the impact of adding uplinks on application throughput. We form two source-destination pairs using the four servers such that traffic for both pairs must traverse both packet switches. We use `NTttcp` [11] to saturate the NIC bandwidth by creating 20 TCP flows, one per core, from each source to its destination and measure the throughput at the destination with a 15 ms interval. Initially, there are no circuits on the circuit switch, so packet switches can only use one 40 Gbps link for all traffic. The link is fair-shared across both source-destination pairs. We then set up a circuit that creates an additional link between the packet switches and configure them such that each source desti-



(a) Tail FCT. (b) Rack Throughput. (c) Tail FCT for all workloads.

(d) Tail FCT. (e) Impact of queuing on experiment duration. (f) Tail FCT for all workloads.

Figure 6: Base performance with the Storage (top) and Production (bottom) workloads.

nation pair uses its own link. Figure 5(b) shows the receive throughput on one receiver over time. We can see that the application is able to benefit from reconfiguration and doubles its average throughput. This is achieved transparently without modifications to packet switches or servers. The reconfigurable uplink is similar to a physical cable: we observe no packet loss on any of the links, and both uplinks achieve the same throughput.

5.2 Simulations

We now demonstrate that deploying our design in existing DCs offers performance and cost benefits. At the high level, we aim to answer the question: *what is the benefit of adding Larry to a static topology?*

To scale our evaluation to our traces, we use a simple flow-level simulator that represents each TCP flow by a single flow in the topology graph. We obtained the simulator that has been used in [21, 37, 38]. It has been cross-validated with real hardware in the context of storage [21] and offers a good trade-off between scalability and accuracy. Each flow is routed through one randomly chosen shortest path between the source and the destination. Network components, including ToRs, circuit switches and servers, are represented as vertices, while edges simulate links. Each link is bidirectional, has a bandwidth and keeps track of all the flows traversing it at any time. The simulator computes the bandwidth dedicated to each flow using max-min fairness. On flow creation or completion, the throughputs of all impacted flows are recomputed. Overall, this simulates the high-level behavior of a typical TCP network with ECMP routing. During reconfiguration, the topology graph is updated, and all flow throughputs are recomputed. Unless otherwise stated we do not simulate the downtime on links during reconfiguration as it typically occurs only a few times per hour (see Section 5.2.4).

Topologies. Our baseline topology, denoted as *static*, has m racks directly attached to aggregation switches. The performance of the static topology depends on the oversubscription at the ToRs. For Larry, we augment the rack with a circuit switch and interconnect the circuit and packet switches as described in Section 3. For both topologies, we use 32-port 100 Gbps ToR and aggregation switches and have 32 servers per rack with 50 Gbps per server as described in Section 3.3. We use rack sets of 4 racks; the controller monitors traffic every 500 ms and sets the link capacity threshold to 0.85. Evaluation of alternative values showed no qualitative impact for the examined workloads.

Performance metrics. The metrics of interest are *FCT*, *rack throughput* and *workload duration*. The FCT is the time span between the creation of the flow and its completion and directly impacts application performance. In the following experiments, we focus on 99th percentile FCT, denoted as *tail FCT*. We also measure the throughput of each ToR every second by aggregating throughputs of all the flows sent or received by the ToR and taking the maximum between ingress and egress throughput. We compute the average throughput achieved by the racks during the periods for which the rack is not idle. Finally, we measure the *workload duration* as the timespan between the beginning of the first and the completion of the last request. We now detail our evaluation workloads.

5.2.1 Workloads

Storage (open loop workload). We use the traces described in Section 2.2. The data written to the storage tier is stored on 159 storage racks. For each rack, the writes are controlled by the storage service and are hence uniformly distributed over the day. Reads are very bursty and dominate over writes during peaks.

Production (closed loop workload). We use the traces from [27] as described in Section 2.2 that are representative of a typical production cluster. The traces contain the number of bytes sent each second between ToR source destination pairs, without flow- or request-level information. We model this as a closed loop workload with at most one outstanding request between each source destination pair. For a source-destination pair, each second in which $x > 0$ bytes were sent corresponds to a request. The request is sent at a throughput of x Bps. For large requests that exceed 200 MB we use multiple flows to leverage ECMP across all the uplinks. In a topology provisioned for peak, each request finishes within a second and the achieved throughput equals the throughput observed in the trace. We detect the fully provisioned bandwidth for each workload by down-scaling the link bandwidth to the minimal bandwidth that allows every request to finish on time for a given workload. Then, we oversubscribe the network and examine the impact on performance. Namely, oversubscription introduces queuing delays since there is only one outstanding request per source destination pair. We generate multiple *workloads* by mapping randomly selected rack-level traces to individual racks in a rack set. Within a rack, requests are randomly distributed across servers.

5.2.2 Base performance

We now compare our design to the static topology at different oversubscription ratios. We first describe the results for the Storage workloads shown in Figure 6. Figure 6(a) shows the tail FCT for both static and reconfigurable topologies in function of the oversubscription for one representative set of four racks. The tail FCT is relatively high even for well-provisioned topologies, showing that peak bandwidth demand can be high. As expected, the tail FCT drops as the oversubscription decreases because both topologies get more network resources. However, Larry has low tail FCT even when oversubscription is high. For example, the tail FCT for Larry with 3.2x oversubscription is about 64% lower compared to a static topology with the same oversubscription. Despite the 3.2x oversubscription, it has about 42% lower tail FCT than the static topology with 2x oversubscription and within 18% of a fully provisioned network. At the highest oversubscription, Larry improves tail FCT by a factor of 2.

This happens because Larry can reconfigure its topology to efficiently allocate bandwidth to the traffic. Figure 6(b) shows the average rack throughput of Larry normalized to the static topology in function of the oversubscription ratio for the same workload. At the highest oversubscription, the throughput per rack is 1.96 times higher for Larry. As the oversubscription decreases, the number of reconfigurable uplinks increases, which improves performance compared to static topologies. However, eventually, the bandwidth per rack provisioned on static topologies gets high enough to satisfy the demand, and the reconfigurability makes less difference. Overall, Larry has higher rack throughput for all oversubscriptions and up to 2.7 times higher at 3.2x oversubscription.

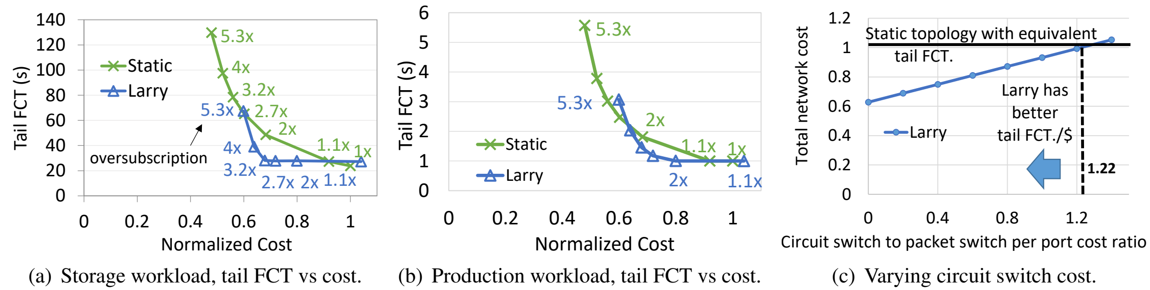
Figure 6(c) examines the generality of our findings depicting 10th and 90th percentiles as error bars, 25th and 75th percentiles as the box and the median as the cross in the box across all the workloads. The low variance reveals that our observations are consistent across all the simulations for the Storage workload.

We now focus on a representative Production workload. Figure 6(d) shows the tail FCT across different oversubscription ratios. Like in the Storage workload, we observe that reconfigurability improves tail FCT by up to approximately a factor of 2. We attribute this behavior to reduced queuing of requests. Namely, due to oversubscription, a delayed request can result in queuing of subsequent requests, which delays the overall completion of the experiment. Figure 6(e) measures the duration of the workload for Larry and static topologies normalized to the actual duration in a non-blocking network. Varying the oversubscription, we show that Larry manages to complete the workload almost in time with up to 3.2x oversubscription while static topologies take much longer to complete. Finally, Figure 6(f) shows the variance of our results across all the workloads, represented as in Figure 6(c). We observe higher variance across simulations because some workload instances have extremely low load. However, median values exhibit the same trends as for the Storage workloads.

5.2.3 Performance per dollar

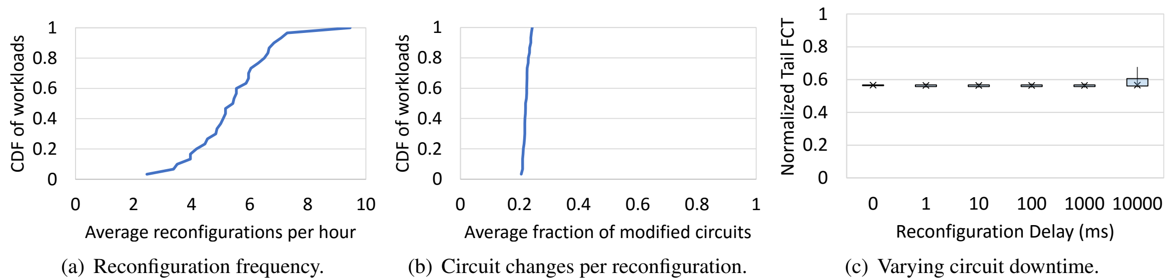
We now examine the performance per dollar of Larry. Intuitively, our design improves performance but requires additional ports and cables at the rack level, and we aim to determine whether its performance is worth the extra cost. We evaluate the deployment cost for the entire data center network using a cost model from a large DC provider that includes the volume costs of cables, merchant silicon [5], and server NICs. Figures 7(a) and 7(b) examine the tail FCT of two representative Storage and Production workloads in function of the cost across static

[5] In the model, packet switch costs are *per device* and not per port.



(a) Storage workload, tail FCT vs cost. (b) Production workload, tail FCT vs cost. (c) Varying circuit switch cost.

Figure 7: Performance as a function of cost.



(a) Reconfiguration frequency. (b) Circuit changes per reconfiguration. (c) Varying circuit downtime.

Figure 8: Reconfiguration overhead.

and reconfigurable topologies. Each data point refers to different oversubscription, ranging from 1x to 5.3x in increasing order from right to left (see the data labels). For confidentiality reasons, we normalize the costs on the x-axis to the cost of a fully provisioned topology. Based on our estimations for the hardware costs, in this analysis we assume a circuit switch to packet switch per port cost ratio of 0.3. In both workloads, Larry has a better performance per dollar than static topologies for oversubscription ratios between 2x and 4x. For example, in Figure 7(a) a static topology with 2x oversubscription has a normalized cost of 0.68, which is approximately the same cost as Larry with an oversubscription of 3.2x that has 43% lower tail FCT. These oversubscription ratios are common in current DCs, meaning that the performance improvements described in Section 5.2.2 compensates for the cost of adding reconfigurable hardware.

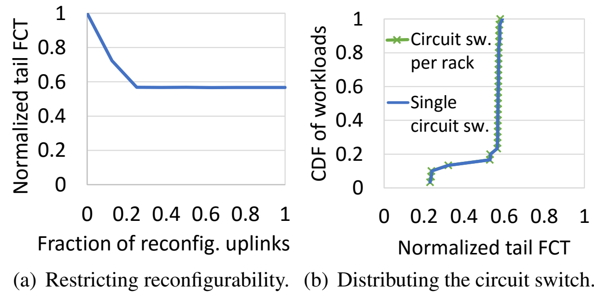
The cost of Larry depends on the cost of the circuit switches. Figure 7(c) shows the total network cost of Larry as a function of the circuit- to packet- switch per port cost ratio, for 2.7x oversubscription, which is a sensible design point in today's DC topologies. We compare it to the cost of a static topology with 1.1x oversubscription that has an equivalent 99th percentile FCT for a representative Storage workload. We can see that if the circuit switch port cost is below 1.22, Larry is cheaper while offering the same performance. This means that Larry remains cost-effective even if the circuit switch has the same per port cost as a packet switch.

5.2.4 Properties of reconfiguration

We now focus on the Storage workloads and explore the properties of the reconfiguration. Larry incurs overheads in both control and data planes, so we first estimate the reconfiguration frequency. Figure 8(a) shows the CDF of the average reconfiguration frequency. The frequency is only a few times per hour for all workloads. The highest rate is 9 reconfigurations per hour, for a median of 5. This corresponds to one reconfiguration every 7 minutes for the worst case observed. Furthermore, when reconfiguration occurs, it only affects a small fraction of all the circuits. Figure 8(b) shows the fraction of all circuits changed per reconfiguration. On average, about 22% of the circuits are modified, and 24% in the worst case. This happens because of the conservative reconfiguration policy of the controller: reconfiguration happens only when imbalance across racks is high.

We now evaluate the impact of the circuit downtime on performance. The PHY negotiation delay is expected to dominate over the circuit reconfiguration, hence we conservatively vary the downtime between 1ms and 10s. Figure 8(c) shows that downtimes up to one second have low impact on the tail FCT because during peak demand FCT is dominated by queuing at the 99th percentile.

We now vary the number of reconfigurable uplinks per rack, focusing on 2x oversubscription. With this oversubscription ratio, a rack has 8 uplinks, so the number of reconfigurable uplinks is varied between 0 and 8, and



(a) Restricting reconfigurability. (b) Distributing the circuit switch.

Figure 9: Design flexibility.

having 0 reconfigurable uplinks is equivalent to the static topology. Figure 9(a) shows the tail FCT for each setup, normalized to the tail FCT of the static topology with 2x oversubscription. Larry efficiently reduces tail FCT with as little as 2 reconfigurable uplinks per rack. This is because when a rack is congested, the other 3 are typically not. They can hence give away 6 uplinks to the congested rack, nearly doubling its aggregate uplink bandwidth.

Finally, to evaluate the overhead of using multiple circuit switches, we rerun all the Storage workloads using our uplink assignment algorithm on single large circuit switch. The algorithm then has no constraints on how the circuits are assigned and assigns uplinks only based on the traffic. Figure 9(b) shows how the tail FCT using a single switch compares to our design for all workloads. We can see that both lines overlap, showing that our greedy algorithm efficiently allocates uplink bandwidth in function of the rack demand.

6 Related Work

Local reconfigurability was described in GRIN [18] and Subways [40], showing that network resources are likely to be available locally in the DC. Both systems offload traffic at layers 2 or 3, with oveheads described in 2.2.

There has been extensive prior work on network reconfigurability using optical circuit switches [25, 39, 43, 50,54,55], free-space optics [27,30] and 60GHz wireless radios [23,29,58]. As in our design, packets sent by ToRs are forwarded over a fabric reconfigurable at the physical layer. Flat-tree is a DC network design can also adapt to different workloads by dynamically changing the topology between Clos [19] and random graphs [49] via small port-count packet or circuit switches [56]. However, these systems require substantial changes to the data plane or/and control plane of the entire DC network. Larry is not aiming to provide direct links between any ToRs in the DC, or change DC-wide network properties. Instead, it targets local uplink congestion on the ToRs. It hence does not require a DC-wide controller and can be deployed incrementally and transparently to the DC.

Flexibility can be achieved without reconfiguring the network topology. Hedera and Swan perform traffic engineering at the control plane [20, 32] and dynamically redirect traffic though least congested paths in the network. Expander topologies, such as Jellyfish [49] and Xpander [52] directly interconnect ToRs via static links and dynamically change the routing protocol in function of the load [35]. Expander topologies are not transparent, incremental deployments. They require re-cabling and custom routing policies for the entire DC network (or a large fraction of it). Larry is orthogonal to these designs and can interface with them if they are deployed.

ReacToR is a hybrid ToR switch design which combines circuit and packet switching [39]. A local classifier directs high-bandwidth flows to a set of uplinks connected core network circuit switches, while the rest traffic is directed to packet switches. ReacToR relies on host-side buffering of bursts until the circuits become available. Given the recent trend towards increasing server density per rack, XFabric proposes a hybrid design for rack-scale computers, where intra-rack traffic is forwarded by packet switches embedded on the servers' SoC over a circuit-switched physical layer [37]. Larry can be incrementally deployed without involving changes to any existing network components.

7 Conclusion

DC applications increasingly have high bandwidth demand and tight latency requirements cross rack. Larry is a network design that dynamically adapts the aggregate uplink bandwidth on the ToRs as a function of the rack demand. It ensures a low and predictable forwarding latency overhead by reconfiguring the network at the physical layer instead of re-routing traffic at layers 2 and 3. Larry can be deployed incrementally in existing DCs at the scale of a few racks, and transparently co-exist with the DC-wide controllers. It is hence well-suited for targeted deployments that improve the performance of specific tiers or services in existing DCs. We have built a prototype that uses a custom 40 Gbps electrical circuit switch and a small local controller. Using workload traces, we show that Larry improves the performance per dollar of the traditional oversubscribed networks by up to 2.3x.

Acknowledgments

We are grateful to Monia Ghobadi and Ratul Mahajan for sharing their traces. We also thank our shepherd George Porter and the anonymous reviewers for their helpful feedback.

References

[1] 100 Gbps revenues dominate in the cloud. https://aka.ms/P27bgu.

[2] Analog Devices Digital Crosspoint Switches. http://www.analog.com/en/products/switches-multiplexers/digital-crosspoint-switches.html.

[3] Arista 7050X Series. https://www.arista.com/en/products/7050x-series.

[4] Arista 7160 Series. https://www.arista.com/assets/data/pdf/Datasheets/7160-Datasheet.pdf.

[5] ConnectX-3 User Manual. http://bit.ly/2iA7Ken.

[6] Latency in Ethernet Switches. http://www.plexxi.com/wp-content/uploads/2016/01/Latency-in-Ethernet-Switches.pdf.

[7] Macom M21605 Crosspoint Switch Specification. http://www.macom.com/products/product-detail/M21605/.

[8] Macom MAXP-37161 Crosspoint Switch. https://www.macom.com/products/product-detail/MAXP-37161.

[9] Mellanox SN2000 Series. http://www.mellanox.com/page/products_dyn?product_family=251.

[10] Microsemi Digital Crosspoint Switches. http://www.microsemi.com/products/switches/digital-cross-point-switches.

[11] NTttcp Utility. https://gallery.technet.microsoft.com/NTttcp-Version-528-Now-f8b12769.

[12] OpenFlow Switch Specification. http://bit.ly/2kk3Wyo.

[13] Project Floodlight. http://www.projectfloodlight.org/floodlight/.

[14] Project Olympus Universal Motherboard. http://bit.ly/2jAeUOM.

[15] SerDes signal integrity challenges at 28Gbps and beyond. http://bit.ly/2BiqCq8.

[16] zQSFP+ Cable Assembly, 5.0m Length. http://bit.ly/2kk47FO.

[17] zQSFP+ Interconnect System. http://www.molex.com/molex/products/family?key=zqsfp_interconnect_system.

[18] AGACHE, A., DEACONESCU, R., AND RAICIU, C. Increasing Datacenter Network Utilisation with GRIN. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (2015), pp. 29–42.

[19] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (2008), pp. 63–74.

[20] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (2010), pp. 19–19.

[21] BALAKRISHNAN, S., BLACK, R., DONNELLY, A., ENGLAND, P., GLASS, A., HARPER, D., LEGTCHENKO, S., OGUS, A., PETERSON, E., AND ROWSTRON, A. Pelican: A Building Block for Exascale Cold Data Storage. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (2014).

[22] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., HARIDAS, J., UDDARAJU, C., KHATRI, H., EDWARDS, A., BEDEKAR, V., MAINALI, S., ABBASI, R., AGARWAL, A., HAQ, M. F. U., HAQ, M. I. U., BHARDWAJ, D., DAYANAND, S., ADUSUMILLI, A., MCNETT, M., SANKARAN, S., MANIVANNAN, K., AND RIGAS, L. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 143–157.

[23] CUI, Y., XIAO, S., WANG, X., YANG, Z., ZHU, C., LI, X., YANG, L., AND GE, N. Diamond: Nesting the Data Center Network with Wireless Rings in 3D Space. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation* (2016), pp. 657–669.

[24] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 54–70.

[25] FARRINGTON, N., PORTER, G., RADHAKRISHNAN, S., BAZZAZ, H. H., SUBRAMANYA, V., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proceedings of the ACM SIGCOMM 2010 Conference on Data Communication* (2010), pp. 339–350.

[26] GAO, P. X., NARAYAN, A., KARANDIKAR, S., CARREIRA, J., HAN, S., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Network Requirements for Resource Disaggregation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), pp. 249–264.

[27] GHOBADI, M., MAHAJAN, R., PHANISHAYEE, A., DEVANUR, N., KULKARNI, J., RANADE, G., BLANCHE, P.-A., RASTEGARFAR, H., GLICK, M., AND KILPER, D. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *Proceedings of the ACM SIGCOMM 2016 Conference on Data Communication* (2016), pp. 216–229.

[28] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (2009), pp. 51–62.

[29] HALPERIN, D., KANDULA, S., PADHYE, J., BAHL, P., AND WETHERALL, D. Augmenting Data Center Networks with Multi-gigabit Wireless Links. In *Proceedings of the ACM SIGCOMM 2011 Conference on Data Communication* (2011), pp. 38–49.

[30] HAMEDAZIMI, N., QAZI, Z., GUPTA, H., SEKAR, V., DAS, S. R., LONGTIN, J. P., SHAH, H., AND TANWER, A. FireFly: A Reconfigurable Wireless Data Center Fabric Using Free-space Optics. In *Proceedings of the ACM SIGCOMM 2014 Conference on Data Communication* (2014), pp. 319–330.

[31] HE, K., KHALID, J., GEMBER-JACOBSON, A., DAS, S., PRAKASH, C., AKELLA, A., LI, L. E., AND THOTTAN, M. Measuring Control Plane Latency in SDN-enabled Switches. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (2015), p. 25.

[32] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving High Utilization with Software-driven WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on Data Communication* (2013), pp. 15–26.

[33] HOPPS, C. Analysis of an Equal-Cost Multi-Path Algorithm, 2000.

[34] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), pp. 185–201.

[35] KASSING, S., VALADARSKY, A., SHAHAF, G., SHAPIRA, M., AND SINGLA, A. Beyond fat-trees without antennae, mirrors, and disco-balls. *Proceedings of the ACM SIGCOMM 2017 Conference on Data Communication* (2017).

[36] KLIMOVIC, A., KOZYRAKIS, C., THERESKA, E., JOHN, B., AND KUMAR, S. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 29.

[37] LEGTCHENKO, S., CHEN, N., CLETHEROE, D., ROWSTRON, A., WILLIAMS, H., AND ZHAO, X. XFabric: A Reconfigurable In-Rack Network for Rack-Scale Computers. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation* (2016), pp. 15–29.

[38] LEGTCHENKO, S., LI, X., ROWSTRON, A., DONNELLY, A., AND BLACK, R. Flamingo: Enabling Evolvable HDD-based Near-Line Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (2016), pp. 213–226.

[39] LIU, H., LU, F., FORENCICH, A., KAPOOR, R., TEWARI, M., VOELKER, G. M., PAPEN, G., SNOEREN, A. C., AND PORTER, G. Circuit Switching Under the Radar with REACToR. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation* (2014), pp. 1–15.

[40] LIU, V., ZHUO, D., PETER, S., KRISHNAMURTHY, A., AND ANDERSON, T. Subways: A Case for Redundant, Inexpensive Data Center Edge Links. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (2015), pp. 27:1–27:13.

[41] MARANDI, P. J., GKANTSIDIS, C., JUNQUEIRA, F., AND NARAYANAN, D. Filo: Consolidated Consensus as a Cloud Service. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (2016), pp. 237–249.

[42] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference* (2013), pp. 103–114.

[43] PORTER, G., STRONG, R., FARRINGTON, N., FORENCICH, A., CHEN-SUN, P., ROSING, T., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Integrating Microsecond Circuit Switching into the Data Center. In *Proceedings of the ACM SIGCOMM 2013 Conference on Data Communication* (2013), pp. 447–458.

[44] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 123–137.

[45] SCHMIDTKE, K. Facebook Network Architecture and Its Impact on Interconnects. In *Proceedings of 23rd Annual Symposium on High-Performance Interconnects* (2015), IEEE.

[46] SCOTT, C., WUNDSAM, A., RAGHAVAN, B., PANDA, A., OR, A., LAI, J., HUANG, E., LIU, Z., EL-HASSANY, A., WHITLOCK, S., ET AL. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. *ACM SIGCOMM Computer Communication Review 44*, 4 (2015), 395–406.

[47] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 183–197.

[48] SINGLA, A. Fat-FREE Topologies. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (2016), pp. 64–70.

[49] SINGLA, A., HONG, C.-Y., POPA, L., AND GODFREY, P. B. Jellyfish: Networking Data Centers, Randomly. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), vol. 12, pp. 17–17.

[50] SINGLA, A., SINGH, A., AND CHEN, Y. OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation* (2012), pp. 239–252.

[51] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. IOFlow: a Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), pp. 182–196.

[52] VALADARSKY, A., SHAHAF, G., DINITZ, M., AND SCHAPIRA, M. Xpander: Towards optimal-performance datacenters. In *CoNEXT* (2016), pp. 205–219.

[53] WANG, A., GUO, Y., HAO, F., LAKSHMAN, T., AND CHEN, S. Scotch: Elastically scaling up sdn control-plane using vswitch based overlay. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (2014), pp. 403–414.

[54] WANG, G., ANDERSEN, D. G., KAMINSKY, M., PAPAGIANNAKI, K., NG, T. E., KOZUCH, M., AND RYAN, M. c-Through: Part-time Optics in Data Centers. In *Proceedings of the ACM SIGCOMM 2010 Conference on Data Communication* (2010), pp. 327–338.

[55] XIA, Y., SCHLANSKER, M., NG, T. S. E., AND TOURRILHES, J. Enabling Topological Flexibility for Data Centers Using OmniSwitch. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing* (2015), pp. 4–4.

[56] XIA, Y., SUN, X. S., DZINAMARIRA, S., WU, D., HUANG, X. S., AND NG, T. S. E. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 295–308.

[57] ZHOU, X., AND NISHIZEKI, T. Edge-coloring and f-coloring for various classes of graphs. *Journal of Graph Algorithms and Applications 3*, 1 (1999), 1–18.

[58] ZHOU, X., ZHANG, Z., ZHU, Y., LI, Y., KUMAR, S., VAHDAT, A., ZHAO, B. Y., AND ZHENG, H. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 443–454.

[59] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion Control for Large-scale RDMA Deployments. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 523–536.

[60] ZILBERMAN, N., GROSVENOR, M., POPESCU, D. A., MANIHATTY-BOJAN, N., ANTICHI, G., WÓJCIK, M., AND MOORE, A. W. Where Has My Time Gone? In *International Conference on Passive and Active Network Measurement* (2017), Springer, pp. 201–214.

Semi-Oblivious Traffic Engineering: The Road Not Taken

Praveen Kumar *Cornell*, Yang Yuan *Cornell*, Chris Yu *CMU*, Nate Foster *Cornell*, Robert Kleinberg *Cornell*

Petr Lapukhov *Facebook*, Chiun Lin Lim *Facebook*, Robert Soulé *Università della Svizzera italiana*

Abstract

Networks are expected to provide reliable performance under a wide range of operating conditions, but existing traffic engineering (TE) solutions optimize for performance or robustness, but not both. A key factor that impacts the quality of a TE system is the set of paths used to carry traffic. Some systems rely on shortest paths, which leads to excessive congestion in topologies with bottleneck links, while others use paths that minimize congestion, which are brittle and prone to failure. This paper presents a system that uses a set of paths computed using Räcke's *oblivious routing* algorithm, as well as a centralized controller to dynamically adapt sending rates. Although oblivious routing and centralized TE have been studied previously in isolation, their combination is novel and powerful. We built a software framework to model TE solutions and conducted extensive experiments across a large number of topologies and scenarios, including the production backbone of a large content provider and an ISP. Our results show that semi-oblivious routing provides near-optimal performance and is far more robust than state-of-the-art systems.

1 Introduction

Two roads diverged in a wood, and I –
I took the one less traveled by,
And that has made all the difference.

—Robert Frost

Networks are expected to provide good performance even in the presence of unexpected traffic shifts and outright failures. But while there is extensive literature on how to best route traffic through a network while optimizing for objectives such as minimizing congestion [3, 9, 13, 14, 15, 22, 24, 26, 47], current traffic engineering (TE) solutions can perform poorly when operating conditions diverge from the expected [32, 42].

The tension between performance and reliability is not merely a hypothetical concern. Leading technology companies such as Google [18, 24] and Microsoft [22, 32] have identified these properties as critical issues for their private networks. For example, a central goal of Google's B4 system is to drive link utilization to 100%, but doing this means that packet loss is "inevitable" when failures occur [24]. Meanwhile a different study of availability at Google identified "no more than a few minutes of downtime per month" as a goal, where downtime is defined as packet loss above 0.1%-2% [18].

Stepping back, one can see that there are two fundamental choices in the design of any TE system: *(i)* which forwarding paths to use to carry traffic from sources to destinations, and *(ii)* which *sending rates* to use to balance incoming traffic flows among those paths. Any TE solution can be viewed in terms of these choices, but there are also practical considerations that limit the kinds of systems that can be deployed. For example, setting up and tearing down end-to-end forwarding paths is a relatively slow operation, especially in wide-area networks, which imposes a fundamental lower bound on how quickly the network can react to dynamic changes by modifying the set of forwarding paths [25]. On the other hand, modifying the sending rates for an existing set of forwarding paths is a relatively inexpensive operation that can be implemented almost instantaneously on modern switches [32]. Another important consideration is the size of the forwarding tables required to implement a TE solution, as there are limits to how many paths can be installed on each switch [7, 22, 42].

These considerations suggest that a key factor for achieving reliable performance is to select a small set of diverse forwarding paths that are able to route a range of demands under a variety of failure scenarios. Unfortunately, existing TE solutions fail to meet this challenge. For example, using k-shortest paths works well in simple settings but leads to excessive congestion in topologies with shortcut links, which become bottlenecks. Using k-edge-disjoint paths between pairs of nodes does not fare much better, since paths between different node pairs still contend for bandwidth on bottleneck links [24, 32]. Using

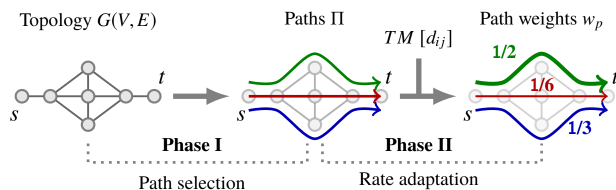


Figure 1: Semi-oblivious TE system model

Algorithm	*Load balanced*		*Diverse*	*Low-stretch*
	Capacity aware	*Globally optimized*		
SPF / ECMP	×	×	×	✓
CSPF	✓	×	×	✓
k-shortest paths (KSP)	×	×	?	✓
Edge-disjoint KSP	×	×	✓	✓
MCF	✓	✓	×	×
Robust MCF [42]	✓	✓	✓	×
VLB [45]	×	×	✓	×
B4 [24]	✓	✓	?	?
SMORE / Oblivious [39]	✓	✓	✓	✓

? : difficult to generalize without considering topology and/or demands.

Table 1: Properties of paths selected by different algorithms.

a constraint solver to compute forwarding paths that optimize for given scenarios and objectives effectively avoids bottlenecks, but it can also "overfit" to specific scenarios, yielding a brittle and error-prone solution. In addition, it is difficult to impose a budget on the number of paths used by the solver, and common heuristics for pruning the set of paths degrade performance [42].

Our approach. We present SMORE, a new TE system based on two key ingredients. First, it uses a set of forwarding paths computed using *oblivious routing*[1] [38,39], rather than shortest, edge-disjoint, or optimal paths, as in current approaches [22, 24, 32]. The paths computed by oblivious routing enjoy three important properties: they are low-stretch, diverse, and naturally balance load. Second, it dynamically adapts sending rates [9,22,26,28,47] on those paths to fit current demands and react to failures. While these ideas have been explored in isolation previously [3, 7, 20], their combination turns out to be surprisingly powerful, and allows SMORE to achieve near-optimal performance. Our work is the first practical implementation and comprehensive evaluation of this combined approach, called *semi-oblivious routing*. Through extensive experiments with data from the production networks of a large content provider (anonymized as BigNet) and a major ISP, as well as large-scale simulations, we demonstrate that SMORE achieves performance that is near-optimal, competitive with state-of-the-art solutions [22, 24], and better than the worst-case scenarios predicted in the literature. SMORE also achieves a level of robustness that improves on solutions explicitly designed to be fault tolerant [32, 42].

Contributions. Our contributions are as follows:

1. We identify a general model for TE systems, and survey various approaches to wide-area TE (§2).
2. We present SMORE's design and discuss key properties that affect performance and robustness (§3).
3. We demonstrate the deployability of SMORE on a production network, and develop a framework for modeling and evaluating TE systems (§4).
4. We conduct an extensive evaluation comparing SMORE with other systems in a variety of scenarios (§5-§6).

Overall, SMORE is a promising approach to TE that is based on solid theoretical foundations and offers attractive performance and robustness.

[1] We use the term *oblivious routing* to refer to Räcke's algorithm, and not other demand-oblivious approaches.

2 System Model and Related Work

This section develops a general model that captures the essential behavior of TE systems and briefly surveys related work in the area.

Abstractly, a TE system can be characterized in terms of two fundamental choices: which forwarding paths to use to carry traffic, and how to spread traffic over those paths. This is captured in the two phases of the model shown in Fig. 1: *(i) path selection* and *(ii) rate adaptation*. The first phase maps the network topology to a set of forwarding paths connecting each pair of nodes. Typically this phase is executed only infrequently—e.g., when the topology changes—since updating end-to-end forwarding paths is a relatively slow operation. In fact, in a wide-area network it can take as long as several minutes to update end-to-end paths due to the time required to update switch TCAMs on multiple geo-distributed devices. In the second phase, the system takes information about current demands and failures, and generates a weighted set of paths that describe how incoming flows should be mapped onto those paths. Because updating path weights is a relatively fast operation, this phase can be executed continuously as conditions evolve. For example, the system might update weights to rebalance load when demands change, or set the weight of some paths to zero when a link breaks. The main challenge studied in this paper is how to design a TE system that selects a small set of paths in the first phase that is able to flexibly handle many different scenarios in the second phase.

2.1 Path Properties

The central thesis of this paper is that path selection has a large impact on the performance and robustness of TE systems. Even for systems that incorporate a dynamic rate adaption phase to optimize for specific performance

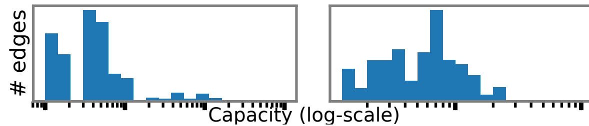


Figure 2: Link capacities in two production WANs.

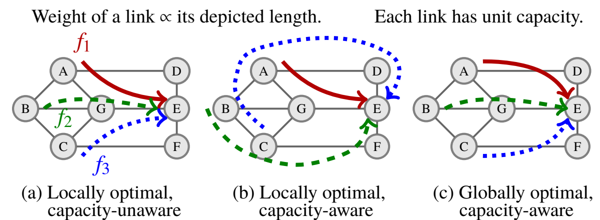


Figure 3: Local vs. globally optimal path selection

objectives, the set of paths selected is crucial as it defines the state space for optimization. Desirable path properties include:

A. *low stretch* for minimizing latency,
B. *high diversity* for ensuring robustness, and
C. *good load balancing* for achieving performance.

Unfortunately, current TE systems fail to guarantee at least one of these properties, as shown in Table 1. For example, approaches based on k-shortest paths (KSP) fail to provide good load balancing properties in many topologies due of two fundamental reasons. First, KSP is not capacity-aware. Note that wide-area topologies evolve over time to meet growing demands [18], leading to heterogeneous link capacities as shown in Fig. 2. As KSP does not consider capacities, it often over-utilizes low-capacity links that lie on many shortest paths. Using inverse of capacity as link weight is a common technique to handle this, but it can lead to increased latency due to capacity heterogeneity. Second, because KSP computes paths between each pair of nodes independently, it does not consider whether any given link is already heavily used by other pairs of nodes. Hence, lacking this notion of globally optimized path selection, even if one shifts to using seemingly more diverse edge-disjoint k-shortest paths, the union of paths for all node pairs may still over-utilize bottleneck links.

In general, to achieve low stretch and good load balancing properties, a path selection algorithm must be *capacity aware* and *globally optimized*. To illustrate, consider the topology in Fig. 3 where unit flows arrive in the order f_1, f_2, and f_3. In Fig. 3a, we use the shortest paths to route, as in KSP, and thus link (G,E) becomes congested. In Fig. 3b, we greedily assign the shortest path with sufficient capacity to each flow in order of arrival, as in CSPF, which leads to a locally optimal but globally suboptimal set of paths since some paths have high latency. Finally, in Fig. 3c we depict the globally optimal set of paths. The challenge is to compute a set of paths that closely approximates the performance of these optimal paths while remaining feasible to implement in practice.

2.2 Related Work

The textbook approach to TE merges the two phases in our model and frames it as a combinatorial optimization problem: given a capacitated network and a set of demands for flow between nodes, find an assignment of flows to paths that optimizes for some criterion, such as minimizing the *maximum link utilization* (MLU). This is known as the *multi-commodity flow* (MCF) problem in the literature, and has been extensively studied. If flows are restricted to use a single path between node pairs, then the problem is NP-complete. But if fractional flows are allowed, then optimal solutions can be found in strongly polynomial time using linear programming (LP) [44].

Another approach, which has been widely used in practice, is to tune link weights in distributed routing protocols, such as OSPF and ECMP, so they compute a good set of forwarding paths [14, 15], and not perform any rate adaptation. This approach is simple to implement as it harnesses the capabilities of widely-deployed conventional protocols, but optimizing link weights for ECMP is NP-hard. Moreover, it often performs poorly when failures occur, or during periods of re-convergence after link weights are modified to reflect new demands. COYOTE [8] aims to improve performance of such distributed approaches by carefully manipulating the view of each switch via fake protocol messages.

Several recent centralized TE systems explicitly decouple the phases of path selection and rate adaption. SWAN [22] distributes flow across a subset of k-shortest paths, using an LP formulation that reserves a small amount of "scratch capacity" for configuration updates. The system proposed by Suchara et al. [42] (henceforth referred as "R-MCF") performs a robust optimization that incorporates failures into the LP formulation to compute a diverse set of paths offline. It then uses a simple local scheme to dynamically adapt sending rates at each source. Recent work by Chang et al. [6] also used robust optimization to validate designs that provide performance and robustness guarantees that are better than worst-case bounds. FFC [32] recommends (p,q) link-switch disjoint paths and spreads traffic based on an LP to ensure resilience to up to k arbitrary failures. B4 [24] selects paths greedily based on demands. It uses BwE [28] and heuristic optimizations to divide flows onto paths to improve utilization while ensuring fairness.

Another line of work has explored the space of oblivious approaches that provide strong guarantees in the presence of arbitrary demands [2, 3, 7]. Valiant Load Balancing (VLB) routes traffic via randomly selected intermediate nodes. Originally proposed as a way to balance load

in parallel computers [45], VLB has recently been applied in a number of other settings including WANs [48]. However, the use of intermediate nodes increases path length, which can dramatically increase latency—e.g., consider routing traffic from New York to Seattle via Paris.

Oblivious routing, which generalizes VLB, computes a probability distribution on low-stretch paths and forwards traffic according to that distribution no matter what demands occur when deployed—in other words, it is *oblivious* to the demands. Remarkably, there exist oblivious routing schemes whose congestion ratio is never worse than $O(\log n)$ factor of optimal. One such scheme, proposed in a breakthrough paper by Räcke [38], constructs a set of tree-structured overlays and then uses these overlays to construct random forwarding paths. While the $O(\log n)$ congestion ratio for oblivious routing is surprisingly strong for a worst-case guarantee, it still requires overprovisioning capacities by a significant amount. Applegate and Cohen [3] developed an LP formulation of optimal oblivious routing. They showed that in contrast to the $O(\log n)$ overprovisioning suggested by Räcke's result, in most cases, it is sufficient to overprovision the capacity of each edge by a factor of 2 or less. While better than the worst-case bounds, it is still not competitive with the state-of-the-art. This lead us to explore augmenting oblivious routing for path selection with dynamic rate adaption in order to achieve better performance.

3 SMORE Design

Our design for SMORE follows the two-phase system model introduced in the preceding section: we use oblivious routing (§3.1) to select forwarding paths, and we use a constraint optimizer (§3.2) to continuously adapt the sending rates on those paths. This approach ensures that the paths used in SMORE enjoy the properties discussed in §2 by construction—i.e., they are low-stretch, diverse, and load balanced.

Performing a robust, multi-objective optimization to compute paths based on anticipated demands is challenging in practice in the presence of resource constraints [6, 32]. Moreover, if the actual conditions differ from what was predicted—e.g., due to failures, or in an ISP where customers may behave in ways that are difficult to anticipate—performance will suffer in general [3]. In contrast, because the paths in oblivious routing are computed without knowledge of the demands, they avoid overfitting to any specific scenario, which makes the system naturally robust. Finally, SMORE comes pre-equipped with a simple mechanism for imposing a budget on the total number of paths used, which allows it to degrade gracefully in the presence of resource constraints, unlike many other approaches.

3.1 Path selection

The core of SMORE's oblivious path selection is based on a structure we call a *routing tree* that implicitly defines a unique path for every node pair in the network $G(V,E)$. A routing tree comprises: *(i)* a logical tree $T(V_t,E_t)$ whose leaves correspond to nodes of G, i.e., there is a one-to-one mapping $m'_V : V \to V_t$, and *(ii)* a mapping $m_E : E_t \to E^*$ that assigns to each edge e_T of T a corresponding path in G, such that edges sharing a common endpoint in T are mapped to paths sharing a common endpoint in G. One can obtain a path $\mathrm{Path}_T(u,v)$ from u to v in G by finding the corresponding leaves $m'_V(u)$ and $m'_V(v)$ of T, identifying the edges of the unique path in T that joins these two leaves, and concatenating the corresponding physical paths based on m_E in G. Generalizing this idea, a *randomized routing tree* (RRT) is a probability distribution over routing trees. The corresponding oblivious routing scheme computes a u−v path by first sampling a routing tree T, then selecting the path $\mathrm{Path}_T(u,v)$. One way to think of oblivious routing is as a hierarchical generalization of VLB, where the network is recursively partitioned into progressively smaller subsets, and one routes from u to v by finding the smallest subset in the hierarchy that contains them both, and constructing a path through a sequence of random intermediate destinations within this subset. Räcke's breakthrough discovery [39] was an efficient, iterative algorithm for constructing RRTs.

We illustrate how the set of paths selected by SMORE have the required properties, in contrast to other well known path selection algorithms, such as ECMP, KSP, edge-disjoint k-shortest paths (EDKSP), VLB and MCF using a representative WAN topology (Hibernia Atlantic).[2] Fig. 4 shows the paths selected by various algorithms for all node pairs and uses a color-coding to indicate load (i.e., the sum of weights of paths using each link). The inset images show the latencies of paths selected by each algorithm for different node pairs.

A. ***SMORE's paths have low stretch.*** The central ingredient in Räcke's construction of RRTs is a reduction from oblivious routing to the problem of computing *low-stretch routing trees*, defined as follows. The input is an undirected graph G whose edges are assigned positive *lengths* $\ell(e)$ and *weights* $w(e)$. The length of a path P, $\ell(P)$, is defined to be the sum of its edge lengths, and the average stretch of a routing tree T is defined to be the ratio of weighted sums

$$\mathrm{stretch}(T) = \frac{\sum_{e=(u,v)} w(e)\ell(\mathrm{Path}_T(u,v))}{\sum_{e=(u,v)} w(e)\ell(e)},$$

where both sums range over all edges of G. The problem is to select T so as to minimize this quantity, which can be interpreted as the (weighted) average amount by which we

[2] From the Internet Topology Zoo (ITZ) dataset [23]

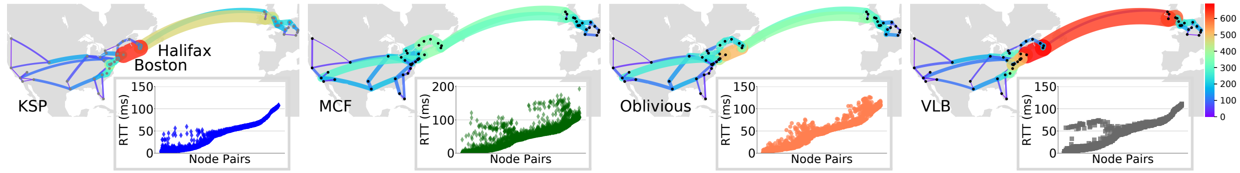


Figure 4: Figure showing sum of path weights for each link, for different path-selection algorithms. Inset shows length of paths selected by each algorithm for different node pairs. x-axis has node pairs sorted by their geographical distances.

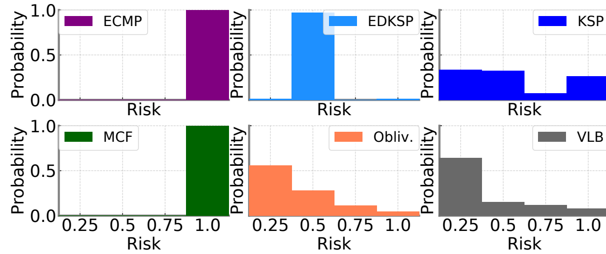


Figure 5: Distribution of *risk* (r_{euv}) for Hibernia Atlantic.

inflate the length of an edge e when we join its endpoints using the path determined by T.

For every instance of the low-stretch routing tree problem, there exists a solution with average stretch $O(\log n)$, and a randomized algorithm, due to Fakcharoenphol, Rao, and Talwar (FRT) [10], efficiently finds such a tree. The algorithm works by computing all-pairs shortest paths in G to define a metric space, then hierarchically decomposing this metric space into clusters of geometrically decreasing diameter, each with a distinguished vertex called the *cluster center*. The topology of the routing tree is defined to be the Hasse diagram of this hierarchical decomposition ordered by inclusion, and the paths associated to its edges are shortest paths between the corresponding cluster centers. Thus, to route from a source u to a destination v, one constructs a path from u by bubbling up through the hierarchy, taking shortest paths to centers of increasingly large clusters, until one reaches the center of a cluster containing both u and v; let's call this center the least common ancestor ($LCA_{u,v}$); then one reverses this process to route from that cluster center to v. If both u and v belong to a cluster C, then the length of the path thus constructed is bounded by a constant times the diameter of C. This explains why paths tend to avoid the lengthy detours that can plague VLB, especially when the source and destination are near one another as shown in insets in Fig. 4. In practice, oblivious routing is often competitive with shortest-path based approaches in terms of latency. Also note that while MCF optimizes for congestion, it may pick long detours to avoid bottlenecks.

B. *SMORE uses diverse paths for robustness.* VLB achieves robustness by routing through random intermediaries, which avoids treating any particular link as critical. Oblivious routing generalizes VLB by allowing for a hierarchy of random intermediate destinations rather than just one. A $u-v$ path is constructed by concatenating paths through a sequence of intermediate destinations representing their ancestors in the sampled routing tree T, up to and including $LCA_{u,v}$. A well-chosen RRT will have the property that the detour through $LCA_{u,v}$ rarely consumes much more capacity than directly taking a shortest path. This allows routing with RRTs to attain aggregate utilization that is nearly as efficient as shortest-path routing, worst-case load balancing that matches or improves VLB, as well as good robustness properties.

One can quantify robustness by generalizing the concept of a SRLG[3] and grouping $u-v$ paths, $\Pi(u,v)$, by the edges they share, such that an edge failure can break all the paths in the shared risk group. We define *risk*, r_{euv}, of an edge e with respect to a node pair (u,v) as the fraction of $\Pi(u,v)$ paths using e. If e is not used by (u,v), then r_{euv} is undefined. For highest resilience, $\Pi(u,v)$ consists of pairwise edge-disjoint paths, and for any edge e, $r_{euv} \leq \frac{1}{|\Pi(u,v)|}$. A fragile $\Pi(u,v)$ has paths sharing some common edge e', and $r_{e'uv} = 1$. Thus, low risk implies high resilience to faults, and low impact on congestion when reacting to failures as more paths are available to share the load. A robust set of paths will have less high risk edges. Fig. 5 shows the distribution of risk when using up to 4 paths per node pair. Ideally, EDKSP should have the entire mass at 0.25. But, a closer look at the topology reveals that for most node pairs, only two edge-disjoint paths exist, implying risk of 0.5 for all edges in those paths, as illustrated in Fig. 6. KSP always finds 4 (u,v) paths which differ slightly and these differing edges have low risk, but the significant number of overlapping edges in these paths have high risk. Interestingly, the set of paths computed using MCF also tend to be brittle. On this topology, both oblivious routing and VLB compute diverse sets of paths, and thus are robust to failures.

C. *SMORE's paths are optimized for load-balancing.* SMORE's path selection algorithm is a capacity-aware iterative algorithm that constructs a sequence of instances of the low-stretch routing tree problem, with the same graph topology but varying edge lengths, and solves each in-

[3]Shared Risk Link Groups (SRLGs) usually refer to links sharing a common physical resource. If one link fails because of the shared resource, other links in the group may fail too.

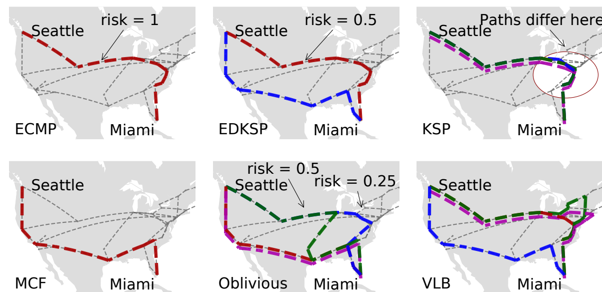


Figure 6: Paths from Seattle to Miami in the Hibernia Atlantic topology used by various TE schemes.

stance using FRT. In a given iteration, the root is selected randomly and the length of each edge is multiplied by an exponential function of the "cumulative usage", relative to capacity, of that edge in previously computed routing trees[4]. The tree computed in each iteration is thus penalized for re-using edges that have been heavily utilized in previous trees, and consequently, the ensemble of all routing trees in the sequence (with suitable probabilities) balances load among edges in a way that ensures $O(\log n)$ congestion ratio regardless of the traffic matrix (TM).

To illustrate this, consider the nodes Boston and Halifax in Fig. 4. There is a direct "shortcut" link connecting these nodes, as well as a slightly longer path to the north. Shortest path based algorithms overload the shorter link and ignore the detour, while MCF balances load equally on the two paths. Oblivious routing distributes load unequally, preferring the direct link but reducing its load by using the detour for a fraction of traffic.

We note another interesting observation based on the Seattle-Miami paths depicted in Fig. 6. Paths selected by KSP are identical for most part, and the only variation occurs at nodes in close proximity within the US northeast. In our experience, this phenomenon occurs often, and failures of such shared edges can adversely affect performance. In contrast, oblivious routing and VLB select a more diverse set of paths which are edge-disjoint with higher probability. Another intuitive way to look at this is that most path selection algorithms compute paths greedily for individual node pairs while oblivious routing globally optimizes paths considering all pairs simultaneously, like MCF. For instance, even though EDKSP computes diverse paths for individual pairs, when paths for all pairs are considered together, the shortcut links can become overloaded, as they may be used by many pairs.

[4]This is an instance of the *multiplicative weights update method* [4], a general iterative method for solving programs such as packing and covering LPs. The method has a tunable parameter ϵ which governs the trade-off between the approximation accuracy and the number of iterations required. Our implementation uses $\epsilon = 0.1$.

	Variable	*Definition*
Input	G(V, E)	Input graph
	Π	The base set of paths allowed in G
	D	Predicted traffic matrix
Auxiliary	$\Pi(s,t)$	The set of all s to t paths in Π
	$d_{(s,t)}$	Demand from s to t specified by **D**
	$\mathsf{cap}(e)$	Capacity of link e
	U_e	Expected utilization of link e
	Z	Expected maximum link utilization
	$ep(P)$	End-points of path P
Output	w_P	Weight of path P. ($w_P \in [0,1]$)

$$\begin{aligned} &\textit{minimize} \quad Z \\ &s.t.: \quad \forall s,t \in V: \sum_{P \in \Pi(s,t)} w_P = 1 \\ &\qquad \forall e \in E: \quad U_e \le Z \\ &\qquad\qquad U_e = \sum_{P \in \Pi: e \in P} \frac{w_P \cdot d_{ep(P)}}{\mathsf{cap}(e)} \\ &\qquad \forall P \in \Pi: \quad w_P \ge 0 \end{aligned}$$

Table 2: SMORE LP formulation for rate adaptation.

3.2 Rate adaptation

These observations on properties of paths selected by oblivious routing motivate using a static set of paths while dynamically adjusting the distribution of traffic over those paths as the demand varies and/or network elements fail and recover. This combination of a static set of paths and time-varying adaptation of flow rates on those paths has been called *semi-oblivious routing* [20]. From a worst-case standpoint, this approach is not significantly better than oblivious routing. Hajiaghayi et al. [20] proved that any semi-oblivious routing scheme that uses polynomially many forwarding paths must suffer a congestion ratio of $\Omega(\log n / \log(\log(n)))$ in the worst case. However, the proof of the lower bound involves constructing highly unnatural TMs and topologies such as recursive series-parallel graphs and grids satisfying specific properties. In contrast, WAN topologies grow in a planned manner, and capacities are augmented based on forecasted demands. Hence, *real-world topologies and TMs are implicitly correlated*. This raises the question of whether it is possible for semi-oblivious routing schemes to approach or match the performance of optimal MCF in practice.

In SMORE, we select the static set of paths, Π, using Räcke's algorithm to obtain a distribution over routing trees, taking the union of the path sets defined by each routing tree in the support of this distribution, pruning this distribution to the paths with the highest weights to respect path budget constraints, and then re-normalizing. To distribute flow over paths, we solve a variation of MCF using a linear program (LP). This is similar to the

usage of LP in SWAN and FFC, for instance, but with different objective function and constraint set. In SMORE the LP formulation (Table 2) is used to minimize MLU by balancing traffic over the allowed base set of paths. The output variables w_P express the relative weight of paths for each source-destination pair. The constraints ensure that the weights sum to 1 (i.e., all flows are assigned some path) and that capacity constraints are respected.

4 Implementation

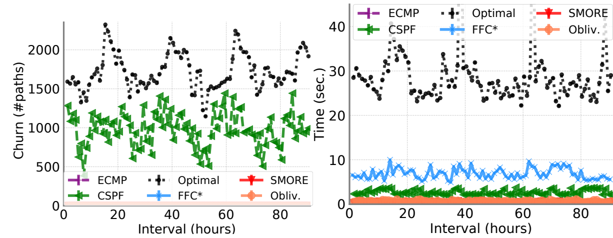
We discuss two implementations to understand and evaluate TE systems. First, we describe a real-world deployment in a production WAN. We highlight a number of practical issues that arose in that deployment (§4.1). Second, we present an implementation using YATES [29], a general framework for rapid prototyping and evaluation of TE approaches (§4.2). We discuss how we calibrated YATES's simulator against a hardware testbed and the production WAN.

4.1 IDN Deployment

To better understand the practical challenges associated with bringing SMORE to production, we deployed a TE system, which dynamically load-balances traffic over a static set of paths, on BigNet's inter-datacenter network (IDN), which is similar to Google's B4 [24] and Facebook's EBB [11].

Architecture. IDN consists of four identical planes (topologies), each of which can be programmed independently. The backbone routers at each datacenter site are connected to an aggregation layer similar to Fat-Cat [41], which distributes outgoing traffic across the planes equally using ECMP. IDN employs a hybrid control model with distributed LSP agents as well as a centralized controller. It supports two traffic classes (high and low priority) which can be managed using different TE algorithms. This architecture facilitates experimenting with different TE algorithms on a subset of planes while the other planes provide a safe fallback.

Controller. The IDN controller allows the routes for each plane to be updated every 15 seconds. The inputs to the controller are obtained from a *state snapshotter* service that captures the live state of the network including: *(i)* configured components for IDN from a central repository for network information [43], *(ii)* live link state information from Open/R [12], *(iii)* any operational overrides (link, router, or plane drains), and *(iv)* real-time TM estimated from sFlow [37] samples exported by routers. When it receives a snapshot, the controller first computes a new set of routes and splitting ratios and then sends instructions to reconfigure the routers as needed.



(a) Path churn per TM **(b)** Re-solver time per TM

Figure 7: Overhead of OPTIMAL TE on BigNet's LBN.

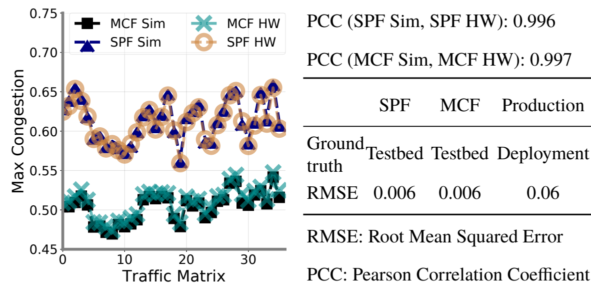
Path budget. The IDN controller maintains an *MPLS LSP mesh*—i.e., a set of LSPs connecting every pair of end points. For operational simplicity and to (indirectly) bound the number of forwarding entries that must be installed on routers, we limit the number of LSPs per node pair to a fixed budget, typically 4.

Traffic splitting. To allow splitting traffic over different LSPs, IDN supports programming up to 64 *next-hop groups* on the ingress router for each pair of nodes. Multiple next-hop groups can map to the same path, and thus we can split traffic among LSPs at granularities of up to $\frac{1}{64}$. Since packets are mapped to next-hop groups (and paths) based on hashing header fields, packets belonging to the same flow take the same path and avoid any issues related to packet reordering in multipath transmission.

Failures. In the event of a failure, traffic is routed along pre-programmed backup paths until the IDN controller computes a new routing scheme. In addition to data-plane faults, failures can also arise due to control-plane errors, such as router misconfiguration or control-plane having an inconsistent view of the network. We proactively test resilience using a *fault-injector* that can introduce both kinds of failures in a controlled manner.

State churn and update time. To quantify the operational overheads of different TE approaches, we measure path churn and the time required to compute an updated routing scheme. Churn is undesirable for several reasons: it increases CPU and memory load on routers and adds significant complexity to the management infrastructure. Fig. 7a shows the number of paths that would be changed every hour when running MCF and CSPF[5]. Likewise, routing schemes that are expensive to compute impose a burden on the controller. Fortunately, the LP that SMORE uses for rate adaptation is less complex than the LP used to solve MCF—the time needed to solve each problem instance is two orders of magnitude less than MCF (order of 100ms vs. 10s), as shown in Fig. 7b, using Gurobi [19] for optimization on a 16-core machine with 2.6 GHz CPU. Furthermore, because SMORE only updates path weights, which takes just a few milliseconds, it is more responsive than MCF, which requires updating whole paths, which can take tens of seconds [24, 32].

[5] We implement a centralized version with 80% link capacities.



PCC (SPF Sim, SPF HW): 0.996

PCC (MCF Sim, MCF HW): 0.997

	SPF	MCF	Production
Ground truth	Testbed	Testbed	Deployment
RMSE	0.006	0.006	0.06

RMSE: Root Mean Squared Error

PCC: Pearson Correlation Coefficient

Figure 8: Calibration: simulator vs. SDN testbed and LBN.

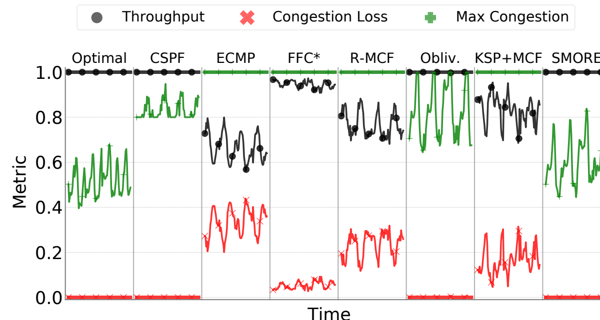


Figure 9: Expected performance on LBN over half a week.

4.2 YATES Framework

We evaluate the performance of a wide range of TE approaches under a variety of workloads and operational scenarios using YATES [29], which consists of a TE simulator and a SDN prototype. Although numerous simulators and emulators have been proposed over the years [5,21,30,34,36,46], YATES is designed specifically to evaluate TE algorithms. With YATES, TE algorithms are implemented as modules against a general interface. Table 3 contains a partial list of TE algorithms that we have implemented and made available under an open-source license[6]. For clarity, we report results from only a subset of these.

Simulator. YATES's simulator can model diverse operational conditions and record detailed statistics. It requires three inputs: *(i)* topology, *(ii)* a timeseries of *actual* TMs to simulate network load, and *(iii)* corresponding *predicted* TMs. For each predicted TM, it computes the routing scheme based on the algorithm (OPTIMAL uses actual TMs) and then simulates the flow of traffic where each source generates traffic based on the actual TM. We choose the *fluid model* [27] to simulate traffic owing to its scalability without sacrificing accuracy of macroscopic behavior. In case the actual TM is unsatisfiable using the routing scheme, YATES still admits the entire demand at each source. However, it assigns each flow its max-min fair share at each oversubscribed link and drops any traffic exceeding the flow's fair share for that link.

SDN prototype. The prototype, which consists of a SDN controller and an end-host agent, allows us to evaluate different TE algorithms using an approach similar to centralized MPLS-TE. The controller manages the forwarding rules installed on OpenFlow-enabled switches, while the end-host agent, implemented as a Linux kernel module, splits flows and assigns them to paths at the source. Although SDN allows us to easily implement the backend, it is not a requirement. It can be easily replaced with other control mechanisms, such as PCEP [31].

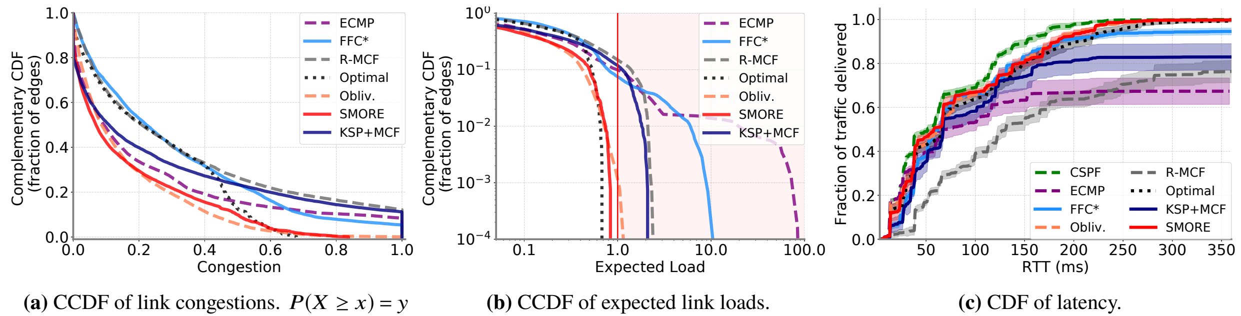
Simulator calibration. For simulation results to be credible, it is critical that they accurately correspond to results from real deployments. We validate the accuracy of YATES's simulator with: *(i)* a small-scale hardware testbed, and *(ii)* BigNet's large backbone network (LBN) (§5). We use the SDN backend to emulate Internet2's Abilene backbone network [1] on a testbed of 12 switches and replay traffic based on NetFlow traces collected from the actual Abilene network. As shown in Fig. 8, the simulation results closely match observed results in the hardware testbed. We also implement a centralized approximation of the distributed TE algorithm used in production in LBN. The network and demands are highly dynamic, and the production TE scheme reacts to such changes at a very fine time scale. As a result, we are able to only approximate its behavior. Still, the values reported by YATES closely match those seen in production.

[6]http://github.com/cornell-netlab/yates

5 BigNet WAN

We evaluate SMORE in a production setting using multiple criteria. For the setting, we use data from BigNet's large backbone network (LBN). LBN is one of the largest global deployments and carries a mix of traffic ranging from real-time video streaming and messaging to massive data synchronization globally. For criteria, we focus on four key questions: How close is SMORE to optimal in terms of performance (§5.1)? What is the impact on latency for not choosing strictly shortest paths (§5.2)? How is performance impacted under failures (§5.3) and other operational constraints (§5.4)? §6 explores whether these results generalize to other settings, using large-scale simulations over a diverse set of network scenarios.

Overview of BigNet's WAN. The network models a common content-provider design, with connections between several large datacenters across Asia, Europe, and the US as well as connectivity to numerous Points-of-Presence (PoPs) around the globe. The topology has hundreds of routers and thousands of high-speed interconnecting links, varying vastly in capacity and latency. This heterogeneity largely stems from the way the net-



(a) CCDF of link congestions. $P(X \geq x) = y$ **(b)** CCDF of expected link loads. **(c)** CDF of latency.

Figure 10: BigNet LBN: predicted distribution of expected load, congestion and latency.

work evolved over many years. The topology exhibits clustered structure, with clusters following geographic constraints imposed by continents and links between clusters running over transoceanic paths, similar to the topology that we used for illustration in §3.

Traffic on LBN exhibits multiple strong diurnal patterns, modulated by the activities of billions of users in different time zones. Being a global system, different parts of the network experience peak loads at different times. Overall traffic patterns over the WAN can be split into two major categories: *(i)* traffic between datacenters, and *(ii)* traffic from datacenters to PoPs. Inter-datacenter traffic typically consists of various replication workflows, such as those related to cache consistency or bulk traffic for Hadoop and database replication. A significant amount of this traffic is delay tolerant, and could be routed over non-shortest paths between the datacenters. However, the traffic from datacenters to PoPs is latency sensitive, as it represents content routed to BigNet users.

Methodology. We collect production data from LBN consisting of accurate network topology, link capacities, link latencies, aggregate site-to-site TMs, and paths used by traffic in production. Using this data, we perform high-fidelity simulations with Yates and present results based on the statistics reported. Traffic with different latency requirements are routed separately in production to avoid excessive path stretch for latency-sensitive traffic. For simplicity, we choose to route both types of traffic using the same TE scheme. Traffic on BigNet's WAN is growing at a rapid pace, and so the network also evolves with it. We present results based on the network state and demands for a month in late 2016.

5.1 Performance

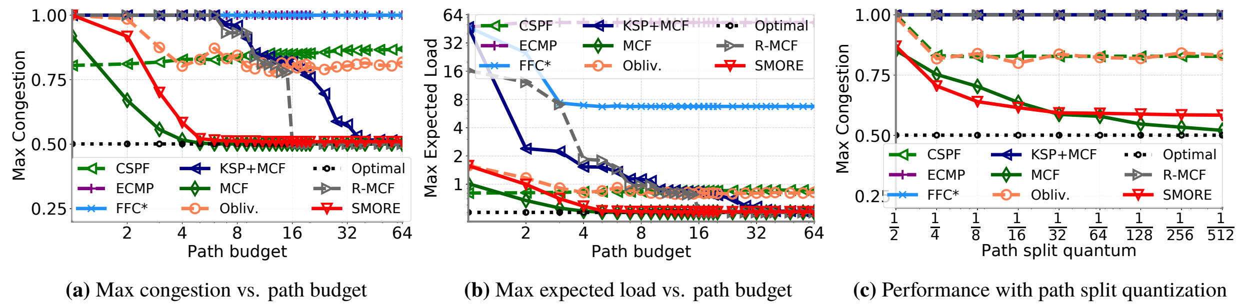
For each hourly snapshot of the network under regular operating conditions (i.e. without any failures), we measure various performance statistics (with a path budget $k = 4$, same as reported in B4 [24]) using different TE approaches. Fig. 9 shows the *(i)* throughput normalized to total demand, *(ii)* maximum congestion (fractional link utilization), and *(iii)* normalized traffic dropped due to congestion over a period of half a week. CSPF and Optimal[7] dynamically compute paths with sufficient capacity for each flow, and thus avoid congestion. Remarkably, only oblivious routing and Smore are able to achieve 100% throughput, while other centralized TE approaches aren't able to do so and introduce bottlenecks.

As expected, Optimal (which uses MCF to minimize MLU) achieves the lowest maximum congestion, which varies between 0.40 and 0.67 following a diurnal pattern. We find that oblivious routing performance remains within a factor of 2 as had been previously studied [3], while Smore is closest to optimal with maximum congestion within 16% of Optimal, on average and within 41% in the worst case.

Clearly, Smore's path selection plays a crucial role in it being so competitive. To gain further insight, we examine the distribution of congestion and expected link utilizations, i.e., how much traffic each link would have carried if packets weren't dropped due to capacity constraints. Figs. 10a and 10b show the corresponding complementary CDFs. We observe that ECMP[8], FFC*[9], R-MCF and KSP+MCF (an approximation of SWAN's path selection and rate adaptation)[10] scheduled ~10% of links to carry traffic exceeding their capacity—ECMP even oversubscribed a link 80×! We find that these bottleneck links usually appear in the shortest paths between many pairs of nodes. In contrast, Räcke's algorithm iteratively samples paths while avoiding overloading any link, and Smore load balances over these paths to reduce congestion even further. On scaling up the demands, we do expect to see congestion loss with all the approaches, including Smore. From Fig. 10a, we also note that Smore maintains a lower congestion consistently for all links and has a 95th-percentile congestion of 0.57—same as Optimal.

Even though FFC*, KSP+MCF and R-MCF dynamically load balance traffic over a set of paths, they perform suboptimally. This could be because the paths selected under the budget constraint did not provide enough flexi-

[7] Optimal does not have any budget or other operational constraints.
[8] Using RTTs as link weights for computing shortest paths.
[9] Our implementation configures FFC to handle single link failures by combining edge-disjoint k-shortest paths with fault-tolerance LP.
[10] We implement a version that uses k-shortest paths as tunnels and uses MCF to assign path weights.



(a) Max congestion vs. path budget **(b)** Max expected load vs. path budget **(c)** Performance with path split quantization

Figure 11: Performance with different operational parameters on BigNet's LBN network.

bility to eliminate bottlenecks for any traffic splitting ratio, and this was further exacerbated as the inadmissible fraction of demands also contributed to congestion before being dropped. To validate this, we measure performance with increasing budget in Figs. 11a and 11b. KSP+MCF and R-MCF, indeed, become near-optimal when the sets of paths become diverse enough. However, FFC* doesn't improve beyond a point as the number of disjoint paths is very limited. Even though the "working set" [22] of paths for KSP+MCF is small, it needs 8× as many total number of paths as SMORE to achieve similar performance.

5.2 Latency

Fig. 10c shows the distribution of latency as a fraction of total demand that is delivered within a given latency[11]. To compute latency experienced by traffic along a path, we simply sum the measured RTTs for each hop along the path. TE approaches which route over shortest paths while respecting capacity constraints, like CSPF, have the least latency. Oblivious routing doesn't ensure that the shortest paths are necessarily selected. However, as we showed in §3.1, the paths computed have low stretch. SMORE uses the same set of low-stretch paths. Intuitively, longer paths can increase congestion as the same set of packets contribute to congestion at more links. As SMORE optimizes for congestion, it also indirectly favors shorter paths. We find that SMORE is competitive with other shortest-path based approaches. Even if we ignore dropped traffic and normalize y−values in Fig. 10c with throughput, the median latency for SMORE (58.3ms) is similar to KSP+MCF (62.7ms). We find that for any node pair, SMORE finds a path with latency within 1.09× the shortest path, on average. Furthermore, if we include factors such as buffering at routers, which depends on congestion, we expect to see better latency for SMORE as it has better congestion guarantees.

5.3 Robustness

There is a trade-off between performance and robustness, and often TE systems that optimize for performance

[11] Assuming dropped traffic has infinite latency, curves reach a maximum y-value equal to throughput.

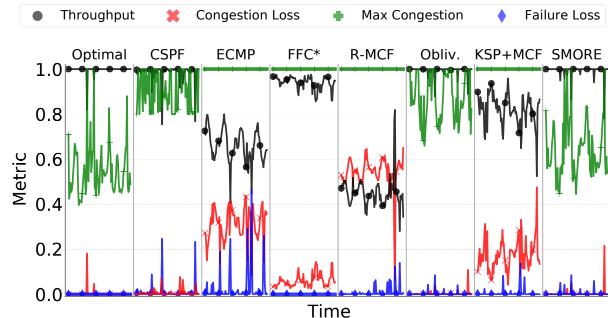


Figure 12: Expected robustness on LBN over half a week.

tend to overfit and become brittle. In Fig. 12, we evaluate the robustness of TE approaches to network failures. Here, we fail a unique link every hour and note the impact on performance. We implement a simple recovery mechanism which re-normalizes path weights to shift traffic from failed paths on to unaffected paths, if available. The recovery method is fast as it does not need setting up new paths, and it decreases loss due to failure at the cost of congestion. Usually, this increases throughput, but there are exceptions as illustrated in §B.1. We see this with R-MCF in Fig. 12. OPTIMAL knows failures in advance and reacts by setting up globally optimal paths instantaneously. FFC* is always able to find backup paths as it uses disjoint paths, and thus avoids loss due to failure. However, these paths are suboptimal for achieving the best throughput as congestion causes packet drops.

SMORE continues to deliver ~100% throughput. Although maximum congestion increases because of recovery, SMORE remains within 18% of OPTIMAL, on average and within 71% in the worst case. SMORE's high resilience can be attributed to the fact that the paths it uses are diverse and have low risk, as we saw earlier in §3.1. This ensures that, in most cases, SMORE has sufficient options to re-route traffic and load balance efficiently without overloading any link.

5.4 Operational constraints

Various operational constraints need to be accounted for while deploying a TE system. We describe one such constraint—*path-split quantization*. So far, our evalu-

ation has assumed that traffic could be split in arbitrary proportions. This is usually not the case, and path weights are quantized. Most routers support splitting by allowing to specify a certain number, typically up to 64, of next-hop groups [24]. This means that path weights should be multiples of the path-split quantum, $\frac{1}{64}$. Fig. 11c shows the impact of quantization on performance (at path budget of 4). We approximate traffic split ratio generated by different TE schemes to be multiples of the path-split quantum using a greedy approach. SMORE degrades gracefully when quantization becomes restrictive and performs well for practical settings when path-split quantum $\geq \frac{1}{64}$.

6 Large-Scale Simulations

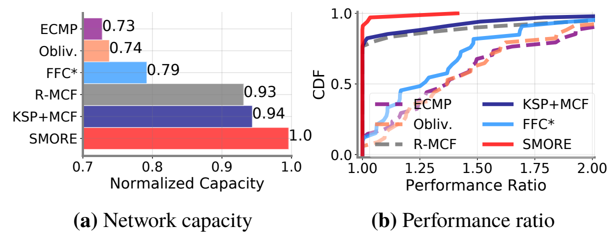
The evaluation on LBN in the preceding section showed that SMORE achieves near-optimal performance and robustness for the topology and workload in a large production network. We also obtained similar results for experiments conducted using data from a major ISP (omitted from paper due to space constraints). In this section, we show that these performance (§6.1) and robustness (§6.2) results generalize to a wider range of scenarios.

Methodology. We evaluate 17 TE algorithms over 262 ISP and inter-DC WAN topologies using YATES. We model a diverse set of operational conditions by varying demands, failures, TM prediction, and path budget. We present a subset of our experimental data that illustrates our main results over the scenarios described next.

Topologies. We select 28 topologies, shown in §C, from ITZ and other real-world networks, to overlap with ones used to evaluate TE approaches in the literature [24, 26].

TM Generation. We use YATES to generate TMs based on the gravity model [40], which assigns a weight w_i to each host i and assumes that $i \longrightarrow j$ demand $\propto w_i \cdot w_j$. We sample w_i from a heavy-tailed Pareto distribution obtained by fitting real-world TMs. We model diurnal and weekly variations by randomly perturbing the Fourier coefficients of the observed time-series and temporal variations at a finer scale by using the Metropolis-Hastings algorithm to sample from a Markov chain on the space of TMs, whose stationary distribution is the gravity model with Pareto-distributed weights described above. The algorithm updates w_i at each time step to model gradual variation over time. The following experiments scale TMs such that the minimum possible MLU for the first TM is 0.4, which matches the average MLU observed in traditional overprovisioned WANs [22, 24].

TM Prediction. YATES offers a suite of algorithms to predict future TMs. These include standard ML methods such as linear regression, lasso/ridge regression, logistic regression, random forest prediction etc., as well as algebraic methods like FFT fit and polynomial fit. For each pair of hosts, we perform an independent time series prediction where, at each time step t, the demand is predicted using demands observed in $[t-k, t)$. We optimize the size of the sliding window (k) separately using cross-validation. Regression and random forest models are trained using the previous k time steps as k features. FFT fit finds a function with a bounded number of non-zero Fourier coefficients, while polynomial fit finds a bounded-degree polynomial function that minimizes the absolute difference between predicted and actual TMs over the past k time steps. This best-fit function is evaluated in the current time step to yield the predicted demand. Finally, YATES exposes an error parameter to assess the sensitivity of TE algorithms to inaccuracy in prediction.



(a) Network capacity **(b)** Performance ratio

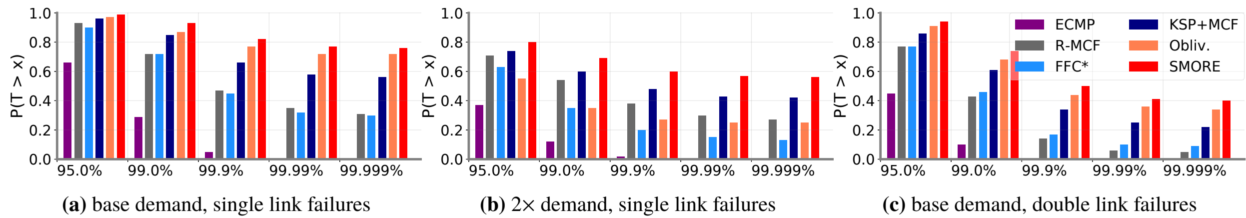
Figure 13: Aggregate performance at base demand.

6.1 Performance

We start by evaluating basic properties of TE approaches including the effective capacity of the network and the performance ratio with respect to OPTIMAL.

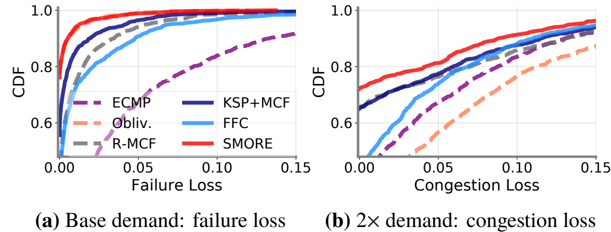
Network capacity. During normal operation, the MLU of a network is usually below 1, meaning there is spare capacity in the network. Given a routing scheme and a TM, we can define *network capacity* as the factor by which the TM can be scaled up before it experiences congestion. This spare capacity could be used to handle unexpected surges in traffic, or to schedule background traffic. As OPTIMAL minimizes MLU, it has the highest possible network capacity. Fig. 13a shows network capacity for different TE approaches, normalized with respect to OPTIMAL. As expected, oblivious routing, which can use up more bandwidth on more links to route the same TM, is unable to admit a significant fraction of TMs that OPTIMAL can handle. We find that *SMORE has the highest network capacity and is near-optimal* owing to efficient load balancing over a diverse set of paths.

Performance ratio. Another way to compare TE approaches is to measure how far they are from OPTIMAL, with respect to minimizing MLU for a given set of TMs. Here, we follow the metric defined by Applegate and Cohen [3], but use throughput after accounting for congestion loss, if any, instead of using the demand TM to compute congestion and performance ratio. Fig. 13b compares the distribution of performance ratio over various topologies and TMs. We find that SMORE and KSP+MCF



(a) base demand, single link failures **(b)** 2× demand, single link failures **(c)** base demand, double link failures

Figure 14: Robustness: Probability of achieving a throughput SLA (x) under different conditions.



(a) Base demand: failure loss **(b)** 2× demand: congestion loss

Figure 15: Robustness: CDF of throughput loss

remain optimal in 75-80% of the cases but *SMORE has closest to optimal performance ratio* (§C.1).

6.2 Robustness

Similar to §5.3, we systematically inject failures in the network to study the trade-off between performance and robustness. Fig. 15 shows the distribution of loss in throughput over all possible single link failures. With base demands, failures are the main cause of loss. As demand scales up, loss due to congestion becomes significant. *SMORE performs better in both cases by being more fault tolerant as well as spreading traffic evenly to avoid congestion*. Although TE approaches like FFC* and R-MCF are designed to be fault tolerant, they do not achieve the best throughput. This is because FFC* relies on disjoint paths to be available to reroute traffic; the number of such paths is limited in real-world topologies. For instance, GÉANT's topology has nodes with degree 1. These nodes have a single edge-disjoint path to any other node; the failure of any edge along this path leads to loss of traffic. Using (p,q) link-switch disjoint paths also doesn't improve the robustness much. Moreover, FFC* is not congestion-optimal by design and incurs high loss due to congestion as demands increase. R-MCF relies on using a large number of paths for fault tolerance and applying a budget deteriorates its performance [42].

Typically, SLAs refer to availability of a network in terms of "nines". This can also be translated in terms of throughput and given a failure characteristic [17, 18, 33], a network operator could be interested in questions such as "what is the probability that throughput is greater than 99.9%?" Fig. 14 compares TE approaches on how likely are they to achieve different levels of SLA under various operational conditions. In addition to the scenario where single link failures can happen with uniform probability under regular load (Fig. 14a), we perform two more experiments where we study robustness under increased load (Fig. 14b), and concurrent failures (Fig. 14c). We find that SMORE consistently outperforms other TE approaches. Oblivious TE is robust under both single and concurrent link failures at base demands, but its resilience deteriorates for increased load. SMORE benefits from the robust set of paths selected by oblivious routing, and also load balances efficiently even during increased load. Thus, *SMORE is highly robust and achieves SLAs with highest probability under diverse operational conditions.*

7 Conclusion

In TE, there is a fundamental trade-off between performance and robustness. Most systems are designed to optimize for one or the other, but few manage to achieve both. This challenge is further exacerbated by operational restrictions such as the number of paths, overhead due to churn, quantized splitting ratio imposed by hardware, etc.

This paper presents SMORE, a new approach that navigates these trade-offs by combining careful path selection with dynamic weight adaptation. As shown through a detailed evaluation on a production backbone network, SMORE achieves near-optimal performance in terms of congestion and load balancing metrics, is competitive with shortest-path based approaches in terms of latency, and is also robust, allowing traffic to be re-routed around failures without introducing bottlenecks while respecting operational constraints. Our large-scale evaluation shows that these performance and robustness guarantees hold across a broader class of networks. More generally, our experiences designing and implementing SMORE suggests lessons that are broadly applicable to TE systems including the importance of capacity-aware and globally optimized selection of low-stretch and diverse paths, as well as the consideration of operational constraints when building a practical TE system.

Acknowledgments. We would like to thank the anonymous NSDI reviewers and our shepherd Srikanth Kandula for their valuable feedback. This work was partially supported by NSF grant CCF-1637532 and ONR grant N00014-15-1-2177. We are grateful to Omar Baldonado and Sandeep Hebbani for their continued support, and we thank Bruce Maggs, Ratul Mahajan, Nick McKeown, Jennifer Rexford, Michael Schapira, Amin Vahdat and Minlan Yu for helpful discussions.

References

[1] Historical Abilene Data. http://noc.net.internet2.edu/i2network/live-network-status/historical-abilene-data.html.

[2] Altin, A., Fortz, B., and Ümit, H. Oblivious OSPF Routing with Weight Optimization under Polyhedral Demand Uncertainty. *Networks 60*, 2 (2012), 132–139.

[3] Applegate, D., and Cohen, E. Making Intra-domain Routing Robust to Changing and Uncertain Traffic Demands: Understanding Fundamental Tradeoffs. In *ACM SIGCOMM* (2003).

[4] Arora, S., Hazan, E., and Kale, S. The Multiplicative Weights Update Method: a Meta-Algorithm and Applications. *Transactions on Computers 8*, 1 (2012), 121–164.

[5] Chang, X. Network Simulations with OPNET. In *31st Conference on Winter Simulation* (1999), ACM.

[6] Chang, Y., Rao, S., and Tawarmalani, M. Robust Validation of Network Designs under Uncertain Demands and Failures. In *USENIX NSDI* (2017).

[7] Chiesa, M., Kindler, G., and Schapira, M. Traffic Engineering with Equal-Cost-MultiPath: An Algorithmic Perspective. *IEEE/ACM Transactions on Networking* (2016).

[8] Chiesa, M., Rétvári, G., and Schapira, M. Lying Your Way to Better Traffic Engineering. In *ACM CoNEXT* (2016).

[9] Elwalid, A., Jin, C., Low, S., and Widjaja, I. MATE: MPLS Adaptive Traffic Engineering. In *IEEE INFOCOM* (2001).

[10] Fakcharoenphol, J., Rao, S., and Talwar, K. A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics. In *35th STOC* (2003), pp. 448–455.

[11] Building Express Backbone: Facebook's new long-haul network. http://code.facebook.com/posts/1782709872057497/building-express-backbone-facebook-s-new-long-haul-network.

[12] Introducing Open/R - a new modular routing platform. http://code.facebook.com/posts/1142111519143652/introducing-open-r-a-new-modular-routing-platform.

[13] Fischer, S., Kammenhuber, N., and Feldmann, A. REPLEX: Dynamic Traffic Engineering Based on Wardrop Routing Policies. In *ACM CoNEXT* (2006).

[14] Fortz, B., Rexford, J., and Thorup, M. Traffic Engineering with Traditional IP Routing Protocols. *IEEE Communications Magazine 40*, 10 (Oct. 2002).

[15] Fortz, B., and Thorup, M. Internet Traffic Engineering by Optimizing OSPF Weights. In *IEEE INFOCOM* (2000), vol. 2.

[16] Garg, N., and Könemann, J. Faster and Simpler Algorithms for Multicommodity Flow and Other Fractional Packing Problems. *SICOMP 37*, 2 (May 2007), 630–652.

[17] Gill, P., Jain, N., and Nagappan, N. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. *ACM SIGCOMM CCR 41*, 4 (2011).

[18] Govindan, R., Minei, I., Kallahalla, M., Koley, B., and Vahdat, A. Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure. In *ACM SIGCOMM* (2016).

[19] Gurobi Optimizer. http://www.gurobi.com.

[20] Hajiaghayi, M., Kleinberg, R., and Leighton, T. Semi-oblivious Routing: Lower Bounds. In *SODA* (2007), pp. 929–938.

[21] Henderson, T. R., Lacage, M., Riley, G. F., Dowell, C., and Kopena, J. Network Simulations with the ns-3 Simulator. *SIGCOMM demonstration 14* (2008).

[22] Hong, C.-Y., Kandula, S., Mahajan, R., Zhang, M., Gill, V., Nanduri, M., and Wattenhofer, R. Achieving High Utilization with Software-Driven WAN. In *ACM SIGCOMM* (2013).

[23] The Internet Topology Zoo. http://www.topology-zoo.org.

[24] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., and Vahdat, A. B4: Experience with a Globally Deployed Software Defined WAN. In *ACM SIGCOMM* (2013).

[25] Jin, X., Liu, H., Gandhi, R., Kandula, S., Mahajan, R., Rexford, J., Wattenhofer, R., and Zhang, M. Dionysus: Dynamic Scheduling of Network Updates. In *ACM SIGCOMM* (2014).

[26] Kandula, S., Katabi, D., Davie, B., and Charny, A. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *ACM SIGCOMM* (2005).

[27] Kelly, F., and Williams, R. Fluid Model for a Network Operating under a Fair Bandwidth-Sharing Policy. *The Annals of Applied Probability 14*, 3 (2004).

[28] Kumar, A., Jain, S., Naik, U., Raghuraman, A., Kasinadhuni, N., Zermeno, E. C., Gunn, C. S., Ai, J., Carlin, B., Amarandei-Stavila, M., et al. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *ACM SIGCOMM* (2015).

[29] Kumar, P., Yu, C., Yuan, Y., Foster, N., Kleinberg, R., and Soulé, R. YATES: Rapid Prototyping for Traffic Engineering Systems. In *ACM SOSR* (2018).

[30] Lantz, B., Heller, B., and McKeown, N. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *ACM HotNets* (2010).

[31] Le Roux, J., and Vasseur, J. Path Computation Element (PCE) Communication Protocol (PCEP). https://tools.ietf.org/html/rfc5440, 2009.

[32] Liu, H. H., Kandula, S., Mahajan, R., Zhang, M., and Gelernter, D. Traffic Engineering with Forward Fault Correction. In *ACM SIGCOMM* (2014).

[33] Markopoulou, A., Iannaccone, G., Bhattacharyya, S., Chuah, C.-N., Ganjali, Y., and Diot, C. Characterization of Failures in an Operational IP Backbone Network. *IEEE/ACM Transactions on Networking 16*, 4 (2008), 749–762.

[34] McCanne, S., and Floyd, S. NS network simulator, 1995.

[35] Mitra, D., and Ramakrishnan, K. A Case Study of Multiservice, Multipriority Traffic Engineering Design for Data Networks. In *IEEE GLOBECOM* (1999), vol. 1.

[36] Cisco NetSim Network Simulator. http://www.boson.com/netsim-cisco-network-simulator.

[37] Panchen, S., Phaal, P., and McKee, N. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. https://tools.ietf.org/html/rfc3176, 2001.

[38] Räcke, H. Minimizing Congestion in General Networks. In *43rd FOCS* (2002), pp. 43–52.

[39] Räcke, H. Optimal Hierarchical Decompositions for Congestion Minimization in Networks. In *40th STOC* (2008), pp. 255–264.

[40] Roughan, M., Greenberg, A., Kalmanek, C., Rumsewicz, M., Yates, J., and Zhang, Y. Experience in Measuring Backbone Traffic Variability: Models, Metrics, Measurements and Meaning. In *IMC* (2002), ACM, pp. 91–92.

[41] Roy, A., Zeng, H., Bagga, J., Porter, G., and Snoeren, A. C. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM* (2015).

[42] Suchara, M., Xu, D., Doverspike, R., Johnson, D., and Rexford, J. Network Architecture for Joint Failure Recovery and Traffic Engineering. In *SIGMETRICS* (2011), pp. 97–108.

[43] Sung, Y.-W. E., Tie, X., Wong, S. H., and Zeng, H. Robotron: Top-down Network Management at Facebook Scale. In *ACM SIGCOMM* (2016).

[44] Tardos, E. A Strongly Polynomial Algorithm to Solve Combinatorial Linear Programs. *Operations Research 34*, 2 (1986), 250–256.

[45] Valiant, L. A Scheme for Fast Parallel Communication. *SICOMP 11*, 2 (1982), 350–361.

[46] Varga, A., et al. The OMNeT++ Discrete Event Simulation System. In *European Simulation Multiconference* (2001).

[47] Wang, H., Xie, H., Qiu, L., Yang, Y. R., Zhang, Y., and Greenberg, A. COPE: Traffic Engineering in Dynamic Networks. In *ACM SIGCOMM* (2006).

[48] Zhang-Shen, R., and McKeown, N. Designing a Fault-Tolerant Network Using Valiant Load-Balancing. In *IEEE INFOCOM* (Apr. 2008).

Metron: NFV Service Chains at the True Speed of the Underlying Hardware

Georgios P. Katsikas[1,3], Tom Barbette[2], Dejan Kostić[3], Rebecca Steinert[1], Gerald Q. Maguire Jr.[3]

[1]RISE SICS, [2]University of Liege, [3]KTH Royal Institute of Technology

Abstract

In this paper we present Metron, a Network Functions Virtualization (NFV) platform that achieves high resource utilization by jointly exploiting the underlying network and commodity servers' resources. This synergy allows Metron to: (*i*) offload part of the packet processing logic to the network, (*ii*) use smart tagging to setup and exploit the affinity of traffic classes, and (*iii*) use tag-based hardware dispatching to carry out the remaining packet processing at the speed of the servers' fastest cache(s), with zero inter-core communication. Metron also introduces a novel resource allocation scheme that minimizes the resource allocation overhead for large-scale NFV deployments. With commodity hardware assistance, Metron deeply inspects traffic at 40 Gbps and realizes stateful network functions at the speed of a 100 GbE network card on a single server. Metron has 2.75-6.5x better efficiency than OpenBox, a state of the art NFV system, while ensuring key requirements such as elasticity, fine-grained load balancing, and flexible traffic steering.

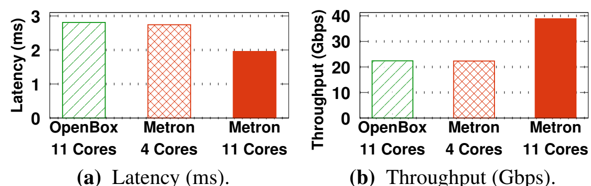
1 Introduction

Following the success of Software-Defined Networking (SDN), Network Functions Virtualization (NFV) is poised to dramatically change the way network services are deployed. NFV advocates running chains of network functions (NFs) implemented as software on top of commodity hardware. This is in contrast with chaining expensive, physical middleboxes, and brings numerous benefits: (*i*) decreased capital expenditure and operating costs for network service providers and (*ii*) facilitates the deployment of exciting new services.

Achieving high performance (high throughput and low latency with low variance) using commodity hardware is a hard problem. As 100 Gbps switches and network interface cards (NICs) are starting to be standardized and deployed, maintaining high performance at the ever-increasing data rates is vital for the success of NFV.

In an NFV service chain, packets move from one physical or virtual server (hereafter simply called server) to another to realize a programmable data plane. The servers themselves are predominantly multi-core machines. Different ways of structuring the NFs exist, e.g., one per physical core or using multiple threads to leverage multiple cores within each NF. Network functions range from simple stateless ones to complex, such as deep packet inspection (DPI), and potentially stateful (e.g., proxy) ones. Regardless of the deployment model and NF types, every time a packet enters a server, a fundamental problem occurs: how to locate the core within the multi-core machine that is responsible for handling this packet? This problem reoccurs every step of the chain and can cause costly inter-core transfers.

Our work, Metron, eliminates unnecessary inter-core transfers and in a 40-Gbps setup (Figure 1) achieves: (*i*) about a factor of 3 better efficiency, (*ii*) lower, predictable latency, and (*iii*) 2x higher throughput than OpenBox [13], a state of the art NFV system.



(a) Latency (ms). **(b)** Throughput (Gbps).

Figure 1: Thanks to zero inter-core transfers, Metron has almost 3x better efficiency than the state of the art when deeply inspecting (Firewall→DPI) traffic at 40 Gbps.

1.1 NFV Processing Challenges

To identify the core that will process an incoming packet, the NFV framework can typically only examine the header fields. Here, there is a big mismatch between the way modern servers are structured and the desired packet dispatching functionality. Figure 2 shows three widely used categories of packet processing models in NFV.

The first category (see Figure 2a), augments the weak programmability of current NICs with a software layer that acts as a programmable traffic dispatcher between the hardware and the overlay NFs. E2 [59], with its software component called SoftNIC [25], falls into this category. SoftNIC requires at least one dedicated CPU core for traffic dispatching and steering (see Figure 2a), while the NFs run on other CPU cores. Earlier works, such as ClickOS [49] and NetVM [29], also used software switches on dedicated cores to dispatch packets to virtual machines, but without the flexibility of E2.

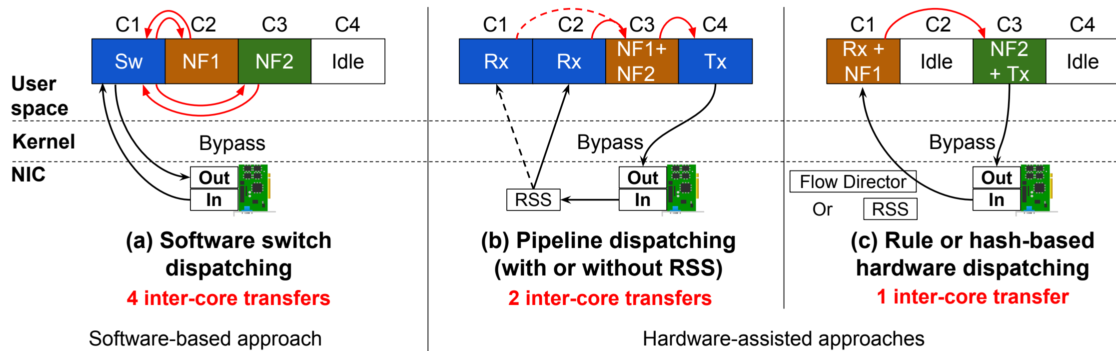


Figure 2: State of the art packet processing models either have too many inter-core packet transfers or load balancing problems due to load imbalance and/or idle CPU cores. RSS stands for Receive Side Scaling.

Rather than having a shim layer between the NFs and the NICs to select the next hop in a service chain, the second category of packet processing models (see Figure 2b) involves a pipeline of reception, processing, and transmission threads, each on a different (set of) core(s). If more than one reception core is required, this model uses RSS [30] as described below. For example, OpenNetVM [71], Flurries [70], and NFP [63] (a parallel version of OpenNetVM) fall into this category. Similar to E2, these works introduce programmability by augmenting the reception and processing parts of the pipeline with traffic steering abilities.

The last category of packet processing models relies on two hardware features provided by a large fraction of NIC vendors today. First, RSS uses a static function to dispatch traffic to a set of CPU cores by hashing the values of specific header fields. Second, NICs can be programmed via a vendor specific "match-action" API to dispatch traffic to specific cores (e.g., Intel's Flow Director [31]). Unlike all previous models, these approaches do not require dedicated dispatchers, hence they offer higher performance. OpenBox [1] [13], FastClick [6], SNF [36], and RouteBricks [18] use RSS, while CoMb [62] uses Flow Director.

None of these schemes guarantee that the core that receives the incoming packet will be the one processing it. Flow hashing as in RSS can introduce serious load imbalances under skewed (e.g., heavy flows with the same hashes) workloads. Flow Director permits explicit flow affinity, but suffers from the limited classification capabilities of today's commodity NICs. When there is a mismatch, the packet is handed off to the correct core. However, this requires transferring the packet via DRAM or last level cache (LLC) to the target processing core. This is a slow operation, as the LLC takes several tens of cycles even for a cache hit! Our earlier work [37] demonstrates that dramatic slowdowns occur due to this effect. In particular, an order of magnitude better performance (both higher throughput and lower latency variance) is possible if the correct core receives the packet straight from the NIC, and the packet remains in the core-specific L1 or L2 cache.

[1] Originally, OpenBox was built on top of Mininet and Click [42] using Linux-based I/O. To fairly compare it against our work, we accelerated OpenBox using FastClick's DPDK engine and RSS [4].

1.2 Metron Research Contributions

We present Metron, a system for NFV service chain placement and request dispatching. To the best of our knowledge, Metron is the first system that automatically and dynamically leverages the joint features of the network and server hardware to achieve high performance. Metron eliminates inter-core transfers (unlike recent work with 4 [59], 2 [70], or 1 [13] inter-core transfers as shown in Figure 2), making it possible to process packets potentially at L1 cache speeds. Also, we overcome the load balancing issues of "run-to-completion" approaches [18, 6, 13, 36], by combining smart identification, tagging, and dispatching techniques. We had to address a number of challenging problems to realize our vision. First, making efficient use of all the available hardware is hard because of the in-machine request dispatching overheads (described earlier). Second, discovering and dealing with the heterogeneous network (switches, NICs) and server hardware, in a generic way, is non-trivial from a management perspective. Third, detecting and dealing with load imbalances that reduce the performance of the initially placed service chains requires rapid and stable adaptation. We state our research contributions, while dealing with the aforementioned challenges:

Contribution 1: We orchestrate programmable network's hardware to perform stateless processing and packet classification. We deal with hardware heterogeneity by building upon the unified management abstractions of an industrial-grade SDN controller (ONOS [7]). This allows Metron to leverage popular management protocols, such as OpenFlow [50] and P4 [11], and easily integrate future ones. We contributed a new driver for programmable NICs and servers [35].

Contribution 2: We overcome the network/server architecture mismatch by instructing Metron to tag packets as early as possible, enabling them to be quickly and efficiently switched and dispatched throughout the entire chain. To do so, Metron first uses SNF [36] to identify the traffic classes of a service chain and produce a synthesized NF that performs the equivalent work of the entire chain (see §2.3.1). Then, Metron divides the synthesized NF into stateless and stateful operations (see §2.3.3) and instructs all available programmable hardware (i.e., switches and NICs) to implement the stateless operations, while dispatching incoming packets to those CPU cores that execute their stateful operations. Metron runs stateful NFs on general purpose servers, while fully leveraging their generic processing power.

Contribution 3: We propose a way to efficiently and quickly obtain the network state in order to make fast placement decisions at low cost with high accuracy (see §2.3.3). We devised a mechanism to coordinate load balancing among servers and their CPU cores, demonstrating that Metron provides comparable elasticity with purely software-based approaches, but at the true speed of the hardware (see §2.3.4).

Our evaluation shows that Metron realizes deep packet inspection at 40 Gbps (§3.1.1) and stateful service chains at the speed of a 100 GbE NIC on a single server (§3.1.2). This results in up to 4.7x lower latency, up to 7.8x higher throughput, and 2.75-6.5x better efficiency than the state of the art. It is difficult to improve on this performance unless we completely offload stateful chains to hardware, which is impossible with today's commodity hardware.

2 System Architecture

This section describes Metron's system design, starting with a high-level overview via an illustrative example in §2.1. In §2.2 we describe the Metron data plane, which is configured by the Metron controller (see §2.3).

2.1 Overview

To understand how Metron works, consider a simple network consisting of two OpenFlow switches connected to a server as shown at the bottom of Figure 3. Assume that an operator wants to deploy a Firewall→DPI service chain, as shown in Step 1 of Figure 3.

In Step 2, the Metron controller identifies the traffic classes [2] of the service chain, by parsing the packet processing graphs (each graph has a set of packet processing elements as in [42, 59, 13]) of the input NFs. In Step 3, Metron composes a single service chain-level graph by synthesizing the read and write operations of the individual graphs (see §2.3.1). Because Metron detects the availability of resources (i.e., the

[2] Traffic class is a (set of) flow(s) treated identically by an NF chain.

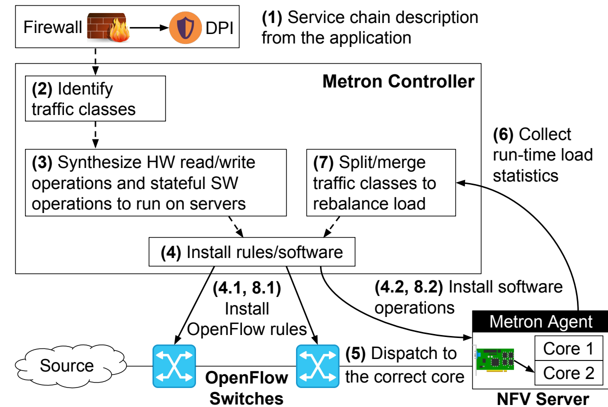


Figure 3: Metron overview using an example NF chain.

OpenFlow switches) along the path to the server, it associates stateless read and write operations with these components and automatically translates these operations into OpenFlow rules (Step 4.1). The remaining, potentially stateful, operations are translated into software instructions targeting the Metron agent at the server (Step 4.2). The key to Metron's high performance is exploiting hardware-based dispatching (Step 5) that annotates the traffic classes matched by the OpenFlow rules with tags that are subsequently matched by the server's NIC to identify the CPU core to execute the stateful operations. In this way, Metron guarantees that each traffic class will be processed by a single core, thus eliminating costly inter-core communications. This guarantee is maintained even when a CPU core becomes overloaded (see §2.3.4) as the Metron agent reports run-time statistics (Step 6) that allow the Metron controller to rebalance the load (Step 7) by splitting traffic classes into multiple groups that are dispatched to different cores using different tags (Steps 8.1 and 8.2). We conclude this overview with a survey of popular NFs; noting that in Table 1 a substantial portion of these NFs can be (fully or partially) offloaded to commodity hardware.

Table 1: Survey of popular NFs. The offloadability of "Hybrid" NFs depends on the use case.

Network Function	Offloadable to Hardware
L2/L3 Switch, Router	Yes
Firewall/Access Control List (ACL)	Hybrid
Carrier Grade NA(P)T, IPv4 to IPv6	No
Broadband Remote Access Server	Partially [17]
Evolved Packet Core	Partially
Intrusion Detection/Prevention	Partially [32]
Load Balancer	Hybrid
Flow Monitor	Yes
DDoS Detection/Prevention	Yes [43]
Congestion Control (RED, ECN)	Yes
Deep Packet Inspection (DPI)	No
IP Security, Virtual Private Network	Yes [61]

2.2 Metron Data Plane

The Metron data plane follows the master/slave approach depicted in Figure 4. The master process is an agent that interacts with (*i*) the underlying hardware by establishing bindings with key components, such as NICs, memory, and CPU cores and (*ii*) the Metron controller through a dedicated channel.

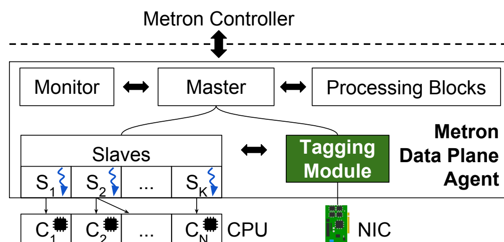


Figure 4: The Metron data plane.

The key differentiator between Metron and earlier NFV works is the tagging module shown in Figure 4. This module exposes a map with tag types and values that each NIC uses to interact with each CPU core of a server; this map is advertised to the Metron controller. The controller dynamically associates traffic classes with specific tags in order to enforce a specific flow affinity, thus controlling the distribution of the load. Most importantly, this traffic steering mechanism is applied by the hardware (i.e., NICs), hence Metron does not require additional CPU cores (as E2 does) to perform this task, thus packets are directly dispatched to the CPU core that executes their specific packet processing graph. In §3, we use a tagging scheme for trillions of service chains.

When the master boots, it configures the hardware and registers with the controller by advertising the server's available resources and tags. Then, the master waits for controller instructions. For example, the master executes a deployment instruction by spawning a slave process that is pinned to the requested core(s) and by passing the processing graph to the slave. In the context of service chaining, a Metron slave needs to execute multiple processing graphs, each corresponding to a different NF in the chain. Such graphs can be implemented either in hardware or software. Earlier works implement these graphs in software and use metadata to share information among NFs and to define the next hop in a chain. Although Metron supports this type of software-based chaining, as shown in §1.1, this approach introduces unnecessary overhead due to excessive inter-core communication and potentially under-utilizes the available hardware. Next, §2.3 explains how we approach and solve this problem.

2.3 Metron Control Plane

Here, we describe the key design choices and properties of the Metron controller.

2.3.1 Synthesis of Packet Processing Graphs

Given a set of input packet processing graphs, one per NF, Metron combines them into a single service chain graph. To ensure low latency, the Metron controller adopts SNF [36]; a more aggressive variant of OpenBox for merging packet processing graphs, which provides a heuristic for solving the graph embedding problem (see [68, 27, 15]) in the context of NFV. Metron uses SNF to eliminate processing redundancy by synthesizing those read and write operations that appear in a service chain as an optimized equivalent packet processing graph. SNF guarantees that each header field is read/written only once, as a packet traverses the graph.

Another benefit of SNF's integration into Metron is the ability to encode all the individual traffic classes of a service chain using a map of disjoint packet filters (Φ) to a set of operations (Ω). In §2.3.4 we use this feature to automatically scale packet processing in and out, providing greater elasticity than available today.

2.3.2 Initial Resource Allocation

To allocate resources for the synthesized graph, we allow application developers to input the CPU and network load requirements of their service chains. Alternatively, this information can be obtained by running a systematic NFV profiler, such as SCC [37], or by using more generic profilers, such as DProf [60]. Even in the absence of accurate resource requirements, Metron dynamically adapts to the input load as discussed in §2.3.4.

2.3.3 Placement

Metron needs to decide where to place the synthesized packet processing graph. Such a decision is not simple, because Metron not only considers servers but also the network elements along the path to these servers.

Table 1 showed that a large fraction of NFs cannot be implemented in commodity hardware today, mainly because they require maintaining state. This means, that the synthesized graph of such NFs cannot be completely offloaded. To solve this, we designed a graph separation module to traverse and split the synthesized graph into two subgraphs. The first subgraph contains the packet filters and operations that can be completely offloaded to the network (we call this a stateless subgraph), while the second (stateful) subgraph will be deployed on a server. The average complexity of this task is $O(\log m)$, where m is the number of vertices of the synthesized graph.

Given these two subgraphs, Metron needs to find a pair of nodes (a server and a network element) that satisfy two requirements: (*i*) the server has enough processing capacity to accommodate the stateful subgraph and (*ii*) the network element has enough capacity to store the hardware instructions (e.g., rules) that encode the stateless subgraph.

Scalable Placement with Minimal Overhead
In large networks with a large number of servers and switches, it is both expensive and risky to obtain load information from all the nodes. This is expensive because a large number of requests need to be sent frequently and this would occupy bandwidth to each node, generate costly interrupts to fetch the data, and occupy additional bandwidth to return responses to the controller. This is risky because the round-trip time required to obtain the monitoring data is likely to render this data stale, leading to herd behaviors and suboptimal decisions. To make a placement decision with minimal overhead, we use the simple, yet powerful, opportunistic scheme of "the power of two random choices" [52]. According to Mitzenmacher, this number offers exponentially better load balancing than a single random choice, while the additional gain of three random choices only corresponds to a constant factor.

Metron queries the load of two randomly selected servers and selects the least loaded of them, provided that the necessary resource requirements (i.e., number of NICs and CPU cores) can be met. If the first two choices fail, then these two servers are removed from the list and the process is repeated until a server is found. Note that this scheme prioritizes deployments that exhibit spatial correlation with respect to the processing location because spreading this processing results in lower performance, which is undesirable.

This server selection procedure also greatly simplifies the second placement decision (i.e., the network element(s) to offload processing to). Well designed networks, such as datacenters, provision several fixed shortest paths between ingress nodes (e.g., core switches) and servers, where each server might be associated with a single core switch [1, 2]. Given this, we find the most suitable network element to offload the stateless graph, using the following inputs: (*i*) the topology graph, (*ii*) the server where the stateful subgraph will be deployed (chosen by the server selection scheme), and (*iii*) the rule capacity required to offload the stateless subgraph.

Handling Partial Offloading and Rule Priorities
Metron carefully treats the cases when (*i*) a stateless subgraph contains rules with different priorities and (*ii*) one or more rules of such a subgraph cannot be offloaded to hardware. The latter can occur, e.g., due to the hardware's inability to match specific header fields. In such a case, Metron will selectively offload only the supported rules, while respecting rule priorities. To exemplify these two cases, assume a service chain that needs to be deployed on the topology shown in Figure 3. Assume that this service chain implements four rules that can be offloaded to the first programmable switch, while the remaining part of the service chain will be deployed on the server. If rule 3 cannot be offloaded and all of the rules have the same priority, then Metron will offload rules 1, 2, and 4. However, if these rules have, e.g., decreasing priorities (i.e., rule 3 has a higher priority than rule 4), then Metron will offload only the first two rules, to guarantee that the server applies rule 4 after rule 3.

2.3.4 Dynamic Scaling

In §2.3.1 we explained how Metron encodes a service chain as a set of individual traffic classes, where each traffic class is a set of packet filters mapped to write operations. This abstraction gives great flexibility when scaling a service chain in/out. As an example, when E2 detects an overloaded NF, it scales this NF by introducing an additional (duplicate) instance of the entire NF and then evenly splitting the flows across the two instances. In contrast, Metron splits the traffic classes of this NF across the two instances, such that each instance executes the code responsible for each of its traffic classes (rather than the code of the entire NF).

To trigger a scaling decision, Metron gathers port statistics from key locations in the network in order to detect load changes. Such a change results in Metron asking for instantaneous CPU load and network statistics from the affected service chains. Given this information, Metron applies the following, globally orchestrated, scaling strategy to react to load imbalances.

Traffic Class-level Scaling
We leverage a grouping technique when creating a service chain's traffic classes. A set of T traffic classes $\{TC_i^j \mid j \in [1,T]\}$ that belong to service chain i can be grouped together, if and only if their packet filters $\{\Phi_i^j \mid j \in [1,T]\}$ are mapped to the same write operations: $\forall k,l \in [1,T], \Omega_i^k = \Omega_i^l$

For example, an HTTP and an FTP traffic classes heading to a NAT will both exhibit the same stateful write operations from this NF, thus they can be grouped together. The Metron controller has this information available once the traffic classes of a service chain are created (see §2.3.1). To dynamically scale out a group of traffic classes, Metron needs to split this group into two or more subgroups, where the first subgroup remains on the same CPU core as the original group, while the other subgroup(s) are deployed and scheduled on a different (set of) CPU core(s). These new traffic classes are annotated with different tags, such that the NIC at the server can dispatch them to the appropriate CPU cores. We call this mechanism "traffic class deflation" to differentiate it from the opposite "traffic class inflation" process, where two or more groups of traffic classes that exhibit the same write operations are merged together, when Metron detects low CPU utilization.

To simplify load balancing, while keeping a reasonable degree of flexibility, the split and merge processes always use a static factor of 2 (i.e., one group is split into

two, or two groups are merged into one). This decision also minimizes the amount of state that Metron needs to transfer across CPUs. A fully dynamic solution with additional visibility into the load of each traffic class would achieve better load distribution; however, such a solution is considered impractical in the case of large networks with potentially millions of traffic classes. Split and merge operations may repeat until Metron can no longer split/merge a traffic class. A single flow is an example of non-splittable traffic class. The reaction time of this strategy is mainly affected by the time required for the controller to monitor and reconfigure the data plane. In §3.2 we show how this strategy performs in practice.

Once an inflation/deflation decision has been made, Metron needs to guarantee that the state of the affected traffic classes (e.g., those being redirected to a different CPU core in the case of inflation) will remain consistent. To do so we adopt a scheme that quickly duplicates the stateful tables of a group of traffic classes across the involved CPU cores, when inflation occurs. Similarly, we merge the stateful tables of two groups during the deflation process. Although this scheme introduces some redundancy (entries of migrated traffic classes will still occupy space in the memory of the previous CPU core until they expire), it offers a quick solution to a problem that is beyond the scope of this work. StateAlyzr [39], OpenNF [23], or the work by Olteanu and Raiciu [54] could be integrated into Metron to provide more efficient state management solutions. Alternatively, state management could be delegated to a remote distributed store as per Kablan, et al. [33].

2.3.5 Integrating Blackbox NFs

Some NF providers might not wish to disclose the source code of their NFs. In this case we offer two integration strategies: (i) partially synthesize a service chain, while using DPDK ring buffers to interconnect synthesized NFs with blackbox NFs and (ii) input only an NF configuration (e.g., DPI rules, omitting DPI logic) using Metron's high-level API and let Metron use its own data plane elements to realize this NF (see §3.1).

2.4 Routing (Updates) and Failures

To explain how Metron's routing and dispatching works and how Metron reacts to routing updates and failures, we use the example shown in Figure 5. We assume a software-defined[3] network on which the network operator has deployed a routing application that routes HTTP traffic[4] between source and destination (through the path s1→s3). The routing is done using the information shown within green dashed-dotted outlines.

[3] Metron can also operate in legacy networks by adding one or more programmable switches before the NFV servers.

[4] We assume only HTTP traffic to keep the example simple.

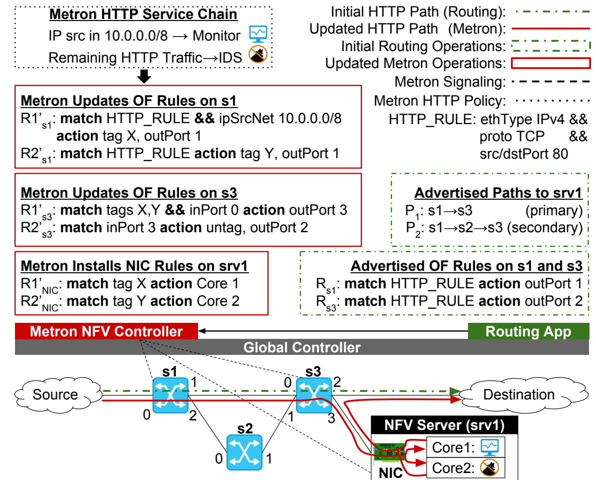


Figure 5: Metron's routing & CPU dispatching scheme.

A policy change forces the network operator to further process the HTTP traffic before it reaches its destination. Thus, she deploys an HTTP service chain (described by the top box with dotted outline in Figure 5) using Metron. When Metron boots it obtains the current routing policy and paths for the HTTP traffic, as advertised by the routing application. Next, the Metron controller performs a set of updates (see the left-side boxes with solid outlines, where OF stands for OpenFlow). The updates focus on two aspects: (*i*) to extend the existing HTTP rules (i.e., R_{s1} and R_{s3} at the bottom right box with dashed-dotted outline) with rules that also perform part of the service chain's operations (i.e., $R1'_{s1}$ and $R2'_{s1}$) and (*ii*) to tag the HTTP traffic classes to allow the NFV server to dispatch them to different CPU cores.

In this example, Metron identifies two traffic classes and tags them with tags X and Y. The tagging is applied by the first switch (i.e., s1 as explained in §2.3.3) using the rules $R1'_{s1}$ and $R2'_{s1}$ (top left box with solid outline). The next switch (s3) uses the tags (i.e., rule $R1'_{s3}$) to redirect the HTTP traffic classes to the NFV server, where Metron has installed NIC rules (i.e., $R1'_{NIC}$ and $R2'_{NIC}$) to dispatch packets with tags X and Y to CPU cores 1 and 2 respectively. The first core executes a monitoring NF, while the second core runs an intrusion detection system (IDS) NF. After traversing the service chain, the packets return to s3, where another Metron rule (i.e., $R2'_{s3}$) redirects them to their destination.

If not carefully addressed, a routing change or failure might introduce inconsistencies. Metron avoids these problems by using the paths to the NFV server (i.e., P_1 and P_2), as advertised by the routing application, to precompute: (*i*) alternative switches that can be used to offload part of a service chain's packet processing operations (see §2.3.3) and (*ii*) the actual rules to be installed in these switches. In this example, a routing change from path P_1 to P_2 (due to a routing update or

a link failure between s1 and s3) will result in Metron installing 2 additional rules in s2 (these rules follow same logic with the rules in s3). Metron also updates the first rule of s3 by changing the inPort value to 1 rather than 0.

Backup configurations are kept in Metron's distributed store and are replicated across all the Metron controller instances in order to maintain a global network view. When a routing change or failure occurs, Metron applies the appropriate backup configuration. In §3.3 we show that Metron can install 1000 rules in less than 200 ms, hence quickly adapting to routing changes and failures, even those requiring a large number of rule updates.

3 Evaluation

Implementation: We built the Metron controller on top of ONOS [7, 56], an open source, industrial-grade network operating system that is designed to scale well. Key to our decision was the fact that ONOS exposes unified abstractions for a large variety of network drivers that cover popular network configuration protocols, such as OpenFlow [50], P4 [11], NETCONF/YANG [20, 10], SNMP [14], and REST. We extended ONOS with a new driver that remotely monitors and configures NFV servers and their NICs. This driver is available at [35].

Metron's data plane extends FastClick [6]. We use the Virtual Machine Device Queues (VMDq) of DPDK [19] 17.08 to implement the hardware dispatching based on the values of the destination MAC address or VLAN ID fields. Our prototype (available at [5]) uses the former header field as a filter, because the large address space of a MAC address provides unique tags for trillions [5] of service chains. To scale to 100 Gbps, Metron instructs the hardware classifier of a Mellanox NIC (§3.1.2).

Testbed: Our testbed consists of 3 identical servers, each with a dual socket 16-core Intel® Xeon® CPU E5-2667 v3 clocked at 3.20 GHz. The cache sizes are: 2x32 KB L1 (instruction and data caches), 256 KB L2, and 20 MB L3. Hyper-threading is disabled and the OS is the Ubuntu 16.04.2 distribution with Linux kernel v.4.4. Each server has 2 dual-port 10 GbE Intel 82599 NICs.

We deploy a testbed with a NoviFlow 1132 OpenFlow switch [53] with firmware version NW400.2.2 and we attach 2 servers to this switch. The 4 ports of the first server are connected to the first 4 ports of the switch to inject traffic at 40 Gbps. Then, ports 5-8 of the switch are connected to the 4 ports of the second server, where traffic is processed by the NFV service chains being tested and sent back to the origin server through the switch. The last server is used to run the Metron controller. In §3.3, we study how switch diversity might affect Metron, by comparing the performance and capacity of a NoviFlow 1132 switch with an HP 5130 EI Switch [28] with software version S5130-3106, and the popular Open vSwitch [57] (OVS) software switch.

[5] A few thousands of tags were enough to conduct the study in §3.

Each experiment was conducted 10 times and we report the 10th, 50th (i.e., median), and 90th percentiles.

3.1 Metron Large-Scale Deployment

In this section we test Metron's performance at scale, focusing on two aspects: First, we stress Metron's data plane performance using complex service chains with a large number of deeply-inspected (§3.1.1) and stateful (§3.1.2) traffic classes at 40 and 100 Gbps respectively. In §3.1.3 we test Metron's placement on a set of topologies with a large number of nodes, on which we deploy hundreds to thousands of service chains.

3.1.1 Deep Packet Inspection at 40 Gbps

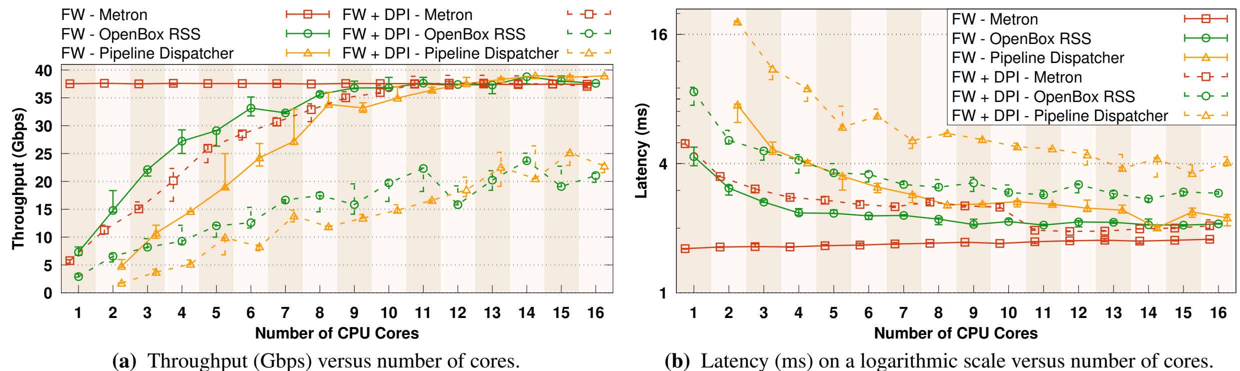
To test the overall system performance at scale, we deploy a service chain of a campus firewall, followed by a DPI. The firewall implements access control using a list of 1000 rules, derived from an actual campus trace. The output of the firewall is sent to a DPI NF that uses a set of regular expressions similar to Snort (see [13]).

We compare Metron against two state of the art systems: (*i*) an accelerated version of OpenBox based on RSS and (*ii*) an emulated version of E2. In the latter case, called "Pipeline Dispatcher", we emulate E2's SoftNIC by using a dedicated CPU core (i.e., core 1) that dispatches packets to the remaining CPU cores of the server (i.e., 2-16), where the NFs of the service chain are executed. This is the reason that the graphs of the emulated E2 in Figures 6 and 7 start from core two.

We injected a campus trace, obtained from University of Liège, that exercises all the rules of the firewall at 40 Gbps and measured the performance of the three approaches. Figure 6 visualizes the results. First, we deploy only the firewall NF of this service chain to quantify the overhead of running this NF in software, as compared to an offloaded firewall (i.e., Metron). To fairly compare Metron against the other two approaches, we start a simple forwarding NF in the server, such that all packets follow the exact same path (generator, switch, server, switch, and sink) in all three experiments.

Figure 6a shows that OpenBox and the emulated E2 can realize this large firewall at line-rate. However, this is only possible if more than half of the server's CPU cores are utilized. Specifically, OpenBox requires 9 cores, while the emulated E2 requires 11 cores. In contrast, Metron completely offloads the firewall to the switch, hence easily realizing its ACL at line-rate; thus one core of the server is enough to achieve maximum throughput.

Looking at the latency of the three approaches in Figure 6b, it is evident that software-based dispatching (yellow solid line with triangles) incurs a large amount of unnecessary latency. Hardware dispatching



(a) Throughput (Gbps) versus number of cores.

(b) Latency (ms) on a logarithmic scale versus number of cores.

Figure 6: Performance of a campus firewall with 1000 rules followed by a DPI at 40 Gbps, using: (*i*) Metron, (*ii*) an accelerated version of OpenBox using RSS, and (*iii*) a software-based dispatcher emulating E2.

using RSS (green solid line with circles) achieves substantially lower latency because it involves less inter-core communication. However, since the firewall executes heavy classification computations in software, OpenBox still exhibits high latency that cannot be decreased by simply increasing the number of cores. Specifically, using 16 cores has comparable latency to 4 cores. In contrast, Metron achieves nearly constant low latency (red solid line with squares) by exploiting the switch's ability to match a large number of rules at line-rate. This latency is 2.9-4.7x lower than the latency of the OpenBox and emulated E2 respectively, when each system uses one core for processing the NF (emulated E2 requires 2 cores in this case). At the full capacity of the server, the latency among the three systems is comparable; but Metron outperforms the emulated E2 and OpenBox by 30% and 19% respectively.

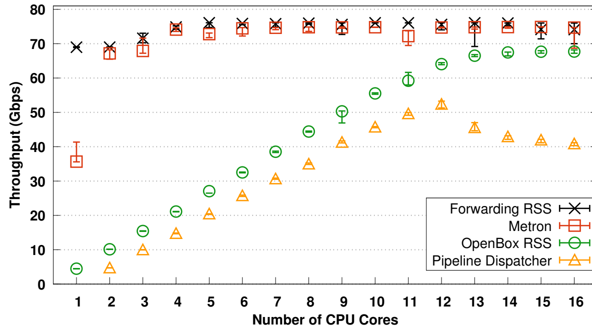
Next, we chain this firewall with a DPI NF in order to realize the entire service chain. This chaining further pushes the performance limits of the three approaches as shown by the dashed lines in Figure 6. In this case, Metron implements the DPI in software. First, we observe that even at the full capacity of the server, OpenBox and the emulated E2 can only achieve at most 25 Gbps. This performance is more than sufficient for a 10 Gbps deployment, hence some operators might not need the complex machinery of Metron. However, several studies indicate that large networks have already migrated from 10 to 40 Gbps deployments [16], while 100 Gbps networks are increasingly gaining traction [67]. In these higher data rate environments, these alternatives would require more than 16 CPU cores (and potentially more than one machine) to have sufficient throughput, and are not guaranteed to scale because of the heavy processing requirements of large service chains.

Metron exploits the joint network and server capacity to scale even complex NFs, such as DPI, at line-rate (red dashed line with squares in Figure 6a). Most importantly, Metron requires only 10 CPU cores in a single machine to achieve this result, thus substantially shifting the scaling point for large service chains. The latency results further highlight Metron's abilities. With 16 CPU cores, the Metron server deeply inspects all packets for this service chain at the cost of only 15.5% higher latency than the latency required to realize only the firewall. At the same time, OpenBox and the emulated E2 incur 35-97% more latency than Metron, while achieving almost half of Metron's throughput. This difference increases rapidly when fewer CPU cores are utilized. For example, when each system uses one CPU core Metron achieves 75% lower latency than OpenBox and 358% lower latency than the emulated E2 respectively.

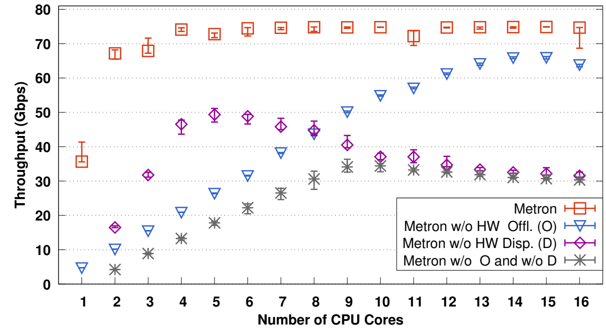
3.1.2 Stateful Service Chaining at 100 Gbps

In this section we further stress the performance of Metron, OpenBox, and the emulated E2 systems by conducting an experiment at 100 Gbps. To achieve this new performance target we use a different testbed. We equipped two of our servers with a 100 GbE Mellanox ConnectX-4 MT27700 card and connected them back-to-back. The first server acts as a traffic generator and receiver, while the second server is the device under test.

We analyzed 4 million packets from the campus trace used in §3.1.1 and found 3117 distinct destination IP addresses. Then, we implemented a standards-compliant router and populated its routing table with these addresses. The router was chained with two stateful NFs: a NAPT and a load balancer (LB) that implements a flow-based round robin policy. In this scenario, Metron can only offload the routing table of the router to the Mellanox NIC using DPDK's flow director. The remaining functions of the router (e.g., ARP handling, IP fragmentation, TTL decrement, etc.) together with the stateful NFs (i.e., NAPT and LB) are executed in software.



(a) Comparison of: (*i*) Metron, (*ii*) OpenBox with RSS, and (*iii*) a software-based dispatcher emulating E2. "Forwarding RSS" shows the forwarding speed of the server (i.e., no service chain).



(b) Metron's hardware offloading (O) and hardware dispatching (D) contributions to the overall system's performance. The word "without" is abbreviated as "w/o".

Figure 7: Throughput (Gbps) of a stateful service chain (Router→NAPT→LB) at 100 Gbps.

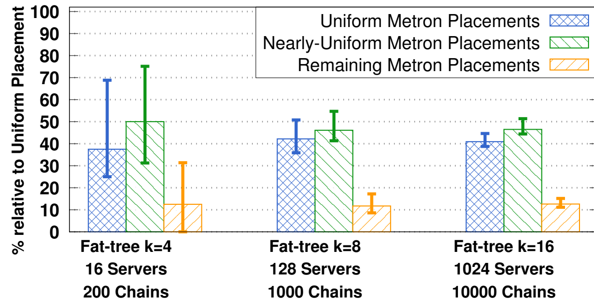
Metron vs. State of the art: The throughput achieved by the three systems is shown in Figure 7a. For comparison, we also show the throughput of the server when a simple RSS-assisted forwarding NF is used to send traffic back to its origin. These results show a slow but linear increase of the throughput with an increasing number of CPU cores for both OpenBox and the emulated E2 approaches. Using linear regression on the medians between 1 and 12 cores (the emulated E2 starts from 2 cores), we found that the throughput of OpenBox increases by 5.37 Gbps with each additional core, while the emulated E2 increased by 4.91 Gbps per core. However, in both cases using more than 12 CPU cores does not bring further performance gains. Specifically, the throughput of OpenBox plateaus around 67 Gbps, while the performance of the emulated E2 drops (from 53 to 41 Gbps). Moreover, with 13-16 cores, the latency of the two systems increases (up to 56% for OpenBox and up to 25% for the emulated E2); we omit the latency graph due to space limitations.

In contrast, Metron achieves 75 Gbps throughput using only a small fraction of the server's CPU cores. Key to this performance is Metron's hardware dispatcher in the NIC, which offers two advantages: (*i*) it saves CPU cycles by performing the lookup operations of the router and (*ii*) it load balances the traffic classes matched by the hardware classifier across the available CPU cores. Exploiting these advantages allows Metron (i.e., red squares in Figure 7) to quickly match the performance of the "Forwarding RSS" case (i.e., black points in Figure 7a) using only two cores, despite running several stateful operations (i.e., NAPT and LB). Moreover, according to a performance report by Mellanox [51], our NIC achieves line-rate throughput with frames greater than 512 bytes. Therefore, the 75 Gbps limit reached in this experiment with the campus trace is mainly due to the large number of small frames (26.9% of the frames are smaller than 100 bytes, while 11.8% of them are in (100, 500]). Finally, Metron's latency plateaus at a sub-millisecond level, which is 21-38% lower than the lowest latency achieved by the other two systems.

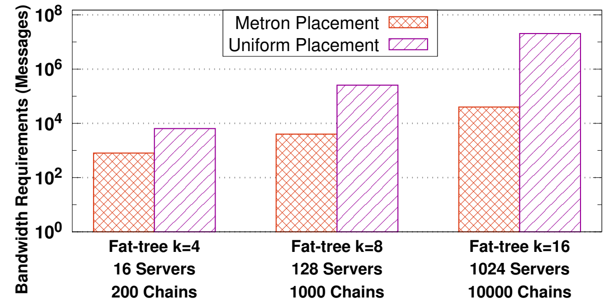
Dissecting Metron's Performance: To quantify the factors that contribute to Metron's high performance, we conducted an additional experiment using the same testbed, input trace, and service chain. The results are depicted in Figure 7b. Note that the red curves (i.e., Metron's throughput) in Figures 7a and 7b are identical. The purpose of Figure 7b is to showcase what performance penalties are expected when one starts removing our key contributions from Metron, as follows:

1. Metron without hardware offloading (i.e., blue triangles in Figure 7b). Hardware offloading corresponds to Contribution 1 in §1.2;
2. Metron without hardware dispatching to the correct core (purple rhombs in Figure 7b). Accurate dispatching corresponds to Contribution 2 in §1.2;
3. Metron without both of the two contributions (gray stars in Figure 7b).

Comparing "Metron" vs. "Metron w/o HW Offl." quantifies the benefits of Metron's hardware offloading feature. In the "Metron w/o HW Offl." case input packets are still dispatched to the correct core (using the Flow Director component of the Mellanox NIC), but each core executes the entire service chain logic in software. The throughput achieved in this case (i.e., blue triangles in Figure 7b) is comparable with the throughput of the "OpenBox RSS" case shown in Figure 7a. A key difference between these cases is that "Metron w/o HW Offl." performs the routing table lookup twice; once in the NIC for traffic dispatching and the second in software (to disable hardware offloading), after packets are dispatched to the correct core. In contrast, OpenBox uses RSS for dispatching and implements the routing table only once in software. Neither of these cases exploits the available capacity of the NIC to offload the routing operations, thus costing CPU cycles.



(a) Metron's placement relative to the uniform placement policy. Metron makes uniform or nearly uniform (with the least distance from uniform) placement decisions with ~90% median probability.



(b) Bandwidth requirements on a logarithmic scale with increasing number of servers and service chains. Metron requires orders of magnitude less bandwidth than the uniform placement policy.

Figure 8: Placement performance and bandwidth requirements on three fat-tree topologies of increasing number of servers (i.e., 16, 128, and 1024), when using (*i*) Metron or (*ii*) the uniform (equal number of CPU cores per server) placement scheme to deploy a large number of service chains.

Next, the comparison between "Metron" and "Metron w/o HW Disp." cases highlights the cost of inter-core communication. "Metron w/o HW Disp." implements the routing lookup in hardware (i.e., hardware offloading is enabled), hence reducing the processing requirements of the software part of the service chain. However, this case exhibits a serious bottleneck compared to Metron, as it requires a software component to (re-)classify input packets to decide which CPU core processes them (i.e., software dispatching similar to the emulated E2 case in Figure 7a). As shown in Figure 7, both "Metron w/o HW Disp." and the emulated E2 cases exhibit similar performance degradation as their software dispatcher communicates with an increasing number of CPU cores. This degradation appears earlier for "Metron w/o HW Disp." (i.e., after 5 cores versus 12 cores for the emulated E2 case). This is because "Metron w/o HW Disp." offloads part of the service chain's processing to the NIC, hence the inter-core communication bottleneck appears sooner. In contrast, Metron exploits the ability of the NIC to directly dispatch traffic to the correct core, thus avoiding the need for a software dispatcher and the concomitant inter-core communication.

Finally, the "Metron w/o O and w/o D" case in Figure 7b shows the throughput attainable when both hardware offloading and accurate dispatching features are disabled. In this case, input packets are always sent to an "incorrect" core (specifically the core where the software dispatcher runs) and the entire service chain runs in software. The inter-core communication bottleneck manifests itself once again, this time after using 9 or more cores.

Key Outcome: As explained in §2, Metron's ability to scale complex (i.e., DPI) and stateful (i.e., NAPT and LB) NFs is due to the way that the incoming traffic classes are identified, tagged, and dispatched to the CPU cores in a load balanced fashion. Metron's ability to realize these service chains at the NIC's hardware limit with a single server is an important achievement.

3.1.3 Metron's Placement in Large Networks

To verify that the performance of our placement scheme (see §2.3.3) can be generalized to real and potentially large networks, we conducted experiments that emulate Metron's service chain placement in datacenters, using fat-tree topologies of increasing sizes (see Figure 8). Our analytic study shows how close Metron's placement decisions are compared to uniform placement and what bandwidth requirements each approach demands for a large number of service chains. Note that the uniform placement allocates equal number of CPUs from the available servers, while a nearly uniform placement exhibits the least distance from the uniform. Note also that our approach is not restricted to datacenter topologies; Metron's placement is topology-agnostic.

Figure 8a compares Metron's placement with the uniform placement policies with increasing number of servers (i.e., 16, 128, and 1024) and service chains (i.e., 200, 1000, and 10000). The first of each set of bars indicate that Metron's placement decisions match the uniform ones with ~40% median probability, regardless of the network's size and number of service chains to be placed. For 16 servers, the upper percentile indicates that Metron makes a uniform decision with 70% probability. According to the other two sets of bars, most of the remaining decisions made by Metron fall very close to uniform (i.e., middle set of bars), confirming that our placement policy makes reasonably balanced decisions, despite its "limited" randomness.

Figure 8b shows the bandwidth savings of our placement policy, compared to the uniform one. To make a uniform placement decision, a controller has to query the CPU availability from all the available servers, thus, incurring a communication overhead proportional to the

network size (which quickly becomes infeasible for large networks). This overhead is shown by the second of each set of bars in Figure 8b. To reduce this overhead, we trade-off some accuracy in placement to minimize Metron's bandwidth requirements. The first of each set of bars in Figure 8b shows that Metron requires orders of magnitude less bandwidth than the uniform policy to place a large number of service chains on these networks. An indirect (but important) benefit of our low overhead placement is that, by querying only 2 servers at a time, we generate a minimal number of events at the servers, hence preserving processing cycles for other tasks.

3.2 Metron's Dynamic Scaling

Next, we evaluate Metron's dynamic scaling strategy (introduced in §2.3.4) using a scenario with a service chain configuration taken from an Internet Service Provider (ISP) [65], targeting a 10 Gbps network. The service chain consists of an ACL with 725 rules, followed by a NAPT gateway that interconnects the ISP with the Internet while performing source and destination address and port translation & routing.

We deployed this service chain on a single server connected to our switch, to which a real trace was injected at variable bitrates. The solid curve in Figure 9 shows the throughput corresponding to the rate at which the trace was injected, while the dashed curve depicts the throughput achieved by Metron. To highlight Metron's ability to provision resources on demand, we plot the number of cores allocated by Metron over the course of the experiment (yellow circles and right-hand scale).

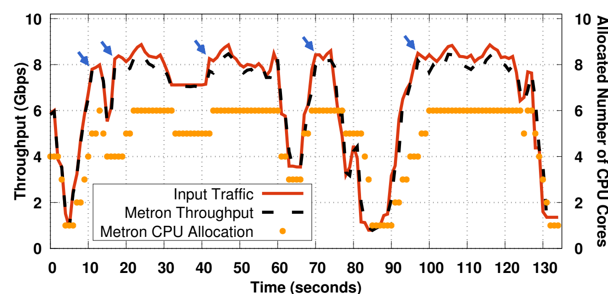


Figure 9: Metron under dynamic workload. Blue arrows indicate load spikes throughout the experiment.

The experiment begins with an allocation of 4 CPU cores (precalculated based upon the initial injection rate). Following this, the Metron controller makes dynamic decisions based on monitoring data gathered from the data plane and dynamically modifies the mapping of traffic classes to tags (thus affecting load distribution). In this experiment Metron requires between 1 and 6 CPU cores to accommodate the input traffic. In some cases, Metron fails to immediately adapt to sudden spikes, thus we observe a slight lag in Metron's reactions (e.g., as shown in the interval between 84 and 90 seconds). This is because our scaling approach involves interaction between the controller and the involved nodes (i.e., the server and the switch) in order to establish the CPU affinity of the traffic classes. To avoid overloading the controller, this interaction occurs every 500 ms, which contributes to the observed lag. However, Metron's throughput is not substantially affected by this lag (the blue arrows indicate the upward spikes in load at 10, 17, 42, 70, and 97 seconds). As we confirm in §3.3.2, Metron is able to quickly install the necessary rules to enforce the traffic class affinity.

3.3 Deployment Micro-benchmarks

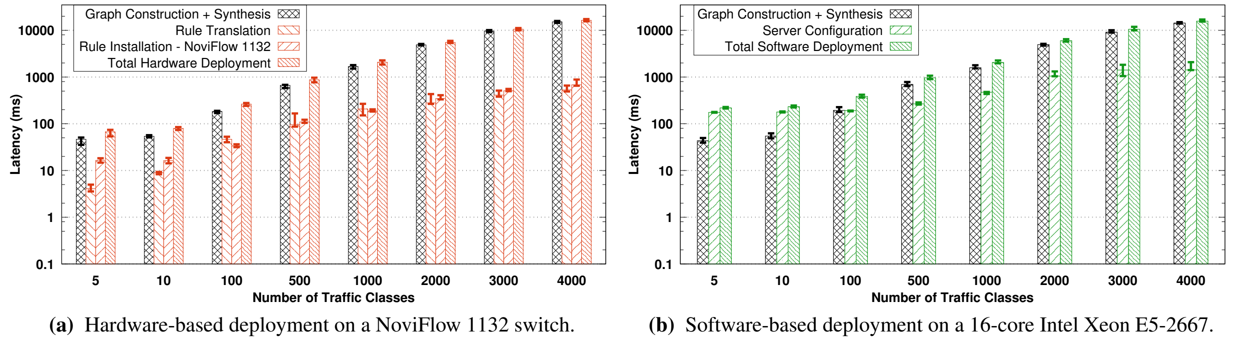
We benchmark how quickly Metron carries out important control and data plane tasks, such as hardware and software configuration, in a fully automated fashion.

3.3.1 Impact of Increasing # of Traffic Classes

To study the impact of increasingly complex service chains on Metron's deployment latency, we use a firewall with an increasing number of rules (up to 4000, derived from actual ISP firewalls [65]). We measure the time between when a request to deploy this NF is issued by an application and the actual NF deployment either in hardware or in software.

In either case, the first task of Metron is to construct and synthesize the packet processing graph of the service chain (as per §2.3.1), as depicted in the first of each group of bars (in black) in Figures 10a and 10b. This latency is the dominant latency in both hardware and software-based deployments (see the last set of bars in each figure). Fortunately, this is a one time overhead for each unique service chain; considering the importance of generating such an optimized processing graph, Metron precomputes and stores the synthesized graph for a given input in its distributed database.

Apart from this fixed latency operation, a purely hardware-based deployment, requires two additional operations, as shown in Figure 10a. The first operation is the automatic translation of the firewall's synthesized packet processing graph into hardware instructions targeting our OpenFlow switch (the second bar in each set of bars). This operation involves building a classification tree that encodes all the conditions of the firewall rules, therefore it has logarithmic complexity with the number of traffic classes. For example, under the specified experimental conditions, the median time to encode a large firewall with 4000 traffic classes is around 500 ms. The last operation in the hardware-based deployment is the rule installation in the OpenFlow switch (the third bar in each set of bars in Figure 10a). Note that even entry-level OpenFlow switches, such as the one used, can install thousands of rules per second;



(a) Hardware-based deployment on a NoviFlow 1132 switch.

(b) Software-based deployment on a 16-core Intel Xeon E5-2667.

Figure 10: Latency (ms) on a logarithmic scale for different Metron deployments with increasing complexity.

a more thorough study is provided in §3.3.2, where we discuss the effects of hardware diversity on Metron.

For a purely software-based deployment of this same service chain, we consider the time following graph construction and synthesis until the chain is deployed at a designated server. This latency is labeled "Server Configuration" in Figure 10b. Note that it takes longer per rule than for the corresponding hardware-based case for a small number of traffic classes because there is a fixed overhead to start a secondary DPDK process (i.e., a Metron slave) at the server. This overhead is ~180 ms as can be seen from the case of 5 traffic classes. However, the (median) deployment time is 0.471 ms/rule (versus 0.411 ms/rule for the hardware case shown in Table 2), hence a large firewall deployment takes a comparable amount of time either in software or hardware.

Overall, apart from the one-time precomputation overhead for constructing and synthesizing a service chain, the worst case deployment time of a firewall with 4000 traffic classes is less than 1200 ms, whereas only 100-200 ms is required for hundreds of traffic classes.

3.3.2 Diversity of Network Elements' Capabilities

Network elements from different vendors and of different price levels might offer different possibilities for NFV offloading. In this section we repeat the hardware-based deployment shown in Figure 10a, where we replace the NoviFlow switch with either a hybrid HP 5130 El hardware switch or the software-based OVS. Table 2 summarizes the results along with key characteristics of these switches, as they affect Metron's deployment choices and performance.

The NoviFlow switch contains 55 OpenFlow tables, each with a capacity of 4096 entries (i.e., 225280 rules in total), while the HP switch offers a single OpenFlow table with either 512/256 entries for IPv4/IPv6-based rules or 16384 entries for L2 rules. The capacity of OVS depends on the amount of memory that the host machine provides; modern servers provide ample DRAM capacity to store millions of rules.

The median rule installation speed of the NoviFlow switch is substantially higher than HP (0.411 vs. 50.25 ms/rule), with the difference being more than two orders of magnitude. However, this difference is partially reflected in the price difference between the two switches (approximately US$ 15000 vs. US$ 2000). OVS is open source, achieves lower data plane performance, but outperforms both hardware-based switches in terms of median rule installation speed (0.263 ms/rule), when running on the processor described for the testbed in §3. This finding is confirmed by earlier studies [45, 46], where the rule installation speed varied especially when priorities are involved. In our test, Metron installed rules of the same priority and we observed low variance.

In summary, today's OpenFlow switches provide Metron with fast median rule installation speed and sufficient capacity at different price/performance levels.

Table 2: Comparison of 3 switches used by Metron. The last column states their median rule installation speed.

Switch		Capacity	Speed
Model	**Type**	**(Rules)**	**(ms/rule)**
NoviFlow 1132 [53]	HW	225280	0.411
HP 5130 El [28]	HW	256/512 /16384	50.250
OVS [57] v2.5.2	SW	Memory -bound	0.263

4 Related Work

Here, we discuss related efforts beyond the work mentioned inline throughout this paper.

NFV Management: E2 [59] and Metron manage service chains mapped to clusters of servers interconnected via programmable switches. E2 only partially exploits OpenFlow switches to perform traffic steering. In contrast, Metron fully exploits the network (i.e., OpenFlow switches and NICs) to both steer traffic

and to offload and load balance NFV service chains, while deliberately avoiding E2's inter-core transfers.

NFV Consolidation: OpenBox [13] merges similar packet processing elements into one, thus reducing redundancy. SNF [36] eliminates processing redundancy by synthesizing multiple NFs as an optimized equivalent NF. Slick [3] and CoMb [62] propose NF consolidation schemes, although these schemes reside higher in the network stack. We integrated SNF into Metron, since this is the most extensive consolidation scheme to date. Metron effectively coordinates these optimized pipelines at a large-scale, while exploiting the hardware.

Hardware Programmability: During the last decade, there has been a large effort to increase hardware programmability. OpenFlow [50] paved the way by enriching the programmability of switches using simple match-action rules. Increasingly, NICs are equipped with hardware components, such as RSS and Flow Director, for dispatching packets from NIC to CPU core.

In an attempt to overcome the static nature of the above solutions, more flexible programmability models have emerged. RMT [12] and its successor P4 [11] are prime examples of protocol-independent packet processors, while OpenState [8] and OPP [9] showed how OpenFlow can become stateful with minimal but essential modifications. FlexNIC [38] proposed a model for additional programmability in future NICs.

All these works have made phenomenal progress towards exposing hardware configuration knobs. Metron acts as an umbrella to foster the integration of this diverse set of programmable devices into a common management plane. In fact, our prototype integrates OpenFlow switches, DPDK-compatible NICs, and servers. Thanks to ONOS's abstractions, additional network drivers can be easily integrated.

Hardware Offloading: Raumer et al. [61] offloaded the cryptographic function of a virtual private network (VPN) gateway into commodity NICs, for increased performance. SwitchKV [48] offloads key-value stores into OpenFlow switches. PacketShader [26], Kargus [32], NBA [41], and APUNet [24] take advantage of inexpensive but powerful graphical processing units to offload and accelerate packet processing. We envision these works as future components of Metron to extend its offloading abilities.

ClickNP [47] showed how to achieve high performance packet processing by completely migrating NFV into reconfigurable, but specialized hardware. In contrast, our philosophy is to explore the boundaries of commodity hardware. Therefore, Metron performs stateful processing in software but combines it with smart offloading using commodity hardware.

Server-level Solutions: Flurries [70] builds atop OpenNetVM [71] to provide software-based service chains on a per-flow basis, while ClickOS [49] and NetVM [29] offer NFs running in VMs. NFP [63] extends OpenNetVM to allow NFs in a service chain to be executed in parallel. Dysco [69] proposes a distributed protocol for steering traffic across the NFs of a service chain. NFVnice [44] and SCC [37, 34] are efficient NFV schedulers. Click-based [42] approaches have proposed techniques to exploit multi-core architectures [6, 64, 40]. None of these works have explored the possibility of using hardware to offload parts of a service chain, nor do they support our optimized flow affinity approach.

Industrial Efforts: European Telecommunications Standards Institute (ETSI) has been driving NFV standardization during the last 5 years [21]. ETSI's specialized group [22] uses OpenStack [58] as an open implementation of the current NFV standards, based on a generic framework for managing compute, storage, and network resources. CORD [55] and OPNFV [66] also use OpenStack. The former re-architects the central office as a datacenter, while the latter facilitates the interoperability of NFV components across various open source ecosystems. Metron and CORD share common controller abstractions (i.e., ONOS); however, we avoid OpenStack's virtualization by integrating native, DPDK-based solutions. Unlike CORD, our controller leverages placement techniques with minimal overhead (see §2.3.3 and §3.3) and sophisticated NF consolidation (see §2.3.1) to achieve high performance.

5 Conclusion

We have presented Metron, an NFV platform that fundamentally changes how service chains are realized. Metron eliminates the need for costly inter-core communication at the servers by delegating packet processing and CPU core dispatching operations to programmable hardware devices. Doing so offers dramatic hardware efficiency and performance increases over the state of the art. With commodity hardware assistance, Metron fully exploits the processing capacity of a single server, to deeply inspect traffic at 40 Gbps and execute stateful service chains at the speed of a 100 GbE NIC.

6 Acknowledgments

We would like to thank our shepherd Vyas Sekar and the anonymous reviewers for their insightful comments on this paper. This work is financially supported by the Swedish Foundation for Strategic Research. In addition, this work was partially supported by the Wallenberg Autonomous Systems Program (WASP).

References

[1] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication (2008), pp. 63–74.

[2] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (2010), NSDI'10.

[3] ANWER, B., BENSON, T., FEAMSTER, N., AND LEVIN, D. Programming Slick Network Functions. In Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (2015), SOSR '15, pp. 14:1–14:13.

[4] BARBETTE, T. Repository with DPDK extensions for OpenBox, 2018. https://github.com/tbarbette/fastclick/tree/openbox.

[5] BARBETTE, T., AND KATSIKAS, G. P. Metron data plane, 2018. https://github.com/tbarbette/fastclick/tree/metron.

[6] BARBETTE, T., SOLDANI, C., AND MATHY, L. Fast Userspace Packet Processing. In Proceedings of the 11th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (2015), ANCS '15, IEEE Computer Society. http://dl.acm.org/citation.cfm?id=2772722.2772727.

[7] BERDE, P., GEROLA, M., HART, J., HIGUCHI, Y., KOBAYASHI, M., KOIDE, T., LANTZ, B., O'CONNOR, B., RADOSLAVOV, P., SNOW, W., AND PARULKAR, G. ONOS: Towards an Open, Distributed SDN OS. In Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (2014), HotSDN '14, pp. 1–6.

[8] BIANCHI, G., BONOLA, M., CAPONE, A., AND CASCONE, C. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. SIGCOMM Comput. Commun. Rev. (2014).

[9] BIANCHI, G., BONOLA, M., PONTARELLI, S., SANVITO, D., CAPONE, A., AND CASCONE, C. Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing. arXiv preprint arXiv:1605.01977 (2016).

[10] BJORKLUND, M. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). Internet Request for Comments (RFC) 6020 (Proposed Standard), Oct. 2010. https://www.rfc-editor.org/rfc/rfc6020.txt.

[11] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-independent Packet Processors. SIGCOMM Comput. Commun. Rev. 44, 3 (July 2014), 87–95.

[12] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In Proceedings of the ACM SIGCOMM 2013 Conference on (2013), pp. 99–110.

[13] BREMLER-BARR, A., HARCHOL, Y., AND HAY, D. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference (2016), SIGCOMM '16, pp. 511–524.

[14] CASE, J., FEDOR, M., SCHOFFSTALL, M. L., AND DAVIN, J. Simple Network Management Protocol (SNMP). Internet Request for Comments (RFC) 1157, May 1990. http://www.ietf.org/rfc/rfc1157.txt.

[15] CHOWDHURY, M., RAHMAN, M. R., AND BOUTABA, R. ViNEYard: Virtual Network Embedding Algorithms with Coordinated Node and Link Mapping. IEEE/ACM Trans. Netw. 20, 1 (Feb. 2012), 206–219.

[16] CISCO. Migrate to a 40-Gbps Data Center with Cisco QSFP BiDi Technology, 2013. http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-729493.html.

[17] DIETZ, T., BIFULCO, R., MANCO, F., MARTINS, J., KOLBE, H., AND HUICI, F. Enhancing the BRAS through virtualization. In Proceedings of the 1st IEEE Conference on Network Softwarization, NetSoft 2015 (2015), pp. 1–5.

[18] 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 ACM SIGOPS 22nd Symposium on Operating Systems Principles (2009), SOSP '09, pp. 15–28.

[19] DPDK. Data Plane Development Kit, 2018. http://dpdk.org.

[20] ENNS, R., BJORKLUND, M., SCHOENWAELDER, J., AND BIERMAN, A. Network Configuration Protocol (NETCONF). Internet Request for Comments (RFC) 6241 (Proposed Standard), June 2011. Updated by RFC 7803, https://www.rfc-editor.org/rfc/rfc6241.txt.

[21] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. Network Functions Virtualisation, 2017. http://www.etsi.org/technologies-clusters/technologies/689-network-functions-virtualisation.

[22] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE (ETSI). Open Source NFV Management and Orchestration (MANO) , 2018. https://osm.etsi.org/.

[23] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. OpenNF: Enabling Innovation in Network Function Control. In Proceedings of the 2014 ACM Conference on SIGCOMM (2014), SIGCOMM '14, pp. 163–174.

[24] GO, Y., ASIM JAMSHED, M., MOON, Y., HWANG, C., AND PARK, K. APUNet: Revitalizing GPU as Packet Processing Accelerator. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17) (2017), USENIX Association, pp. 83–96.

[25] HAN, S., JANG, K., PANDA, A., PALKAR, S., HAN, D., AND RATNASAMY, S. SoftNIC: A Software NIC to Augment Hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[26] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: A GPU-accelerated Software Router. In Proceedings of the ACM SIGCOMM 2010 Conference (2010), SIGCOMM '10, pp. 195–206.

[27] HE, J., ZHANG-SHEN, R., LI, Y., LEE, C.-Y., REXFORD, J., AND CHIANG, M. DaVinci: Dynamically Adaptive Virtual Networks for a Customized Internet. In Proceedings of the 2008 ACM CoNEXT Conference (New York, NY, USA, 2008), CoNEXT '08, ACM, pp. 15:1–15:12.

[28] HEWLETT PACKARD. HPE FlexNetwork 5130 EI Switch Series, Jan. 2017. https://h50146.www5.hpe.com/products/networking/datasheet/HP_5130EI_Switch_Series_J.pdf.

[29] HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (2014), NSDI'14, pp. 445–458.

[30] INTEL. Receive-Side Scaling (RSS), 2007. http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf.

[31] INTEL. Ethernet Flow Director, 2018. http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-flow-director-video.html.

[32] 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), CCS '12.

[33] KABLAN, M., ALSUDAIS, A., KELLER, E., AND LE, F. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17) (2017), pp. 97–112.

[34] KATSIKAS, G. P. Realizing High Performance NFV Service Chains. Licentiate Thesis (Nov. 2016). TRITA-ICT 2016:35, http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1044355&dswid=-1520.

[35] KATSIKAS, G. P. Metron controller's southbound driver for managing commodity servers, 2018. https://github.com/gkatsikas/onos/tree/metron-driver.

[36] KATSIKAS, G. P., ENGUEHARD, M., KUŹNIAR, M., MAGUIRE JR., G. Q., AND KOSTIĆ, D. SNF: Synthesizing high performance NFV service chains. PeerJ Computer Science 2 (Nov. 2016), e98. http://dx.doi.org/10.7717/peerj-cs.98.

[37] KATSIKAS, G. P., MAGUIRE JR., G. Q., AND KOSTIĆ, D. Profiling and accelerating commodity NFV service chains with SCC. Journal of Systems and Software 127C (Feb. 2017), 12–27. https://doi.org/10.1016/j.jss.2017.01.005.

[38] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High Performance Packet Processing with FlexNIC. In Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (2016), ASPLOS '16, pp. 67–81.

[39] KHALID, J., GEMBER-JACOBSON, A., MICHAEL, R., ABHASHKUMAR, A., AND AKELLA, A. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (2016), NSDI'16, USENIX Association, pp. 239–253.

[40] KIM, J., HUH, S., JANG, K., PARK, K., AND MOON, S. The Power of Batching in the Click Modular Router. In Proceedings of the Asia-Pacific Workshop on Systems (2012), APSYS '12, pp. 14:1–14:6.

[41] KIM, J., JANG, K., LEE, K., MA, S., SHIM, J., AND MOON, S. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In Proceedings of the 10th European Conference on Computer Systems (2015), EuroSys '15.

[42] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. ACM Trans. Comput. Syst. 18, 3 (Aug. 2000), 263–297.

[43] KRISHNAN, R., DURRANI, M., AND PHAAL, P. Real-time SDN Analytics for DDoS mitigation, 2014.

[44] KULKARNI, S. G., ZHANG, W., HWANG, J., RAJAGOPALAN, S., RAMAKRISHNAN, K. K., WOOD, T., ARUMAITHURAI, M., AND FU, X. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 71–84.

[45] KUŹNIAR, M., PEREŠÍNI, P., AND KOSTIĆ, D. What You Need to Know About SDN Flow Tables. In Passive and Active Measurement (PAM) (2015), vol. 8995 of Lecture Notes in Computer Science, pp. 347–359. https://doi.org/10.1007/978-3-319-15509-8_26.

[46] KUŹNIAR, M., PEREŠÍNI, P., KOSTIĆ, D., AND CANINI, M. Methodology, Measurement and Analysis of Flow Table Update Characteristics in Hardware OpenFlow Switches. Computer Networks: The International Journal of Computer and Telecommunications Networking, Elsevier, vol. 26 (2018). https://doi.org/10.1016/j.comnet.2018.02.014.

[47] LI, B., TAN, K., LUO, L. L., PENG, Y., LUO, R., XU, N., XIONG, Y., CHENG, P., AND CHEN, E. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In Proceedings of the 2016 ACM SIGCOMM Conference (2016), SIGCOMM '16, pp. 1–14.

[48] LI, X., SETHI, R., KAMINSKY, M., ANDERSEN, D. G., AND FREEDMAN, M. J. Be Fast, Cheap and in Control with SwitchKV. In Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (2016), NSDI'16, USENIX Association, pp. 31–44.

[49] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the Art of Network Function Virtualization. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (2014), NSDI'14, pp. 459–473.

[50] 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.

[51] MELLANOX TECHNOLOGIES. Mellanox NIC's Performance Report with DPDK 17.05, 2017. Document number MLNX-15-52365, Revision 1.0, 2017, http://fast.dpdk.org/doc/perf/DPDK_17_05_Mellanox_NIC_performance_report.pdf.

[52] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. IEEE Trans. Parallel Distrib. Syst. 12, 10 (Oct. 2001), 1094–1104.

[53] NOVIFLOW. NoviSwitch 1132 High Performance OpenFlow Switch, 2013. https://noviflow.com/wp-content/uploads/NoviSwitch-1132-Datasheet-V2_1.pdf.

[54] OLTEANU, V. A., AND RAICIU, C. Efficiently Migrating Stateful Middleboxes. In Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (2012), SIGCOMM '12, ACM, pp. 93–94.

[55] ON.LAB. Central Office Re-architected as a Datacenter (CORD), 2018. http://opencord.org/.

[56] ON.LAB. Open Network Operating System (ONOS), 2018. http://onosproject.org/.

[57] OPEN VSWITCH. An Open Virtual Switch, 2018. http://openvswitch.org.

[58] OPENSTACK. Open Source Cloud Computing Software, 2018. https://www.openstack.org/.

[59] PALKAR, S., LAN, C., HAN, S., JANG, K., PANDA, A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. E2: A Framework for NFV Applications. In Proceedings of the 25th Symposium on Operating Systems Principles (2015), SOSP '15, pp. 121–136.

[60] PESTEREV, A., ZELDOVICH, N., AND MORRIS, R. T. Locating Cache Performance Bottlenecks Using Data Profiling. In Proceedings of the 5th European Conference on Computer Systems (2010), EuroSys '10, pp. 335–348.

[61] RAUMER, D., GALLENMÜLLER, S., EMMERICH, P., MÄRDIAN, L., WOHLFART, F., AND CARLE, G. Efficient serving of VPN endpoints on COTS server hardware. In 2016 IEEE 5th International Conference on Cloud Networking (CloudNet'16) (Pisa, Italy, Oct. 2016).

[62] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M. K., 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), NSDI'12.

[63] SUN, C., BI, J., ZHENG, Z., YU, H., AND HU, H. NFP: Enabling Network Function Parallelism in NFV. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 43–56.

[64] SUN, W., AND RICCI, R. Fast and Flexible: Parallel Packet Processing with GPUs and Click. In Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (Piscataway, NJ, USA, 2013), ANCS '13, IEEE Press, pp. 25–36.

[65] TAYLOR, D. E., AND TURNER, J. S. ClassBench: A Packet Classification Benchmark. IEEE/ACM Trans. Netw. 15, 3 (June 2007), 499–511.

[66] THE LINUX FOUNDATION. Open Platform for NFV (OPNFV), 2018. https://www.opnfv.org/.

[67] VIEJO, A. QLogic and Broadcom First to Demonstrate End-to-End Interoperability for 25Gb and 100Gb Ethernet, 2015. https://globenewswire.com/news-release/2015/01/27/700249/10116850/en/QLogic-and-Broadcom-First-to-Demonstrate-End-to-End-Interoperability-for-25Gb-and-100Gb-Ethernet.html.

[68] YU, M., YI, Y., REXFORD, J., AND CHIANG, M. Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration. SIGCOMM Comput. Commun. Rev. 38, 2 (Mar. 2008), 17–29.

[69] ZAVE, P., FERREIRA, R. A., ZOU, X. K., MORIMOTO, M., AND REXFORD, J. Dynamic Service Chaining with Dysco. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 57–70.

[70] ZHANG, W., HWANG, J., RAJAGOPALAN, S., RAMAKRISHNAN, K., AND WOOD, T. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In Proceedings of the 12th ACM International Conference on Emerging Networking Experiments and Technologies (2016), CoNEXT '16, pp. 3–17.

[71] ZHANG, W., LIU, G., ZHANG, W., SHAH, N., LOPREIATO, P., TODESCHI, G., RAMAKRISHNAN, K., AND WOOD, T. OpenNetVM: A Platform for High Performance Network Service Chains. In Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (August 2016), ACM.

G-NET: Effective GPU Sharing in NFV Systems

Kai Zhang*, Bingsheng He‡, Jiayu Hu†, Zeke Wang‡, Bei Hua†, Jiayi Meng†, Lishan Yang†
*Fudan University ‡National University of Singapore
†University of Science and Technology of China

Abstract

Network Function Virtualization (NFV) virtualizes software network functions to offer flexibility in their design, management and deployment. Although GPUs have demonstrated their power in significantly accelerating network functions, they have not been effectively integrated into NFV systems for the following reasons. First, GPUs are severely underutilized in NFV systems with existing GPU virtualization approaches. Second, data isolation in the GPU memory is not guaranteed. Third, building an efficient network function on CPU-GPU architectures demands huge development efforts.

In this paper, we propose G-NET, an NFV system with a GPU virtualization scheme that supports spatial GPU sharing, a service chain based GPU scheduler, and a scheme to guarantee data isolation in the GPU. We also develop an abstraction for building efficient network functions on G-NET, which significantly reduces development efforts. With our proposed design, G-NET enhances overall throughput by up to 70.8% and reduces the latency by up to 44.3%, in comparison with existing GPU virtualization solutions.

1 Introduction

Network Function Virtualization is a network architecture for virtualizing the entire class of network functions (NFs) on commodity off-the-shelf general-purpose hardware. Studies show that NFs constitute 40–60% of the appliances deployed in large-scale networks [36]. This architecture revolutionized the deployment of middleboxes for its lower cost, higher flexibility, and higher scalability. With the fast increasing data volume, the networking speed is also under rapid growth to meet the demand for fast data transfer. Therefore, achieving high performance is a critical requirement for NFV systems.

With the massive number of cores and high memory bandwidth, GPUs are well known for the capability of significantly accelerating NFs. Existing GPU-based NFs include router [15], SSL proxy [20], SRTP proxy [47], OpenFlow switch and IPsec gateway [15]. By forming CPU and GPU processing in a pipeline, the heterogeneous architecture is capable of delivering a high throughput in packet processing. Moreover, due to the rise of deep learning and other data analytical applications, GPUs are widely deployed in data centers and cloud services, e.g., Amazon EC2 GPU instance [2] and Alibaba Cloud GPU server [1]. Therefore, GPUs serve as a good candidate for building high-performance NFV systems. However, GPUs still have not been widely and effectively adopted in NFV systems. We identify the main reasons as threefold.

GPU Underutilization: Although state-of-the-art GPU virtualization techniques [39, 40, 45] enable multiple VMs to utilize a GPU, a GPU can only be accessed by a VM exclusively at a time, i.e., temporal sharing. Consequently, VMs have to access the GPU in a round-robin fashion. These virtualization approaches fit for GPU kernels that can fully utilize the GPU, such as deep learning [46] and database queries [41]. In production systems such as cloud, the input network traffic volume of an NF is generally much lower than the throughput that a GPU can achieve. As a result, the workload of each kernel in NFV systems is much lighter, which would result in severe GPU underutilization. Batching more tasks can be a feasible way to improve the GPU utilization, but it would result in a much higher latency. This issue largely blocks the adoption of GPUs in NFV systems as the overall throughput may be not enhanced or even degraded.

Lack of Support for Data Isolation: In a GPU-accelerated NFV system, both packets and the data structures of NFs need to be transferred to the GPU memory for GPU processing. When multiple NFs utilize a GPU to accelerate packet processing, they may suffer from information leakage due to the vulnerabilities in current GPU architectures [33]. As a result, a malicious NF may eavesdrop the packets in the GPU memory or even ma-

nipulate traffic of other NFs. As security is one of the main requirements in NFV systems [16], the lack of the system support for data isolation in the GPU memory may cause concern for NFV users.

Demanding Significant Development Efforts: Building an efficient network function on heterogeneous CPU-GPU architectures demands lots of development efforts. First, the complex data flow in network processing should be handled, including I/O operations, task batching, and data transferring in the CPU-GPU pipeline. Second, achieving the throughput and latency requirements needs to carefully adjust several parameters, such as the GPU batch size and the number of CPU and GPU threads. These parameters are highly relevant with the hardware and workloads, which makes it hard for NF deployment. All of the above efforts are repetitive and time-consuming in NF development and deployment.

In this paper, we propose an NFV system called *G-NET* to address the above issues and support efficient executions of NFs on GPUs. The main idea of G-NET is to spatially share a GPU among multiple NF instances to enhance the overall system efficiency. To achieve this goal, G-NET takes a holistic approach that encompasses all aspects of GPU processing, including the CPU-GPU processing pipeline, the GPU resource allocation, the GPU kernel scheduling, and the GPU virtualization. The proposed system not only achieves high efficiency but also lightens the programming efforts. The main contributions of this paper are as follows.

- A GPU-based NFV system, G-NET, that enables NFs to effectively utilize GPUs with spatial sharing.
- A GPU scheduling scheme that aims at maximizing the overall throughput of a service chain.
- A data isolation scheme to guarantee the data security in the GPU memory with both compile and runtime check.
- An abstraction for building NFs, which significantly reduces the development and deployment efforts.

Through experiments with a wide range of workloads, we demonstrate that G-NET is capable of enhancing the throughput of a service chain by up to 70.8% and reducing the latency by up to 44.3%, in comparison with the temporal GPU sharing virtualization approach.

The roadmap of this paper is as follows. Section 2 introduces the background of this research. Section 3 outlines the overall structure of G-NET. Section 4 describes the virtualization scheme and data plane designs of G-NET. Sections 5 and 6 describe the scheduling scheme and the abstraction for NF development. Section 7 evaluates the prototype system, Section 8 discusses related work, and Section 9 concludes the paper.

2 Background and Challenges

In this section, we review the background of adopting GPUs in NFV systems and discuss the major challenges in building a highly-efficient system.

2.1 Network Functions on Heterogeneous CPU-GPU Architectures

GPUs are efficient at network processing because the massive number of incoming packets offers sufficient parallelism. Since CPUs and GPUs have different architectural characteristics, they generally work in a pipelined fashion to execute specific tasks for high efficiency [15, 48]. CPUs are usually in charge of performing I/O operations, batching, and packet parsing. The compute/memory-intensive tasks are offloaded to GPUs for acceleration, such as cryptographic operations [20], deep packet inspection [19], and regular expression matching[42].

Take software router as an example, the data processing flow is as follows. First, the CPU receives packets from NICs, parses packet headers, extracts IP addresses, and batches them in an input buffer. When a specified batch size or a preset time limit is reached, the input buffer is transferred to the GPU memory via PCIe, then a GPU kernel is launched to lookup the IP addresses. After the kernel completes processing, the GPU results, i.e., the NIC ports to be forwarded to, are transferred back to the host memory. Based on the results, the CPU sends out the packets in the batch.

A recent CPU optimization approach G-Opt [21] achieves compatible performance with GPU-based implementations. G-Opt utilizes group prefetching and software pipelining to hide memory access latencies. Comparing with GPU-based implementations, such optimizations are time-consuming to apply, and they increase the difficulty in reading and maintaining the code. Moreover, the optimizations have limited impact on compute-intensive NFs [12], and the performance benefits may depend on the degree of cache contention when running concurrently with other processes.

In the following of this paper, we use NVIDIA CUDA terminology in the GPU related techniques, which are also applicable to OpenCL and GPUs from other vendors.

2.2 GPU Virtualization in NFV Systems

We implement four NFs on CPU-GPU architectures, including an L3 Router, a Firewall, a Network Intrusion Detection System (NIDS), and an IPsec gateway. The implementation follows the state-of-the-art network functions [19, 20, 15], where the GPU kernels are listed

NF	Kernel algorithm
Router	DIR-24-8-BASIC [13]
Firewall	Bit vector linear search [24] (188 rules)
NIDS	Aho-Corasick algorithm [4] (147 rules)
IPsec	AES-128 (CTR mode) and HMAC-SHA1

Table 1: GPU kernel algorithms of network functions.

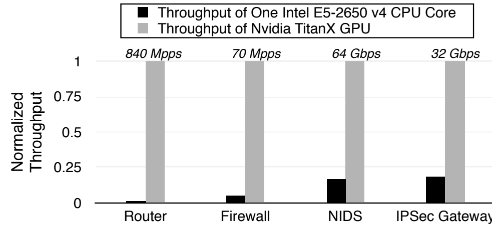


Figure 1: The throughputs of a CPU core and a GPU when forming a pipeline in GPU-accerated NFs (512-byte packet).

in Table 1. In the implementation of the Firewall, bit vector linear search is used to perform fast packet classification with multiple packet fields. For each field, we use trie [8] for IP address lookup, interval tree [7] for matching port range, and hash table for the protocol lookup. Based on the implementations, we conduct several experiments to evaluate the network functions and make the following observations. Please refer to Sect. 7.1 for the hardware and system configurations.

GPU is underutilized: In Figure 1, we show the throughput of the four GPU kernels, where a GPU demonstrates 840 Mpps for the L3 router, 70 Mpps for the Firewall, 64 Gbps for the NIDS, and 32 Gbps for the IPsec gateway. In cloud or enterprise networks, however, the traffic volume of an NF instance can be significantly lower than the GPU throughput. Consequently, such high performance can be overprovision for many production systems. Figure 1 also makes a comparison of the normalized throughput between a CPU core and a GPU when they form a pipeline in the NFs, where the CPU core performs packet I/O and batching, and the GPU performs the corresponding operations in Table 1. As shown in the figure, the throughput of a CPU core is significantly lower (5×-65×) than that of the GPU. As a result, being allocated with only limited number of CPU cores, an NF is unable to fully utilize a GPU. For the above reasons, a GPU can be severely underutilized in cloud or enterprise networks.

Temporal sharing leads to high latency: When only one NF runs on a server, the GPU is exclusively accessed by the NF. By overlapping the CPU processing and the GPU processing, the GPU timeline is shown in Figure 2(1). With the adoption of virtualization techniques [39, 40, 45] that enable temporal GPU sharing, NFs are able to access the GPU in a round-robin fashion. To maintain the same throughput with exclusive access, the GPU processing time of each NF should be reduced to allow multiple kernels utilizing the GPU. One may expect to utilize more GPU resources to reduce the GPU processing time. In Figure 3, we show that the GPU processing time quickly converges with first few Streaming Multiprocessors (SMs) allocated, and the reduction is moderate or even unnoticeable with more SMs. This is because that, with less than a certain number of jobs, an SM has a relatively fixed processing time in handling a specific task. As a result, although the batch size of an SM becomes smaller by assigning more SMs, the overall processing time cannot be further reduced. Consequently, temporal GPU sharing would not enhance the throughput of NFs, but the longer batching time would lead to a much higher latency. For instance, with another two kernels *B* and *C*, the GPU timeline would be like Figure 2(2), where the CPU batching time and the GPU processing time become significantly longer.

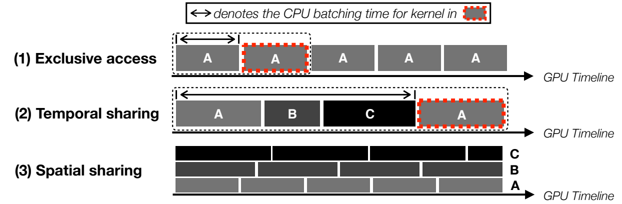


Figure 2: GPU execution with exclusive access, temporal sharing, and spatial sharing.

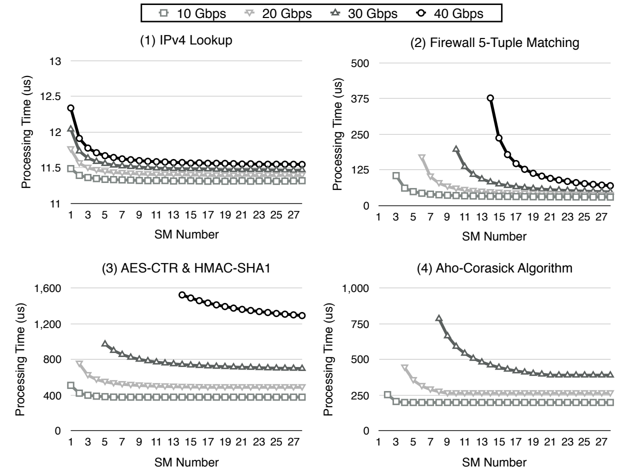


Figure 3: GPU processing time with different number of SMs (512-byte packet).

2.3 Opportunities and Challenges of Spatial GPU Sharing

With the lightweight kernels from multiple NFs, spatial GPU sharing is promising in enhancing the GPU efficiency. Spatial GPU sharing means multiple GPU

kernels run on a GPU simultaneously (shown in Figure 2(3)), with each kernel occupying a portion of GPU cores. This technique has been proved to gain a significant performance improvement on simulated GPU architectures [3, 26]. A recently-introduced GPU hardware feature, *Hyper-Q*, exploits spatial GPU sharing. Adopting *Hyper-Q* in NFV systems faces several challenges.

Challenge 1: GPU virtualization. To run concurrently on a GPU with *Hyper-Q*, the kernels are required to have the same GPU context. Kernels of NFs in different VMs, however, are unable to utilize the feature for their different GPU contexts. Existing GPU virtualization approaches [9, 39, 40] are designed for the situation that the GPU is monopolized by one kernel at any instant, which do not adopt *Hyper-Q*. Utilizing *Hyper-Q* in NFV systems, therefore, demands a redesign of the GPU virtualization approach.

Challenge 2: Isolation. As NFs might come from different vendors, data isolation is essential for ensuring data security. With the same GPU context when utilizing *Hyper-Q*, all GPU memory regions are in the same virtual address space. As runtimes such as CUDA and OpenCL do not provide isolation among kernels from the same context, an NF is able to access memory regions allocated by other NFs. Consequently, utilizing the *Hyper-Q* hardware feature may lead to security issues.

Challenge 3: GPU scheduling. Virtualization makes an NF unaware of other coexisting NFs that share the same GPU, making them lack of the global view in resource usage. With spatial GPU sharing, if every NF tries to maximize its performance by using more GPU resources, the performance of the service chain can be significantly degraded due to resource contention. Existing GPU scheduling schemes [10, 22, 41] focus on sporadic GPU tasks and temporal GPU sharing, which mismatch the requirements and characteristics of NFV systems.

3 G-NET - An Overview

We propose G-NET, an NFV system that addresses the above challenges and effectively shares GPUs among NFs. In this section, we make an overview on the major techniques adopted in G-NET.

3.1 The G-NET Approach

To efficiently and safely share a GPU in an NFV environment, G-NET adopts the following major approaches.

Utilizing Hyper-Q in GPU virtualization: To enable spatial GPU sharing, G-NET utilizes the *Hyper-Q* feature in GPUs. We place a GPU management proxy in the hypervisor, which creates a common GPU context for all NFs. By utilizing the context to perform data transfer and kernel operations, multiple kernels from different NFs can simultaneously run on the GPU.

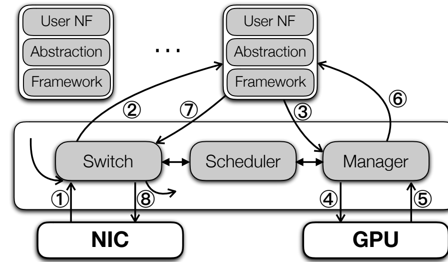


Figure 4: System architecture.

isoPointer for data isolation: We implement *isoPointer*, a software memory access checking layer for GPU kernels. In G-NET, the GPU memory is accessed with *isoPointer*, which behaves like regular pointers but is able to check if the accessed memory address is legal, i.e., whether it belongs to the current kernel. *isoPointer* ensures the data isolation of NFs in the GPU memory.

Service chain based GPU scheduling: G-NET develops a service chain based GPU scheduling scheme that aims at maximizing the throughput of a service chain. Based on the workload of each NF, *Scheduler* calculates the corresponding GPU resources for each NF kernel to optimize the performance of the entire service chain. The scheduling algorithm is capable of dynamically adapting to workload changes at runtime.

Abstraction: We propose an abstraction for developing NFs. By generalizing the CPU-GPU pipelining, data transfer, and multithreading in a framework, NF developers only need to implement a few NF-specific functions. With the abstraction, the implementation efforts of an NF are significantly reduced.

3.2 The Architecture of G-NET

The architecture of G-NET is shown in Figure 4. There are three major functional units in the hypervisor layer of G-NET: *Switch*, *Manager*, and *Scheduler*. *Switch* is a virtual switch that performs packet I/O and forwards network packets among NFs. *Manager* is the proxy for GPU virtualization, which receives GPU execution requests from NFs and performs the corresponding GPU operations. *Scheduler* allocates GPU resources to optimize the overall performance of a service chain.

G-NET adopts a holistic approach in which the NF and the hypervisor work together to achieve spatial GPU sharing. A framework is proposed to handle the data flows and control flows in NF executions. Based on the programming interfaces of the framework, developers only need to implement NF-specific operations, which significantly reduces the development efforts. The processing data flow of an NF is shown in Figure 4. An NF receives packets from *Switch*, which can be from a NIC

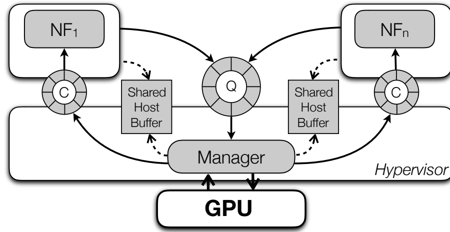


Figure 5: GPU virtualization in G-NET.

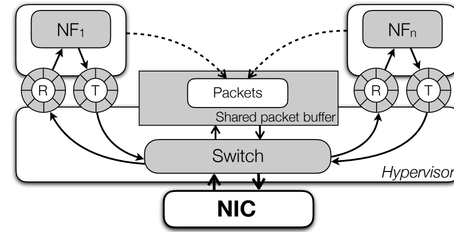


Figure 6: Data plane switching.

or other NFs. The NF batches jobs and utilizes *Manager* to perform GPU data transfer, kernel launch, and other operations. With the GPU processed results, the NF sends packets out through *Switch*.

4 System Design and Implementation

This section presents the main techniques adopted in G-NET, including the GPU virtualization approach, data plane switching, and GPU memory isolation.

4.1 GPU Virtualization and Sharing

As a common GPU context is demanded for spatial GPU sharing, GPU kernels launched in different VMs are unable to run simultaneously for their different contexts. To address this issue, G-NET creates a GPU context in the hypervisor and utilizes the context to execute GPU operations for all NFs. In order to achieve the goal, we virtualize at the GPU API level and adopt API remoting [14, 37] in GPU virtualization. API remoting is a virtualization scheme where device API calls in VMs are forwarded to the hypervisor to perform the corresponding operations. Figure 5 depicts our GPU virtualization approach. *Manager* maintains a response queue for each NF and a request queue that receives GPU requests from all NFs. To perform a GPU operation, an NF sends a message that includes the operation type and arguments to *Manager* via the request queue. *Manager* receives messages from NFs and performs the corresponding operations in the common GPU context asynchronously, so that it can serve other requests without waiting for their completion. Each NF is mapped to a CUDA stream by *Manager*, thus their operations run simultaneously in the same GPU context.

G-NET adjusts the number of thread blocks and the batch size of a kernel to achieve predictable performance (details in Sect. 5). In G-NET, the NF framework and the hypervisor work together to set these parameters. Upon receiving a kernel launch request, *Manager* uses the number of thread blocks that is calculated by *Scheduler* to launch the corresponding kernel. If the resource allocation scheme has been updated by *Scheduler*, *Manager* sends the batch size information to the NF via the response queue after the kernel completes execution. Then the framework in the NF uses the updated batch size to launch subsequent kernels.

There are several challenges for *Manager* to share the GPU. First, GPU kernels are located in VMs, which cannot be directly called by *Manager*. Second, the arguments should be set for launching a GPU kernel, but user-defined types cannot be known in the hypervisor. We utilize the CUDA driver API to address these issues. Each NF passes the source file and the name of its GPU kernel to *Manager* via a shared directory between the VM and the hypervisor. The hypervisor loads the kernel with CUDA driver APIs *cuModuleLoad* and *cuModuleGetFunction* and launches the kernel with *cuLaunchKernel*. Kernel arguments are stored in a shared host memory region by NFs, whose pointer can be directly passed to the kernel in *cuLaunchKernel*. In this way, *Manager* launches GPU kernels disregarding the specific argument details, which will be parsed automatically by the kernel itself.

For GPU operations such as kernel launch and PCIe data transfer, data is frequently transferred between a VM and the hypervisor. In G-NET, we develop a set of schemes to eliminate the overheads. When an NF requests to allocate a host memory region, *Manager* creates a shared memory region (shown in Figure 5) for each NF to transfer data by only passing pointers. For the allocation of the GPU memory, *Manager* directly passes the GPU memory pointer back to the NF, which would be passed back to perform PCIe data transfer or launch GPU kernels.

4.2 Data Plane Switching

Figure 6 shows the data plane switching in G-NET. Memory copy is known to have a huge overhead in high-speed network processing. To enhance the overall performance, we apply zero-copy principle in *Switch* to reduce the data transfer overhead of VM-to-NIC and VM-to-VM. Two communication channels are employed to move packets. One channel is a large shared memory region that stores packets. Packets are directly written into this memory region from NICs, allowing VMs to read and write packets directly. The other channel is two

queues to pass packet pointers. Each VM has an input queue to receive packets and an output queue to send packets. As only packet pointers are transferred between VMs, the data transfer overhead is significantly alleviated.

There are two types of threads in *Switch*, which are called as *RX* thread and *TX* thread. *RX* threads receive packets from NICs. After analyzing packet headers, *RX* threads distribute packets to corresponding NFs through their input queues. An NF sends out packets by enqueueing packet pointers in its output queue. A *TX* thread in *Switch* is in charge of forwarding packets of one or several output queues. Based on the service chain information, a *TX* thread sends the dequeued packets out through NICs or to the following NFs for further processing.

4.3 Isolation

To address the data security issue in GPUs, we develop *isoPointer*, a mechanism to guarantee the data isolation in the GPU device memory. *isoPointer* acts as a software memory access checking layer that guarantees the read and write operations in the GPU memory do not surpass the bound of the legal memory space of an NF. An *isoPointer* is implemented as a C++ class, which overloads the pointer operators, including dereference, pre-increment, and post-increment. Each *isoPointer* instance is associated with the *base* address and the *size* of a memory region. At runtime, *dynamic checking* is enforced to ensure that the accessed memory address of an *isoPointer* is within its memory region, i.e., [*base, base+size*].

Despite dynamic checking, we ensure that all memory accesses in an NF are based on *isoPointer* with *static checking*. Static checking is performed on the source code of each NF, which needs to guarantee two aspects. First, an *isoPointer* can only be returned from system interfaces or passed as arguments, and no *isoPointer* is initiated or modified by users. Second, the types of all pointers in the GPU source code are *isoPointer*. This is performed with type checking, a mature technique in compiler. With both static and dynamic checking, G-NET guarantees data isolation in the GPU memory.

5 Resource Allocation and Scheduling

In this section, we introduce our GPU resource allocation and scheduling schemes. The main goal of the scheduling scheme is to maximize the throughput of a service chain while meeting the latency requirement.

5.1 Resource Allocation

Unlike CPUs, GPUs behave as black boxes, which provide no hardware support for allocating cores to applications. Threads are frequently switched on and off GPU cores when blocked by operations such as memory access or synchronization. As a result, it is unable to precisely allocate a GPU core to a GPU thread. Moreover, whether kernels can take advantage of *Hyper-Q* to spatially share the GPU depends on the availability of the GPU resources, including the registers and the shared memory. When the resources to be taken by a kernel exceed current available resources, the kernel would be queued up to wait for the resources becoming available. Consequently, an improper GPU resource allocation scheme is detrimental to the overall performance.

G-NET uses SM as the basic unit in the allocation of GPU computational resources. A thread block is a set of threads that can only run on one SM. Modern GPUs balance the number of thread blocks on SMs, where two thread blocks are not supposed to be scheduled to run on the same SM when there still exists available ones. We utilize this feature and allow the same or a smaller number of thread blocks as that of SMs to run on a GPU simultaneously, so that each thread block executes exclusively on one SM. By specifying the number of thread blocks of a GPU kernel, an NF is allocated with an exact number of SMs, and multiple GPU kernels can co-run simultaneously.

We propose a GPU resource allocation scheme that uses two parameters to achieve predictable performance: the batch size and the number of SMs. The scheme is based on a critical observation: there is only marginal performance impact ($< 7\%$ in our experiments) from thread blocks running on different SMs. When utilizing more SMs with each SM being assigned with the same load, the overall kernel execution time (w/o PCIe data transfer) stays relatively stable. The main reasons for this phenomenon are twofold. First, the memory bandwidth of current GPUs is high (480 GB/s on NVIDIA Titan X Pascal), which is sufficient for co-running several NFs on 10 Gbps network. Second, SMs do not need to compete for other resources such as register file or cache, as each SM has its independent resources. Therefore, the batch size of an SM controls its throughput and processing time, while allocating more SMs can reap a near-linear throughput improvement.

5.2 Performance Modeling

To achieve predictable performance by controlling the batch size of an SM, we model the relationship between the performance of an SM and the batch size with our evaluation results. Figure 7 shows the GPU kernel execution time on one SM, the PCIe data transfer time, and the corresponding throughputs of four NFs.

As shown in the figure, the throughput of GPU kernels have different patterns, where the throughputs of

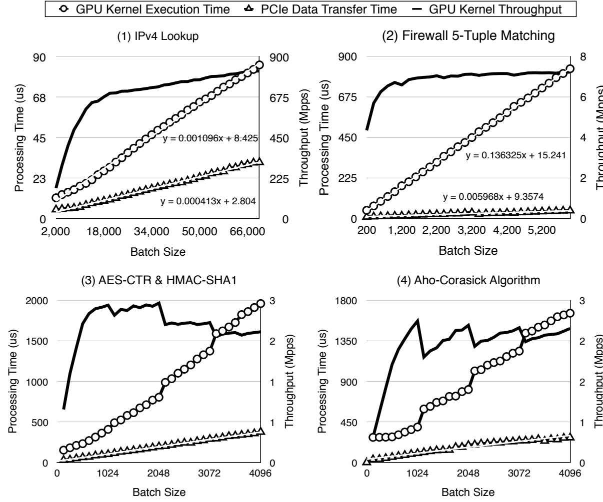


Figure 7: GPU processing time and throughput on one SM (Packet size: 512-byte; Thread block size: 1024).

NIDS and IPsec gateway increase and then drop in steps with the increase of batch size. As the kernels reach the highest throughputs before the first performance degradation, we use the batch size in the first step to adjust the SM throughput. The kernel execution time and the PCIe data transfer time form a near-linear relationship with the batch size (within the first step). Therefore, we adopt linear regression in the model. We use $L_k = k_1 \cdot B + b_1$ to describe the relationship between the batch size (B) and the GPU kernel execution time (L_k), and $L_m = k_2 \cdot B + b_2$ is used to describe the relationship between the batch size (B) and the PCIe data transfer time (L_m). As the kernel execution time stays stable with different number of SMs when the batch size assigned to each SM stays the same, we can get the overall GPU execution time of a kernel as

$$L = L_k + L_m = k_1 \cdot B_0 + b_1 + k_2 \cdot B_0 \cdot N + b_2 \qquad (1)$$

, where N denotes the number of SMs, and B_0 denotes the batch size of each SM. The throughput can be derived as

$$T = N \cdot B_0 / L \qquad (2)$$

The parameters of the linear equations are relative with hardware platform, number of rules installed (e.g., NIDS and Firewall), and packet size. In system deployment, we develop a tool called *nBench* to measure the PCIe data transfer time and the kernel execution time to derive the corresponding linear equations with locally generated packets. With *nBench*, our system can be quickly deployed on servers with different hardware and software configurations by only profiling each NF once.

We have implemented several GPU-based NFs, and we find that NFs can be classified into two main groups. 1) For NFs that inspect packet payloads, the performance has a similar pattern with NIDS and IPsec gateway. The length of the performance fluctuation step equals to the thread block size of the GPU kernel (1024 in our evaluation). 2) For NFs that only inspect packet headers, the performance exhibits the same pattern with Router and Firewall. Therefore, this simple but effective performance model can be applied to other NFs. Moreover, in the G-NET implementation, our scheduling scheme considers the potential model deviations (denote as C in Sect. 5.3) in the resource allocation, making our approach more adaptive in practice.

5.3 Service Chain Based GPU Scheduling

The GPU scheduling in NFV systems has the following specific aspects. (1) The packet arrival rate and the packet processing cost of each NF are dramatically different. If each NF is allocated with the same amount of resources, the NF with the heaviest load would degrade the overall throughput [23]. (2) The workload of an NFV system can be dynamically changing over time. For instance, under malicious attack, the throughput of NIDS should be immediately enhanced.

Based on the performance model, we propose a scheduling scheme that aims at maximizing the throughput of an entire service chain while meeting the latency requirements of NFs. Different with the modeling environment, the scheduling scheme needs to consider the costs brought by the implementation and hardware. First, there is overhead in the communication between NFs and *Manager* in performing GPU operations. Second, as there are only one host-to-device (HtoD) and one device-to-host (DtoH) DMA engine in current GPUs, the data transfer of an NF has to be postponed if the required DMA engine is occupied by other NFs. Third, the model may have deviations. Our scheduling scheme takes these overheads into consideration (denote by C), which works as follows.

At runtime, *Scheduler* monitors the throughput of each NF and progressively allocates GPU resources. We first find the NF that achieves the lowest throughput (denote by T') in the service chain, then allocate all NFs with enough GPU resources to meet the throughput $T = T' \cdot (1+P)$, where $P \in (0,1)$. If there are branches in the service chain, we first allocate resources for NFs in each branch. Then the sum of the throughputs of the child branches is used as the throughput for their father branch in resource allocation. This procedure repeats until GPU resources are exhausted, which improves the overall throughput by P in each round.

In each round, *Scheduler* calculates the minimum number of SMs and the batch size to meet the latency and throughput requirements of each NF. Starting from assigning one SM, the scheme checks if the current number

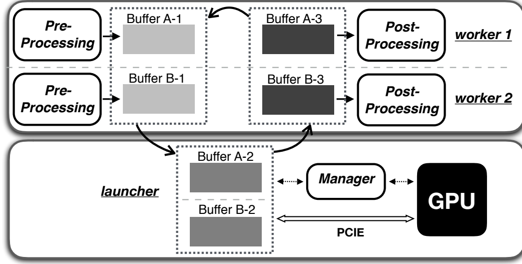


Figure 8: The framework of a network function.

of SMs (denoted as N) is able to meet the schedulability condition of an NF, i.e., achieving a latency lower than the user-specified latency requirement L and a throughput higher than the input speed T. According to Eq. 2, by substituting $N \cdot B_0/T$ for L into Eq. 1, the minimum batch size demanded to achieve the throughput T is derived as

$$B_0 = \lceil T \cdot (b_1 + b_2 + C)/(N - T \cdot (k_1 + k_2 \cdot N)) \rceil \quad (3)$$

Then, the overall processing time can be calculated as

$$L_0 = k_1 \cdot B_0 + b_1 + k_2 \cdot B_0 \cdot N + b_2 + C \quad (4)$$

If $L_0 \leq L$, it means the NF can meet both its latency and throughput requirements with N SMs and a batch size of $B_0 \cdot N$. If $L_0 > L$, one more SM is allocated and the above procedure is performed again. The procedure repeats until no available SMs left in the GPU.

In G-NET, the scheduling scheme runs when the traffic of an NF changes for more than 10%.

6 Abstraction and Framework for NFs

The implementation of NFs on CPU-GPU architectures should be efficient and easy to scale up. In G-NET, we propose an abstraction for NFs to reduce the development efforts.

6.1 Framework

We generalize the CPU-GPU pipeline of network functions as three main stages, namely *Pre-processing*, *GPU processing*, and *Post-processing*. Figure 8 shows our framework. The CPU is in charge of batching and packet I/O in *Pre-processing* and *Post-processing* stages, and the GPU is in charge of compute/memory-intensive operations in the *GPU processing* stage. Our main design choices are as follows.

Thread model in CPU-GPU pipelining: There are two types of CPU threads in the pipeline. *Worker* threads perform packet processing in *Pre-processing* and *Post-processing* stages. To communicate with *Manager* for GPU execution, a specific thread called *launcher* is implemented to manage GPU operations including data transfer and kernel execution. This avoids *worker* threads waiting for the GPU operations and makes them focus on processing continuously coming packets.

Buffer transfer among pipeline stages: We develop a buffer transfer mechanism to prevent *workers* from being stalled when transferring data to the GPU. There are three buffers in each pipeline. The *launcher* thread automatically switches the buffer of the *Pre-processing* stage when the *Scheduler*-specified batch size is reached. The following two stages pass their buffers to the next stages circularly after they complete their tasks.

Scale up: We scale up a network function when the CPU becomes its bottleneck, i.e., launching more *worker* threads to enhance the data processing capability. When executing a GPU operation, the *launcher* passes the thread id to *Manager*, which is mapped with a CUDA stream for independent execution. With an independent input and output queue for each *worker*, the design simplifies NF management and enhances throughput.

6.2 Abstraction

Based on our framework, we propose an abstraction to mitigate the NF development efforts. The abstraction mainly consists of five basic operations, i.e., *pre_pkt_handler*, *mem_htod*, *set_args*, *mem_dtoh*, *post_pkt_handler*. Called by the framework in the *Pre-processing* stage, *pre_pkt_handler* performs operations including packet parsing and batching. The framework manages the number of jobs in the batch, and developers only need to batch a job in the position of *batch->job_num* in the buffer. Before the framework launches the GPU kernel for a batch, *mem_htod* and *set_args* are called to transfer data from the host memory to the GPU memory and set arguments for the GPU kernel. Please note that the order of the arguments should be consistent with the GPU kernel function in *set_args*. Then the framework sends requests to *Manager* to launch the specified GPU kernel. After kernel completes execution, *mem_dtoh* is called to transfer data from the GPU memory to the host memory. *post_pkt_handler* is called for every packet after GPU processing.

As an example, Figure 9 demonstrates the major parts of a router implemented with our abstraction. First, the developer defines the specific batch structure (lines 1-7). In a router, it includes the number of jobs in a batch and the input and output buffers in the host and the GPU memory. Each *worker* thread is allocated with a batch structure for independent processing. With the *kernel_init* function (lines 8-11), developers install its kernel by specifying its *.cu* kernel file and the kernel function name ("*iplookup*"). Developers can also perform other initialization operations in *kernel_init*, such as building

```
1    struct my_batch {
2        uint64_t job_num;
3        isoPtr<uint32_t> host_ip;
4        isoPtr<uint32_t> dev_ip;
5        isoPtr<uint8_t> host_port;
6        isoPtr<uint8_t> dev_port;
7    }
8    void kernel_init(void) {
9        gInstallKernel("/pathto/router.cu", "iplookup");
10       build_routing_table();
11   }
12   void pre_pkt_handler(batch, pkt) {
13       batch->host_ip[batch->job_num] = dest_ip(pkt);
14   }
15   void memcpy_htod(batch) {
16       gMemcpyHtoD(batch->dev_ip, batch->host_ip,
17           batch->job_num * IP_SIZE);
18   }
19   void set_args(batch) {
20       gInstallArgNum(4);
21       gInstallArg(batch->dev_ip);
22       gInstallArg(batch->dev_port);
23       gInstallArg(batch->job_num);
24       gInstallArg(dev_lookup_table);
25   }
26   void memcpy_dtoh(batch) {
27       gMemcpyDtoH(batch->host_port, batch->dev_port,
28           batch->job_num * PORT_SIZE);
29   }
30   void post_pkt_handler(batch, pkt, pkt_idx) {
31       pkt->port = batch->host_port[pkt_idx];
32   }
```

Figure 9: The major parts of a router implemented with G-NET APIs.

the routing table. The *pre_pkt_handler* (lines 12-14) of the router extracts the destination IP address of a packet and batches it in the host buffer. Functions *mem_htod* and *mem_dtoh* transfer the ip address buffer into the GPU memory and the port buffer into the host memory, respectively. *post_pkt_handler* (lines 30-32) records the port number which will be used by the framework to send the packet out. With the abstraction, developers only need to focus on the implementation of specific tasks of an NF, reducing thousands of lines of development efforts.

Based on our abstraction, NFs can be developed by different vendors, where the GPU kernel source code should be provided so that the kernels can be loaded by the *Manager*. If vendors that do not want to leak their source code, they may have to deploy a whole package of the G-NET system, including the NFs and the functionalities in the hypervisor. A secure channel with cryptographic operations can be built between NFs and the *Manager* to pass the source code.

7 Experiment

7.1 Experimental Methodology

Experiment Platform: We conduct experiments on a PC equipped with an Intel Xeon E5-2650 v4 processor running at 2.2 GHz. The processor contains 12 physical cores with hyper-threading enabled. The processor has an integrated memory controller installed with 4×16 GB 2400 MHz DDR4 memory. An NVIDIA Titan X Pascal GPU is deployed, which has 28 streaming multiprocessors and a total of 3584 cores. An Intel XL710 dual port 40 GbE NIC is used for network I/O, and DPDK 17.02.1 is used as the driver. The operating system is 64-bit CentOS 7.3.1611 with Linux kernel 3.8.0-30. Docker 17.03.0-ce is used as our virtualization platform. Each NF runs in a Docker instance, while *Manager*, *Switch*, and *Scheduler* run on the host.

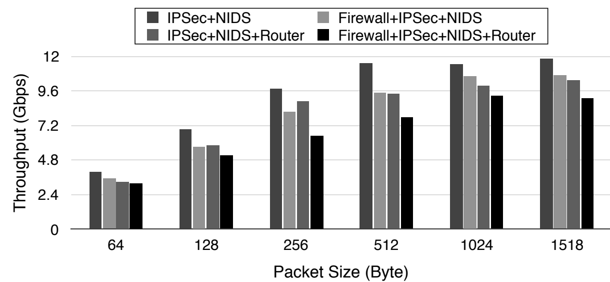


Figure 10: System throughput of G-NET.

Service Chains: We implement four network functions on G-NET, i.e., Router, Firewall, NIDS, IPsec gateway, as listed in Table 1. Composed by the NFs, four service chains are used to evaluate the performance of G-NET: ($\mathbb{S}_a$) IPsec + NIDS; ($\mathbb{S}_b$) Firewall + IPsec + NIDS; ($\mathbb{S}_c$) IPsec + NIDS + Router; ($\mathbb{S}_d$) Firewall + IPsec + NIDS + Router.

7.2 System Throughput

Figure 10 shows the throughput of G-NET for the four service chains. We set one millisecond as the latency requirement for the GPU execution of each NF, as we aim at evaluating the maximum throughput of G-NET. With the service chain $\mathbb{S}_a$ that has two NFs, the system throughput reaches up to 11.8 Gbps with the maximum ethernet frame size 1518-byte. For the service chain $\mathbb{S}_d$ with four NFs, the system achieves a throughput of 9.1 Gbps.

As depicted in the figure, the system throughput increases with the size of packet. When the packet size is small, the input data volume of service chains is limited by the nontrivial per-packet processing overhead, including switching, batching, and packet header parsing. The main overhead comes from packet switching. Switching a packet between two NFs includes at least two enqueue and dequeue operations, and the packet header should be inspected to determine its destination, which is known to have severe performance issues [17, 27, 32, 34]. In an NFV system with a service chain of multiple NFs, the problem gets more pronounced as a packet needs to be forwarded multiple times in the service chain. More-

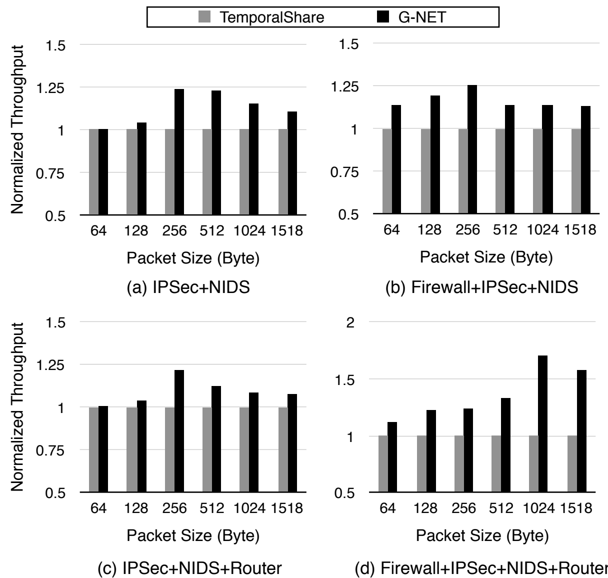


Figure 11: Throughput comparison between G-NET and *TemporalShare*. Performance is normalized to *TemporalShare*.

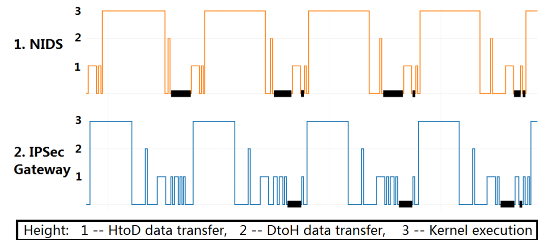


Figure 12: Data transfer conflicts in G-NET.

over, as the NFs in the system demand CPU cores for packet processing, leaving *Switch* limited resources for packet forwarding. Assigning the *Switch* with more threads, however, would deprive the cores allocated for the *worker* threads in NFs, resulting in overall performance degradation.

7.3 Performance Improvement From Spatial GPU Sharing

To evaluate the performance improvement with spatial GPU sharing, we implement a *TemporalShare* mode in G-NET for comparison. The *TemporalShare* mode represents existing GPU virtualization approaches, where kernels from NFs access the GPU serially instead of being executed simultaneously.

Figure 11 compares the throughput of G-NET and *TemporalShare*. As shown in the figure, the spatial GPU sharing in G-NET significantly enhances the overall throughput. For the four service chains, the throughput improvements reach up to 23.8%, 25.9%, 21.5%, and 70.8%, respectively. The throughput improvements for the service chain with four NFs are higher than that of service chains with less NFs. For a small number of NFs, a fraction of the GPU kernels and PCIe data transfer could overlap with *TemporalShare*, leading to less resource contention. When there are more NFs co-running in the system, they can reap more benefits of spatial GPU sharing for the fierce resource competition.

There are two main aspects that limit the performance improvement by spatial GPU sharing. First, although GPU kernels are able to utilize Hyper-Q to co-run on a GPU, their PCIe data transfer have to be sequentially performed. This is due to the limited number of DMA engines. Figure 12 plots the trace of G-NET with IPsec gateway and NIDS, which demonstrates the concurrent kernel executions with spatial GPU sharing and the DMA data transfer conflicts. The delays in NFs caused by the data transfer conflicts are marked as bold black lines in the figure. As shown in the figure, NFs spend a significant amount of time in waiting for the HtoD DMA engine when it is occupied by other NFs. The system performance could be further unleashed when hardware vendors equip GPUs with more DMA engines for parallel data transfer. Second, the bottleneck of G-NET on current evaluation platform lies in the CPU. Our GPU provides abundant computational resources, which is unable to be matched by the CPU. For instance, with four NFs, each NF can only be assigned with two physical cores, resulting in low packet processing capability. For workloads with large packets, the batching operations that perform *memcpy* on packet payloads limit the overall performance. Instead, the switching overhead becomes the main factor that affects the overall performance for workloads with small packets. It is our future work to investigate how to further reduce this overhead.

7.4 Evaluation of Scheduling Schemes

To evaluate the effectiveness of the GPU scheduling in G-NET, we use two other scheduling schemes for comparison, i.e., *FairShare* and *Uncohare*. Different with the scheduling scheme of G-NET, the SMs are evenly partitioned among all NFs in the *FairShare* mode. In the *UncoShare* mode, *Scheduler* is disabled, and each NF tries to use as many GPU resources as possible.

The throughput improvements of G-NET over *FairShare* and *UncoShare* are shown in Figure 13 and Figure 14, respectively. For the four service chains, the average throughput improvements of G-NET are 16.7-34.0% over *FairShare* and 50.8-130.1% over *UncoShare*. The throughput improvements over *UncoShare* are higher than that of *FairShare* for the following reasons. In the *FairShare* mode, although the GPU resources are parti-

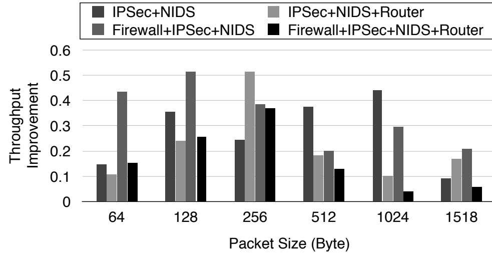


Figure 13: Throughput improvements in G-NET over *FairShare* GPU scheduling.

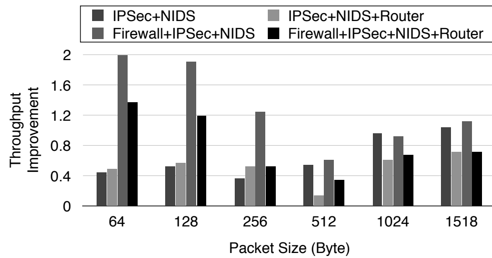


Figure 14: Throughput improvements in G-NET over *UncoShare* GPU scheduling.

tioned among the NFs, they are still able to co-run their GPU kernels simultaneously. In the *UncoShare* mode, however, each NF tries to use as many resources (SMs) as possible. This may exhaust the GPU resources, making part of the GPU kernels cannot run together. Moreover, the performance of a kernel can be degraded due to the interference of kernels running on the same SMs. To conclude, our scheduling scheme demonstrates very high efficiency in enhancing the performance of the service chains.

7.5 The Overhead of IsoPointer

To guarantee the isolation of different NFs, G-NET uses *IsoPointer* to check, validate, and restrict GPU memory accesses. These management activities may add some runtime overhead to NF executions.

We measure the overhead by comparing the performance of G-NET with and without *IsoPointer* under two

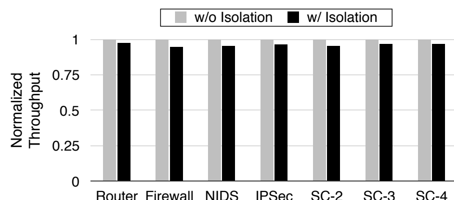


Figure 15: Overhead of IsoPointer.

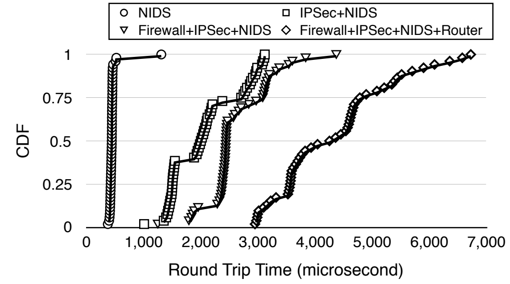


Figure 16: System latency of four service chains.

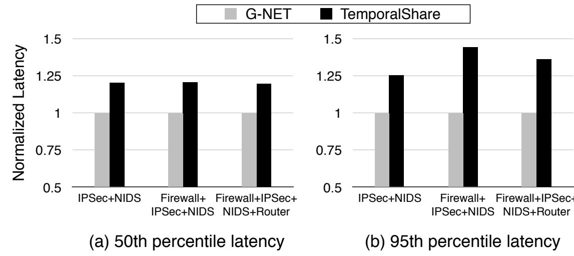


Figure 17: Latency comparison with *TemporalShare*.

groups of workloads. The first group consists of the solo executions of the NFs, and the second group comprises service chains with 2-4 NFs. The results are presented in Figure 15, where *SC-k* denotes a service chain with k NFs. We report the performance of service chain $\mathbb{S}_b$ for *SC-3*, as the measured overheads of $\mathbb{S}_b$ and $\mathbb{S}_c$ are similar. As shown in the figure, the overhead of *IsoPointer* ranges from 2.8% to 4.5%, which is negligibly low.

7.6 Latency

We evaluate system processing latency by measuring the time elapsed from sending a packet to the NFV system to receiving the processed packet. The client keeps sending packets with source IP address increasing by one for each packet. By sample logging the sending/receiving time and the IP address, the round trip latency can be calculated as the time elapsed between the queries and responses with matched IP addresses.

Figure 16 shows the Cumulative Distribution Function (CDF) of the packet round trip latency with four service chains. The latency is measured by setting the maximum GPU execution time of each NF as one millisecond, as it demonstrates the system latency with the maximum throughput. As shown in the figure, the latency of one NF is low and stable. With two to four NFs, the service chains show piecewise CDFs, where latencies are clustered into three or more areas. The main reason for this phenomenon is the PCIe data transfer conflict. As there are only one HtoD and one DtoH DMA engine in current GPUs, a kernel would be postponed for execution

if other NFs are utilizing the engine, leading to higher processing latency. If data is transferred while other NFs are running GPU kernels, they will overlap with no performance loss. Therefore, different degrees of resource competition in NF executions lead to the several clusters of latency distribution in the CDF. The latency in G-NET mainly comes from three parts, i.e., GPU processing, batching, and packet switching. With a lower input network speed, the latency would decrease for the lower batching time and GPU processing time.

Figure 17 compares the latency of G-NET with *TemporalShare*. With temporal GPU sharing, the 50th percentile latency is around 20% higher than that of G-NET, and the 95th percentile latency is 25.5%-44.3% higher than that of G-NET. It is worth noting that the throughput of *TemporalShare* is lower than G-NET (shown in Figure 11). Without spatial GPU sharing, both the kernel execution and the PCIe data transfer are serialized, resulting in low throughputs and high latencies.

8 Related Work

GPU Accelerated Network Functions: There are a class of work that utilize GPUs in accelerating network functions, including router [15], SSL reverse proxy [20], NIDS [19], and NDN system [42]. APUnet [12] studies NF performance on several architectures, which demonstrates the capability of GPUs in efficient packet processing and identifies the overhead in PCIe data transfer.

Network Function Virtualization: Recent NFV systems that are built with high performance data plane include NetVM [18] and ClickNP [25]. NetVM is a CPU-based NFV framework that implements zero-copy data transfer among VMs, which inspires the design of *Switch* in G-NET. NFVnice [23] is an NF scheduling framework that maximizes the performance of a service chain by allocating CPU time to NFs based on their packet arrival rate and the required computational cost. The goal of the CPU resource allocation in NFVnice is the same with that of the GPU resource allocation in G-NET. ClickNP and Emu [28] accelerate the data plane of network functions with FPGA. Same with G-NET, they also provide high-level languages to mitigate the development efforts. NFP [38] is a CPU-based NFV framework that exploits the opportunity of NF parallel execution to improve NFV performance. On the control plane, there is a class of work [6, 11, 29] focus on flexible and easy NF deployment including automatic NF placement, dynamic scaling, and NF migration.

GPU Virtualization: GPU virtualization approaches are generally classified into three major classes, namely, I/O pass-through [2], API remoting [14, 37], and hybrid [9]. Recent work includes alleviating the overhead of GPU virtualization [39, 40] and resolving the memory capacity limitation [45]. These systems do not explore the spatial GPU sharing, as most of them assume a kernel would saturate GPU resources.

GPU Sharing: Recent work on GPU multitasking studies spatial GPU sharing [3] and simultaneous multikernel [43, 44] with simulation. [31] shows that spatial GPU sharing has better performance when there is resource contention among kernels. Instead, simultaneous multikernel, which allows kernels to run on the same SM, has advantage for kernels with complimentary resource demands [31]. NF kernels have similar operations in packet processing, making spatial sharing a better choice.

Isolation: For the implementation of *IsoPointer*, G-NET adopts a similar technique with ActivePointer [35], which intercepts GPU memory accesses for address translation. Paradice [5] proposes a data isolation scheme in paravirtualization, which guarantees that the CPU code of a VM can only access its own host and device memory. Different with G-NET, Paradice is unable to prohibit malicious GPU code from accessing illegal GPU memory regions. Instead of virtualization, NetBricks [30] utilizes type checking and safe runtimes to provide data isolation for CPU-based NFs. This approach discards the flexibility brought by virtualization, such as NF migration and the ability to run on different software/hardware platforms. Moreover, we find the performance penalties caused by virtualization is negligibly low in G-NET (only around 4%).

9 Conclusion

We propose G-NET, an NFV system that exploits spatial GPU sharing. With a service chain based GPU scheduling scheme to optimize the overall throughput, a data isolation scheme to guarantee data security in the GPU memory, and an abstraction to significantly reduce development efforts, G-NET enables effective and efficient adoption of GPUs in NFV systems. Through extensive experiments, G-NET significantly enhances the throughput by up to 70.8% and reduces latency by up to 44.3% for GPU-based NFV systems.

10 Acknowledgement

We would like to thank our shepherd KyoungSoo Park and anonymous reviewers of NSDI'18 for their insightful comments and suggestions. We also acknowledge the support of NVIDIA Corporation with the donation of the Titan X Pascal GPU used for this research. This work was supported in part by a MoE AcRF Tier 2 grant (MOE2017-T2-1-122) in Singapore, NSFC (Grant No. 61732004, 61370080) and the Shanghai Innovation Action Project (Grant No. 16DZ1100200) in China.

References

[1] Alihpc. "https://hpc.aliyun.com/product/gpu_bare_metal/".

[2] Amazon high performance computing cloud using GPU. "http://aws.amazon.com/hpc/".

[3] Adriaens, J. T., Compton, K., Kim, N. S., and Schulte, M. J. The Case for GPGPU Spatial Multitasking. In *HPCA* (2012), pp. 1–12.

[4] Aho, A. V., and Corasick, M. J. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM 18*, 6 (1975), 333–340.

[5] Amiri Sani, A., Boos, K., Qin, S., and Zhong, L. I/O Paravirtualization at the Device File Boundary. In *ASPLOS* (2014), pp. 319–332.

[6] Bremler-Barr, A., Harchol, Y., and Hay, D. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *SIGCOMM* (2016), pp. 511–524.

[7] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*, 3rd ed. 2009.

[8] De La Briandais, R. File Searching Using Variable Length Keys. In *Western Joint Computer Conference* (1959), pp. 295–298.

[9] Dowty, M., and Sugerman, J. GPU Virtualization on VMware's Hosted I/O Architecture. In *SIGOPS Oper. Syst. Rev.* (2009), pp. 73–82.

[10] Elliott, G. A., and Anderson, J. H. Globally Scheduled Real-time Multiprocessor Systems with GPUs. *Real-Time Syst.* (2012), 34–74.

[11] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM* (2014), pp. 163–174.

[12] Go, Y., Jamshed, M. A., Moon, Y., Hwang, C., and Park, K. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *NSDI* (2017), pp. 83–96.

[13] Gupta, P., Lin, S., and McKeown, N. Routing lookups in hardware at memory access speeds. In *INFOCOM* (1998), pp. 1240–1247.

[14] Gupta, V., Schwan, K., Tolia, N., Talwar, V., and Ranganathan, P. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *USENIX ATC* (2011).

[15] Han, S., Jang, K., Park, K., and Moon, S. PacketShader: A GPU-Accelerated Software Router. In *SIGCOMM* (2010).

[16] Hawilo, H., Shami, A., Mirahmadi, M., and Asal, R. NFV: state of the art, challenges, and implementation in next generation mobile networks (vEPC). *IEEE Network 28*, 6 (2014), 18–26.

[17] Honda, M., Huici, F., Lettieri, G., and Rizzo, L. mSwitch: A Highly-scalable, Modular Software Switch. In *SOSR* (2015), pp. 1:1–1:13.

[18] Hwang, J., Ramakrishnan, K. K., and Wood, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *NSDI* (2014), pp. 445–458.

[19] 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 *CCS* (2012), pp. 317–328.

[20] Jang, K., Han, S., Han, S., Moon, S., and Park, K. SSLshader: Cheap SSL Acceleration with Commodity Processors. In *NSDI* (2011).

[21] Kalia, A., Zhou, D., Kaminsky, M., and Andersen, D. G. Raising the Bar for Using GPUs in Software Packet Processing. In *NSDI* (2015), pp. 409–423.

[22] Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. TimeGraph: GPU Scheduling for Real-time Multi-tasking Environments. In *USENIX ATC* (2011).

[23] Kulkarni, S. G., Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K. K., Wood, T., Arumaithurai, M., and Fu, X. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *SIGCOMM* (2017), pp. 71–84.

[24] Lakshman, T. V., and Stiliadis, D. High-speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *SIGCOMM* (1998), pp. 203–214.

[25] Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *SIGCOMM* (2016), pp. 1–14.

[26] Liang, Y., Huynh, H. P., Rupnow, K., Goh, R. S. M., and Chen, D. Efficient GPU Spatial-Temporal Multitasking. *IEEE Transactions on Parallel and Distributed Systems 26* (2015), 748–760.

[27] Molnár, L., Pongrácz, G., Enyedi, G., Kis, Z. L., Csikor, L., Juhász, F., Kőrösi, A., and Rétvári, G. Dataplane Specialization for High-performance OpenFlow Software Switching. In *SIGCOMM* (2016), pp. 43–56.

[28] N, S., S, G., D, G., M, W., J, S., R, C., L, M., P, B., R, S., R, M., P, C., P, P., J, C., AW, M., and N, Z. Emu: Rapid Prototyping of Networking Services. In *USENIX ATC* (2017), pp. 459–471.

[29] Palkar, S., Lan, C., Han, S., Jang, K., Panda, A., Ratnasamy, S., Rizzo, L., and Shenker, S. E2: A Framework for NFV Applications. In *SOSP* (2015), pp. 121–136.

[30] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., and Shenker, S. NetBricks: Taking the V out of NFV. In *OSDI* (2016), pp. 203–216.

[31] Park, J. J. K., Park, Y., and Mahlke, S. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *ASPLOS* (2017), pp. 527–540.

[32] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., and Casado, M. The Design and Implementation of Open vSwitch. In *NSDI* (2015), pp. 117–130.

[33] Pietro, R. D., Lombardi, F., and Villani, A. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. *ACM Trans. Embed. Comput. Syst. 15* (2016), 15:1–15:25.

[34] Rizzo, L., and Lettieri, G. VALE, a Switched Ethernet for Virtual Machines. In *CoNEXT* (2012), pp. 61–72.

[35] Shahar, S., Bergman, S., and Silberstein, M. ActivePointers: A Case for Software Address Translation on GPUs. In *ISCA* (2016), pp. 596–608.

[36] Sherry, J., and Ratnasamy, S. A Survey of Enterprise Middlebox Deployments. Tech. rep., 2012.

[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] Sun, C., Bi, J., Zheng, Z., Yu, H., and Hu, H. NFP: Enabling Network Function Parallelism in NFV. In *SIGCOMM* (2017), pp. 539–552.

[39] Suzuki, Y., Kato, S., Yamada, H., and Kono, K. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *USENIX ATC* (2014), pp. 109–120.

[40] TIAN, K., DONG, Y., AND COWPERTHWAITE, D. A Full GPU Virtualization Solution with Mediated Pass-Through. In *USENIX ATC* (2014), pp. 121–132.

[41] WANG, K., ZHANG, K., YUAN, Y., MA, S., LEE, R., DING, X., AND ZHANG, X. Concurrent Analytical Query Processing with GPUs. *VLDB* (2014), 1011–1022.

[42] WANG, Y., ZU, Y., ZHANG, T., PENG, K., DONG, Q., LIU, B., MENG, W., DAI, H., TIAN, X., XU, Z., WU, H., AND YANG, D. Wire Speed Name Lookup: A GPU-based Approach. In *NSDI* (2013), pp. 199–212.

[43] WANG, Z., YANG, J., MELHEM, R., CHILDERS, B., ZHANG, Y., AND GUO, M. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *HPCA* (2016), pp. 358–369.

[44] XU, Q., JEON, H., KIM, K., RO, W. W., AND ANNAVARAM, M. Warped-slicer: Efficient intra-SM Slicing Through Dynamic Resource Partitioning for GPU Multiprogramming. In *ISCA* (2016), pp. 230–242.

[45] XUE, M., TIAN, K., DONG, Y., MA, J., WANG, J., QI, Z., HE, B., AND GUAN, H. gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space. In *USENIX ATC* (2016), pp. 579–590.

[46] ZHANG, H., ZHENG, Z., XU, S., DAI, W., HO, Q., LIANG, X., HU, Z., WEI, J., XIE, P., AND XING, E. P. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *USENIX ATC* (2017), pp. 181–193.

[47] ZHANG, K., HU, J., AND HUA, B. A Holistic Approach to Build Real-time Stream Processing System with GPU. *JPDC 83*, C (2015), 44–57.

[48] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-KV: A Case for GPUs to Maximize the Throughput of In-memory Key-value Stores. *VLDB* (2015), 1226–1237.

SafeBricks: Shielding Network Functions in the Cloud

Rishabh Poddar
UC Berkeley

Chang Lan
UC Berkeley

Raluca Ada Popa
UC Berkeley

Sylvia Ratnasamy
UC Berkeley

Abstract

With the advent of network function virtualization (NFV), outsourcing network processing to the cloud is growing in popularity amongst enterprises and organizations. Such outsourcing, however, poses a threat to the security of the client's traffic because the cloud is notoriously susceptible to attacks.

We present SafeBricks, a system that *shields* generic network functions (NFs) from an untrusted cloud. SafeBricks ensures that only encrypted traffic is exposed to the cloud provider, and preserves the integrity of both traffic and the NFs. At the same time, it enables clients to reduce their trust in NF implementations by enforcing least privilege across NFs deployed in a chain. SafeBricks does not require changes to TLS, and safeguards the interests of NF vendors as well by shielding NF code and rulesets from both clients and the cloud. To achieve its aims, SafeBricks leverages a combination of hardware enclaves and language-based enforcement. SafeBricks is practical, and its overheads range between ∼0–15% across applications.

1 Introduction

Modern networks consist of a wide range of appliances that implement advanced network functions beyond merely forwarding packets, such as scanning for security issues (*e.g.*, firewalls, IDSes) or improving performance (*e.g.*, WAN optimizers, web caches). Traditionally, these network functions (or NFs) have been deployed as dedicated hardware devices. In recent years, however, both industry and academia have proposed the replacement of the devices with software implementations running in virtual machines [55, 62], a model called Network Function Virtualization (NFV). Inevitably, the advent of NFV has spurred the growth of a new industry wherein third-parties offer traffic processing capabilities as a cloud service to customers [4, 51, 62, 79]. Such a service model enables enterprises to outsource NFs from their networks entirely to the third-party service, bringing the benefits of cloud computing and reducing costs.

However, outsourcing NFs to the cloud poses new challenges to enterprise networks—security.

Need to protect traffic from the cloud. By allowing the cloud provider to process enterprise traffic, enterprises end up granting to the cloud the ability to see their sensitive traffic and tamper with NF processing. While the cloud itself might be a benign entity, it is vulnerable to hackers [56], subpoenas [24, 45, 75], and insider attacks [10, 54, 77]. This is doubly worrisome because not only does network traffic contain sensitive information, but some NFs are also designed to protect enterprises against intrusions which an attacker could try to disrupt.

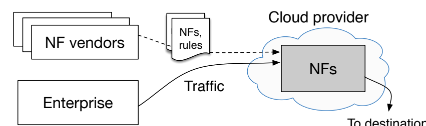


Figure 1: Model for outsourced NFs.

Need to protect traffic from NF. What complicates matters further is that often, an enterprise must also trust another party with its traffic: NF vendors. This is the case when enterprises procure proprietary NF implementations and rulesets from NF vendors [8, 22, 51] instead of using their own, as shown in Figure 1. While such NFs typically need access only to specific portions of the traffic (*e.g.*, IP firewalls only need read access to packet headers), the enterprise by default entrusts the NFs with *both read/write access* over *entire* packets.

Need to protect NF source code. This model threatens the security of the NF vendors as well, who have a business interest in maintaining the privacy of their code and rulesets (often baked into the source code) from both the cloud and the enterprise. NFs have traditionally been shipped as hardware devices, so being shipped as software now exposes them further to untrusted platforms (*e.g.*, it is possible to reverse binaries).

The question is: how can we design an NF processing framework that meets all these security goals?

There has been little prior work in this space, consisting of mostly two approaches. Cryptographic approaches such as BlindBox [63] and Embark [38] are significantly limited in functionality, supporting only simple functions such as $=$ and $>$. They are unable to support more sophisticated operations such as regular expressions (needed in common NFs such as intrusion detection systems) or process custom NF code. Least-privilege approaches such as mcTLS [48] aim to give each NF access to only part of the packet and are designed for *hardware middleboxes*; however, when used in the cloud setting, they provide weak guarantees because the cloud receives the *union* of the permissions of all middleboxes, which often, is everything. Neither of these approaches protects

the NF source code, and both require significant changes to TLS, which is an impediment to adoption.

We present SafeBricks, a system for outsourcing NFs that provides protection with respect to the three security needs above. SafeBricks addresses the discussed limitations of prior work by supporting *generic* NF functionality with significantly stronger security guarantees, without requiring changes to TLS. It builds upon NetBricks [53], a framework for building and executing arbitrary NFs that uses a safe language and runtime, Rust.

To overcome the limited functionality of cryptographic approaches, SafeBricks *shields* [9] traffic processing from the cloud by executing the NFs within *hardware enclaves* (*e.g.*, Intel SGX [43]). This approach promises that neither an administrator with root privileges nor a compromised operating system can observe enclave-protected data in unencrypted form, or tamper with the enclave's execution. Enclaves have already been used to shield *general-purpose* computation from the cloud provider [3, 9, 30, 59]. Applying them to network processing is a natural next step, as recent proposals have pointed out (see §11).

While this idea is simple, designing a system that provides protection with respect to the three security goals above, and simultaneously maintains good performance, is far more challenging.

First, general-purpose approaches result in a large trusted computing base (TCB) inside the enclaves (up to millions of LoC), any vulnerability in which can result in information leakage. In SafeBricks, we investigate *how to partition* the code stack of NF applications (from packet capture to processing) and choose a boundary that reduces the code within the trusted domain without compromising security.

Second, partitioning an application is likely to result in *transitions* between enclave and non-enclave code. These transitions are expensive, introducing a high runtime overhead due to the cost of saving/restoring the state of the secure environment. Consequently, there is a tension between TCB size and the overall performance of the application: the lesser code the enclave contains, the more transitions it is likely to make to non-enclave code. In SafeBricks, we address these challenges simultaneously by developing an architecture that leverages shared memory and splits computation across enclave and non-enclave threads (while verifying the work of the non-enclave threads) without performing transitions.

Third, NFV deployments typically comprise multiple NFs running in a chain, isolated via VMs or containers for safety. In our setting, the straightforward way of achieving this isolation would be to deploy each NF in a separate enclave. However, as we discuss in §6, such an architecture can result in a system that is ∼2–16× slower than the baseline. Instead, SafeBricks supports chains of NFs within the *same* enclave. To isolate them, SafeBricks leverages the semantics of the Rust language.

Nevertheless, this strategy introduces a new difficulty: all NFs must be assembled using a trusted compiler. Though the client enterprise is the natural site for building the NFs safely, doing so would leak the source code of the NFs to the client, which is undesirable for the NF vendors. To address this challenge, SafeBricks runs in an enclave at the cloud a *meta*-functionality: a compiler that creates an encrypted binary, and a loader that runs this binary in a separate enclave. Using the remote attestation feature of hardware enclaves, both the NF vendors and the client can verify that the agreed-upon compiler and loader are running in an enclave, before the vendors share the NF code and the client shares data and traffic.

Finally, none of the above satisfies our requirement for enforcing least privilege across NFs: each NF still has access to entire packets. SafeBricks enforces least privilege by (i) exposing an API to the client for specifying the privileges of each NF, and (ii) ensuring that the SafeBricks framework *mediates* all NF accesses to packets, both reads and writes. To enforce the latter, SafeBricks leverages the safety guarantees of Rust.

We evaluate SafeBricks across four different NF applications using both synthetic and real traffic. Our evaluation shows that the performance impact of SafeBricks is reasonable, ranging between ∼0–15% across NFs.

2 Model and Threat Model

As shown in Figure 1, there are four types of parties in our setting: (1) a *cloud provider* that hosts the outsourced NFs; (2) a *client enterprise* outsourcing its traffic processing to the cloud; (3) *two endpoints* that communicate over the network, at least one of which is within the enterprise; and (4) *NF vendors* that supply the code and rulesets for network functions.

The client enterprise contains a gateway (as shown in Figure 2) which is trusted. The endpoints are trusted only with their communication.

The core of SafeBricks's design builds on the abstract notion of a hardware enclave. Our implementation uses Intel SGX [43], a popular hardware enclave, but few design decisions are tailored to SGX. We provide some relevant background on hardware enclaves, and then define the threat models for the cloud and the NF vendors.

2.1 Hardware enclaves

Hardware enclaves aim to provide an isolated execution environment that preserves the confidentiality and integrity of code and data within the enclave. An important feature of hardware enclaves is remote attestation.

Remote attestation. This procedure allows a remote client system to cryptographically verify that specific software has been securely loaded into an enclave, us-

ing CPU-based attestation [2]. When a client requests remote attestation, the enclave generates a report signed by the processor that contains a hash measurement of the enclave. As part of the attestation, the enclave can also bootstrap a secure channel with the client by generating a public key and returning it with the signed report.

Intel SGX. Intel Software Guard Extensions [43] is a set of ISA extensions that enables the creation of hardware enclaves. Software running outside the enclave, including privileged software such as the kernel or hypervisor, cannot access or tamper with enclave memory.

2.2 Threat model for the cloud and enclaves

Our threat model for the cloud provider is similar to prior works [3,9,59] that build on hardware enclaves. Enclaves strive to provide an abstract security guarantee so that systems like SafeBricks can build on them in a black-box manner; however, current implementations do not yet fully achieve this guarantee as we discuss below.

Abstract enclave assumption. The attacker cannot observe any information about the protected code and data in the enclave, and the remote attestation procedure establishes a secure connection between the correct parties and loads the desired code into the enclave.

Attacker capabilities. Except the out-of-scope attacks described below, we consider an attacker that can compromise the software stack of the cloud provider outside the enclave, which includes privileged software such as the hypervisor and kernel. In particular, whenever the enclave exits or invokes code outside the enclave, the attacker can instead run arbitrary code and/or respond with arbitrary data to the enclave. For example, the OS can mount an Iago attack [15] and respond incorrectly to system calls. Note that this threat model implies that the attacker can observe communication between hardware enclaves as well as communication on the network.

Out-of-scope attacks. In short, all attacks that violate the abstract enclave assumption above are out of scope for SafeBricks. For example, we consider as out of scope all hardware and side-channel attacks, as well as assume that the enclave manufacturer (*e.g.*, Intel) is trusted. Intel SGX's current implementation does not fully achieve the enclave assumption above because it suffers from side-channel attacks, including those based on access pattern leakage amongst others [12, 14, 17, 26, 28, 39, 47, 60, 73, 74]. While these are important issues with SGX, we treat them as out of scope for SafeBricks because solutions to these are orthogonal and complementary to our contribution here. Recently, a number of solutions have been proposed for solving or mitigating these attacks [16, 18, 27, 65, 66].

2.3 Threat model for network functions

Each NF is trusted only with the permissions given to it by the enterprise for specific packet fields. That is, if the enterprise gives a NAT read/write permissions for the IP header, the NF is trusted to not leak the header to unauthorized entities and to modify it correctly. At the same time, if the NAT attempts to access the packet payload, then SafeBricks must prevent it from doing so.

3 SafeBricks: End-to-end Architecture

APLOMB [62] discusses in detail the architecture for outsourcing NF processing to the cloud by redirecting client traffic, as well as the merits of this architecture. Here, we focus on how SafeBricks enhances this architecture with protection against cloud attackers and TLS compatibility, while maintaining performance.

SafeBricks supports three typical architectures considered in the cloud outsourcing model [38, 62], as shown in Figure 2. These architectures have different merits or constraints, and are useful for different cases.

Let S be the source endpoint, G the client gateway, CP the cloud provider running NFs using SafeBricks (SB), and D the destination endpoint. Let G_1 be the gateway near the source, and G_2 be the gateway near the destination. Note that in the Direct architecture, an enclave in the cloud plays the role of G_2, and in the Bounce architecture, a single gateway plays both G_1 and G_2. CP runs hardware enclaves; code and data are decrypted inside enclaves, but remain encrypted outside. D could either be an external site or an endpoint in another enterprise.

1. **Bounce:** In the bounce architecture, SB tunnels processed traffic to G over the secure channel. G then forwards the processed traffic to the destination. The response from D is similarly redirected by G to SB before forwarding it to S. The bounce setup is the simplest in that it does not place any added burden on SB or D from a functionality and security perspective. However, it inflates the latency between S and D as a result of bouncing the processed traffic to G.
2. **Direct:** The direct architecture alleviates the latency added by the bounce setup. SB directly forwards the enterprise traffic to D after processing it without bouncing it off the gateway. However, this setup comes at the cost of security: since there is no secure channel between SB and D over which traffic can be tunneled, SB must necessarily send the processed packets to D in the clear, revealing the headers to CP. If S and D use TLS, CP will not see the payload.
3. **Enterprise-to-enterprise:** If S and D belong to the same enterprise or to enterprises that trust each other, it is possible to have the combined benefits of the bounce and direct architecture. SB tunnels the processed traffic to G_2, so CP does not see any headers at any time. At the same time, this approach does not suffer from the bounce setup's latency.

Though not the focus of this work, it is worth mentioning that SafeBricks can also be used in a local cloud

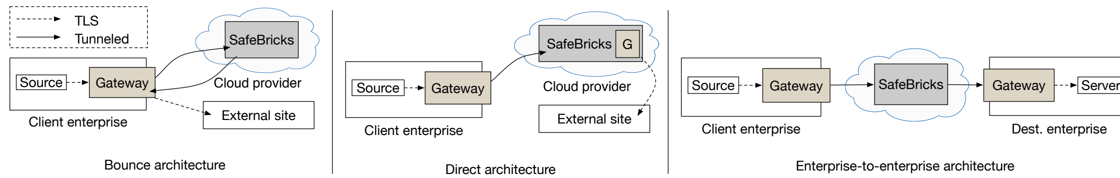


Figure 2: End-to-end system architecture

deployment in which the NFs run within the client enterprise. This benefits the client by providing SafeBricks's isolation and least privilege for NFs, as well as protection against administrators of the local cloud (although the gateway administrators need to remain trusted).

3.1 Overview of the communication protocol

Our protocol for handling connections is the same for all architectures, as we now explain in terms of G_1 and G_2.

System bootstrap. The client enterprise first sets up and verifies the enclaves in the cloud as explained in §7. As part of this process, the gateways are able to set up a set of IPSec tunnels with the cloud in a secure way (such as installing certificates to avoid the risk of a man-in-the-middle attack). To load-balance flows at the cloud server via receive-side scaling (RSS), the number of IPSec tunnels depends on the number of ports at the server.

As with all such interception systems, the source endpoints need to be configured to allow interception. The most common approach is to use an *interception proxy* [34], in which the sources' browsers accept certificates from the proxy which can now terminate the TLS connection. Another approach is to install a browser plugin at the client endpoints, which sends the TLS session keys to the gateway [29] over a secure channel. SafeBricks supports both these approaches.

Upon a new TLS connection from a source. G_1 terminates the connection as described above and informs G_2, which starts the TLS connection to the destination.

Packet processing. G_1 intercepts the TLS traffic from S, decrypts it, and tunnels it over an IPSec connection. Packets from the same flow are sent on the same IPSec connection. Note that as part of this process, the entire packet (including the header) is encrypted and encapsulated in a new header. We use AES in GCM mode as the IPSec encryption algorithm, which includes packet authentication. *SB* receives the packets, decrypts, and processes them. It then tunnels the packets over IPSec to G_2. G_2 terminates the IPSec tunnel and forwards the traffic over TLS to the destination server.

4 Background

Before delving into the design of SafeBricks, we provide a brief overview of NetBricks and some additional details on Intel SGX relevant to our system.

4.1 Intel SGX

Illegal enclave instructions. SGX does not allow instructions within an enclave that result in a change of privilege levels (*e.g.*, system calls) or cause a `VMEXIT`. Applications that need to perform such instructions must exit the enclave and transfer control to host software.

Memory architecture. Enclave pages reside in a protected memory region called the enclave page cache (EPC), whose size is limited to ∼94MB in current hardware. EPC pages are decrypted when loaded into cache lines, and integrity-protected when swapped to DRAM.

4.2 NetBricks

The NetBricks framework [53] enables the development of arbitrary NFs by exposing a small set of customizable programming abstractions (or operators) to developers. In this respect, NetBricks is similar to Click [37], which also enables developers to write NFs by composing various packet processing elements. However, we choose to build our system atop NetBricks instead of Click for the following reasons:

- Unlike Click, the behavior of NetBricks' operators can be heavily customized via user-defined functions (UDFs). This allows us to protect a small number of operators within the enclave (with NetBricks), which are then composed into NFs, as opposed to routinely adding new Click modules.
- More importantly, NetBricks builds upon a safe language and runtime, Rust, to provide isolation between NFs chained together in the same process. In §6, we describe how SafeBricks extends these guarantees to provide least privilege across NFs inexpensively.
- NetBricks' zero-copy semantics also improve performance substantially [53].

We now briefly describe some features of NetBricks relevant to the design of our system.

Programming abstractions. To construct an NF, the developer specifies a directed graph consisting of NetBricks' operators as nodes. For example, the `parse` operator casts packet buffers into protocol structures; `transform` modifies packet buffers; and `filter` drops packets based on a UDF. All nodes in the NF graph process packets in batches.

Execution environment. The NetBricks scheduler implements policies to decide the order in which different nodes process their packets. Chains of NFs are run in a single process by composing their directed graphs to-

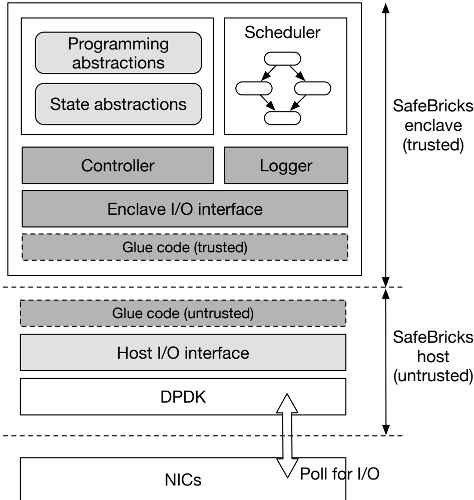


Figure 3: SafeBricks framework: White boxes denote existing NetBricks components, light grey boxes denote modified components, and dark grey boxes denote new components.

gether as function calls, instead of running each NF separately in a container or VM. For isolation between NFs, NetBricks relies on a safe language and runtime, Rust.

Packet I/O. NetBricks builds on top of DPDK [32], a fast packet I/O library. DPDK polls packets from the network devices, buffers them in pools of memory, and maintains a queue of pointers to the packet buffers. NF instances query DPDK via an I/O interface to retrieve pointers to the next batch of packet buffers, and process them in-place without performing any copies.

5 SafeBricks: Framework Design

We now describe how we build our system on top of NetBricks (while redesigning some parts of it). Figure 3 shows the overall design of the framework, highlighting the components modified or introduced by SafeBricks. Our goal in this section is to reduce the size of the TCB while minimizing the overhead of transitions between the enclave and the host. However, these two goals are often at odds with each other—the lesser code the enclave contains, the more transitions it makes to outside code. We now describe how our design balances both these aims.

5.1 Partitioning NetBricks

We carefully split NetBricks into two components—enclave code and host code.

SafeBricks enclave. At a bare minimum, the enclave should include the programming and state abstractions of NetBricks. However, during execution, the NetBricks scheduler takes decisions regarding which node to process next in the directed graph representing the NF (as described in §4.2). These decisions are frequent—every time a node is done processing a batch of packets, it surrenders control to the scheduler. As a result, excluding the scheduler from the TCB would result in a large number of enclave transitions per packet batch. Hence, we include the scheduler in our TCB as well.

SafeBricks host. The remaining components of NetBricks (mostly pertaining to packet I/O) together form the SafeBricks host. As described in §4.2, NFs in NetBricks directly access the packet buffers allocated by the packet capture library (DPDK) without copying them. Simply excluding DPDK from the enclave without other modifications is not a viable option because it would gain access to the packets once they are decrypted. On the other hand, including DPDK within the enclave would drastically inflate the size of the TCB by ~516K LoC.

We circumvent this issue by introducing two new operators in NetBricks: `toEnclave` and `toHost`. The `toEnclave` operator polls the I/O interface for pointers to packet buffers, reads the encrypted buffers from DPDK-allocated memory and decrypts them inside the enclave. Once the processing is complete, the `toHost` operator re-encrypts the packet buffers and returns them outside the enclave into DPDK's memory pool.

More concretely, `toEnclave` and `toHost` implement endpoints of the IPSec tunnel. As a result, even if the host attacker attempts Iago attacks [15] such as modifying packet buffers or queues outside the enclave, these will be detected by the authenticity provided by IPSec.

Excluding DPDK from the TCB enables us to remove NetBricks' I/O module from the TCB as well. The module interfaces with the packet capture library and is used by the NFs to poll DPDK for packets (Figure 3).

5.2 Packet I/O avoiding enclave transitions

Every receive or send operation for a batch of packets results in an invocation of the I/O interface. Since we exclude the packet capture library from the TCB, every such invocation necessarily results in an enclave transition. Batch processing of packets alleviates the overhead of these transitions to some extent, but as we show in §9.2.1, it is far from being a perfect solution.

Prior works [3, 50] have also explored the reduction of enclave transitions, albeit in a different context—they allow enclave threads to delegate system calls to the host with the help of shared queues. In a similar spirit, we propose an alternative design point that allows enclave code to receive and send packet batches from the host via shared memory, without the need for enclave transitions. To do so, we (i) introduce an additional trusted I/O module within the enclave (called `EnclaveIO`) that exposes the I/O APIs transparently to the rest of enclave code, and (ii) modify the NetBricks I/O interface outside the enclave (`HostIO`) to appropriately interface with the `EnclaveIO` module.

SafeBricks allocates two lockless circular queues (`recvq` and `sendq`) on heap memory outside the enclave during the application's initialization, one for re-

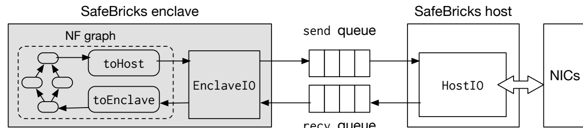


Figure 4: Packet I/O via shared memory.

ceiving pointers to packet buffers and the other for sending. `HostIO` busy polls DPDK for incoming packets and populates `recvq` with the buffer addresses. Enclave code queries the `EnclaveIO` module which in turn reads the packet buffer addresses directly from `recvq` without having to exit the enclave. To send packets, `EnclaveIO` pushes the packet buffer addresses into `sendq`. `HostIO` consumes the buffers asynchronously from this queue, and finally invokes the I/O interface to emit the packets to the network. Figure 4 illustrates the approach.

This mechanism doesn't result in any enclave transitions because (i) enclave code can readily access memory outside the enclave, and (ii) the queue management is asynchronous—the `HostIO` module and the SafeBricks enclave (containing `EnclaveIO`) run in separate threads.

5.3 System calls and other illegal instructions

As described in §4.1, SGX allows neither system calls within enclaves nor instructions that could lead to a `VMEXIT` (such as `rdtsc`, used for reading the timestamp counter). There exist a set of general-purpose systems [3, 9, 30, 50, 67, 72] that add support for such system calls to enclave applications, at the expense of added complexity and/or a significant increase in TCB.

We note that many NFs simply do not make system calls or execute instructions that require VM exits, and those made are typically only of a few types: such as I/O for maintaining logs, or timestamp measurements using `rdtsc`. For example, no application in the NetBricks or Bess source trees [11, 49] implements system calls. This is due to the high-performance goal of NFs, aiming to run exclusively in user-space [32, 33, 52, 57, 70]. The same extends to user-space implementations of networking stacks as well, which are gaining in popularity [20, 35, 40, 41, 46]. Therefore, instead of exposing an exhaustive API within the enclave for these instructions, SafeBricks focuses only on the operations essential for NFs and executes them without the need for enclave transitions. SafeBricks does not expose any other system calls or illegal instructions that would require enclave exits to NFs within the enclave.

Logging. Instead of allowing NFs to write to files, we expose a new state abstraction in SafeBricks that enables them to directly push logs to queues allocated in heap memory outside the enclave (similar to how we perform packet I/O). During system initialization, the Logger module allocates a queue in non-enclave heap per NF that logs information. NFs can push log entries to the respective queue by invoking the Logger module. Host code asynchronously reads the logs off these queues and writes them to files.

However, since this heap memory is untrusted and visible outside the enclave, we need to take additional steps to ensure the security of the logs (as they contain sensitive packet information). We encrypt and chain together log entries via authentication tags, a fairly standard technique. The Logger module encrypts each log item L_i as $C_i = \mathsf{Enc}(\mathtt{id}||L_i)$, where `id` identifies the NF. It then computes the authentication tag $T_i = \mathsf{Auth}(C_i||T_{i-1})$, and pushes (C_i, T_i) to the log queue. Including the previous tag in the computation ensures that host code cannot arbitrarily drop or reorder log items. The Logger module maintains the root authentication tag within the enclave. Verifiers can later validate the log by obtaining the latest tag from the enclave over a secure channel and replaying the log. We note that by itself, the approach doesn't prevent rollback attacks on the logs; however, techniques for avoiding such attacks exist and can be deployed in a complementary fashion [69].

Timestamps. SafeBricks relies on the `HostIO` module to capture the timestamp per incoming packet batch and write it to a slot in the packet buffer reserved for external metadata. NFs that need timestamps for their functionality simply read it off the packets. This approach also reduces latency when chains of NFs are deployed together, as the cost of measuring the timestamp is borne only once. Though it is possible to ensure the monotonicity of timestamps, SafeBricks does not guarantee that the timestamps are correct—this is unavoidable in the current SGX implementation as the reporting module is not trusted hardware.

5.4 Execution model

SafeBricks runs the NFs in a multi-threaded enclave, each enclave thread affinitized to a core. We note that our shared memory mechanism for packet I/O adds extra burden on system resources compared to vanilla NetBricks, as it requires an extra thread for running the `HostIO` module. This cost, however, gets amortized by mapping a single `HostIO` instance to multiple enclave threads.

6 SafeBricks: NF Isolation, Least Privilege

SafeBricks gives enterprises the flexibility to source NFs from different vendors and deploy them together on the same platform, while isolating them from each other and controlling which parts of a packet each NF is able to read or write. For example, consider a chained NF configuration wherein traffic is first passed through a firewall, then a DPI, and finally a NAT. The firewall application only needs read access to packet headers; the DPI needs read access to headers and payload; while the NAT needs read and write access to packet headers.

SafeBricks ensures that each NF is given only the minimum level of access to each packet as required for their functions, *e.g.*, the firewall is unable to write to packet headers, or read/write to the payload. In other words, SafeBricks isolates NFs from one another while enforcing the principle of *least privilege* amongst them.

6.1 Strawman scheme

The importance of least privilege access to traffic has been recognized before in mcTLS [48], which relies on physical isolation of NFs and enforces least privilege by encrypting and authenticating each field of the packet separately using different keys. Each NF is given the keys only for fields that it needs access to. To allow read access, the NF is given the encryption keys; for writes, the NF is given the authentication keys as well. Packets are re-encrypted before being transferred from one NF to the other. In the mcTLS model, NFs are isolated by virtue of being deployed on separate systems (hardware or VMs). Correspondingly in our setting, it suffices to run each NF concurrently in a separate enclave isolating their address spaces, as shown on the left of Figure 5.

Such an approach, however, eliminates much of the performance benefits of the underlying NetBricks framework. In addition to adding significant overheads due to repeated re-encryption of packets, it requires packets to cross core boundaries between NF enclaves (for enclaves affinitized to separate cores). Together, this can result in a system that is up to ∼2–16× slower (as we show in §9.2.3). Instead, it would be ideal to keep all NFs in the same enclave and isolate them efficiently within.

6.2 NF isolation in NetBricks

Before describing how SafeBricks enforces least privilege across NFs, we revisit crucial properties of the Rust language that form the basis of our design.

The NetBricks framework provides isolation between NFs running in the same address space by building on a safe language, Rust [7, 53]. Rust's type system and runtime provide four properties crucial for memory isolation: (i) they check bounds on array accesses, (ii) prohibit pointer arithmetic, (iii) prohibit accesses to null objects, and (iv) disallow unsafe type casts.

In addition to memory isolation, NFs also require packet isolation; *i.e.*, NFs should not be able to access packets once they've been forwarded. NetBricks relies on Rust's unique types [7,25] to isolate packets. Rust enforces an *ownership model* in which only a unique reference exists for each object in memory. Variables acquire sole ownership of the objects they are bound to. When an object is transferred to a new variable, the original binding is destroyed. Rust also allows variables to temporarily *borrow* objects without destroying the original binding. By harnessing Rust's ownership model, NetBricks ensures that once an NF is done processing a packet, its ownership is transferred to the next NF and the previous NF can no longer access the packet.

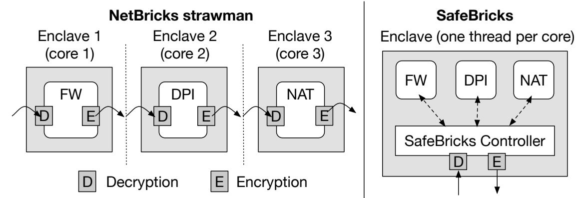


Figure 5: Strawman approach for enforcing least privilege versus SafeBricks. Solid arrows indicates packet transfers. Dotted arrows indicate interaction between NFs and the Controller.

```
pub fn chain<T:'static + Node>(input:T, pmap:HashMap) -> Node {
   let input = input.toEnclave()
      .wList(pmap.get('firewall'));
   let mut chain = firewall(input)
      .wList(pmap.get('dpi'));
   chain = dpi(chain)
      .wList(pmap.get('nat'));
   return nat(chain)
      .toHost();                                             }
```

Figure 6: Code for chaining NFs together (firewall, DPI, and NAT), generated automatically by SafeBricks from a configuration file. Lines in magenta represent code added by SafeBricks over and above NetBricks to enforce least privilege across NFs.

Taken together, the properties of NetBricks suffice for the purpose of running NFs safely within the same address space. However, they do not provide the desired security, as we explain next.

6.3 Isolating NFs within the same enclave

The properties of NetBricks do not satisfy the requirements of our threat model for the following reasons:

- The isolation guarantees only hold if NFs are built using a compiler that *enforces* the safety properties above. In our model, however, enterprises may source NFs from various vendors that compiled them in their own way and lack incentive to enforce these properties.
- Each NF still receives ownership of *entire packets*, instead of limited read / write access to specific fields.

We now describe how SafeBricks addresses both issues.

6.3.1 Ensuring memory safety

SafeBricks needs to ensure that NFs are built using a compiler that prohibits unsafe operations inside NFs. Instead of trusting NF providers, SafeBricks ensures that a trusted compiler gets access to the raw source code of all the NFs which it can then build in a trusted environment.

This strategy is seemingly in conflict with the confidentiality of NF rules. In §7 we show how SafeBricks performs this compilation such that neither the enterprise nor the cloud learns the source code of the NFs.

6.3.2 Enforcing least privilege

SafeBricks extends NetBricks' memory safety guarantees by interposing on its packet ownership model. Instead of transferring packets across NFs, SafeBricks in-

troduces a Controller module that mediates NF access to packets as depicted in Figure 5 (right).

Controlling access to packets. The Controller holds ownership of packet buffers, and NFs can only *borrow* packet fields (or different fragments of the data buffers) by submitting requests to the Controller. To provide least privilege, each packet in SafeBricks encapsulates a bit vector of *permissions*. Each function in the packet API exposed by the Controller is associated with a bit in the permissions vector. Before lending the NF a reference to the requested field, the Controller checks the corresponding bit in the vector and answers the request only if the bit is set. Otherwise, the call returns an error. Furthermore, by controlling whether an API call returns a *mutable* or an *immutable* reference, the framework also disambiguates read access from writes. Rust's type system ensures that once the NF processing completes, the binding between the reference and the field is destroyed, and any later attempt by the NF to access the field will result in a compilation error.

Setting packet permissions. SafeBricks updates the permissions vector in packets with the help of a new packet processing operator: `wList` (whitelist). Chained NFs are interleaved with invocations of the `wList` operator that applies a given vector of permissions to each packet batch before it's processed by the next NF. Figure 6 illustrates the code for chaining NFs together while enforcing least privilege. In §7, we describe how SafeBricks generates this code automatically using a configuration file supplied by the client enterprise.

We need to fulfill two more requirements for the guarantees to hold: (i) NFs should not be able to alter the permissions vector during execution, and (ii) NFs should not be able to parse packet buffers arbitrarily—for example, an NF that has permissions only for IP headers should not be able to incorrectly parse TCP headers as IP, thereby circumventing the policy. SafeBricks therefore does not expose these operations to NFs. NFs in NetBricks invoke the `parse` operator to cast packet buffers into protocol structures before processing them. In contrast, SafeBricks mandates that packets be parsed as required before being processed by NFs (not shown in Figure 6 for simplicity). In §7, we describe how the SafeBricks loader interleaves NFs with `parse` nodes and stitches them together into a directed graph based on enterprise-supplied configuration data.

Runtime overhead. The permissions vector leverages portions of the packet buffers reserved for metadata, and hence does not lead to any memory allocation overhead. Setting and verifying permissions, however, lead to a small overhead at runtime: setting the permissions vector before each NF via the `wList` operator increases the depth of the NF graph, and verifying the permission adds an extra check as all requests are mediated by the Controller. As we see in §9.2.3, the impact on performance is small for real applications.

7 SafeBricks: System Bootstrap Protocol

We now describe the protocol for bootstrapping the overall system. Instead of compiled binaries, SafeBricks needs access to the raw source code of the NFs from the providers so it can pass them through a trusted compiler, which ensures that NFs do not perform unsafe operations and are confined to least privilege access. The natural strategy is to have the client enterprise compile these binaries and upload them to the cloud, as in prior enclave-based systems such as Haven [9]. However, this approach is problematic in our case because NF code is proprietary and the client enterprise may not see it.

To address this problem, the idea in SafeBricks is to run inside the enclave a *meta*-functionality: the enclave assembles the NFs and *compiles them* using a trusted compiler, and only then starts running the resulting code. The key to why this works is that *both* the client enterprise and the NF vendors can invoke the remote attestation procedure to check that the enclave is running an agreed upon SafeBricks loader and compiler (both being public code). In this way, (i) each NF vendor can ensure that the enclave does not run some bad code that exfiltrates the source code to an attacker, and (ii) the client enterprise makes sure the NF vendor cannot change what processing happens in the enclave. The bootstrap process consists of two phases, assembly and deployment.

7.1 Phase 1: NF assembly

For assembly, SafeBricks uses a special enclave provisioned with two trusted modules—a loader and a compiler—that combine the NFs into a single binary.

Loader. The loader exposes a simple API that allows the client enterprise to specify (i) *encrypted* NF source codes, (ii) optionally, unencrypted NF source codes that might be interspersed with the proprietary encrypted NFs, (iii) a configuration file outlining the placement of each NF in the directed graph (when chained together), and (iv) a whitelist of permissions per NF indicating the fields each NF is allowed to access.

For the first two, the loader exposes the following API to the client: `load(name, code, is_encrypted)`. For the third, the client specifies the NF graph as a set of edges: ($\mathtt{name}_i \rightarrow \mathtt{name}_j$). For the fourth, the client supplies a configuration file with a list of items of type: (`name, op, proto:field`) where `op` $\in$ `[read, write]` and `proto:field` indicates a field within a protocol that access is given to. For example, for a firewall, one such entry is `(firewall, read, IP:src)`, in addition to entries for other fields of the IP header.

The loader decrypts the NFs and stitches them together based on the specified graph, before invoking the com-

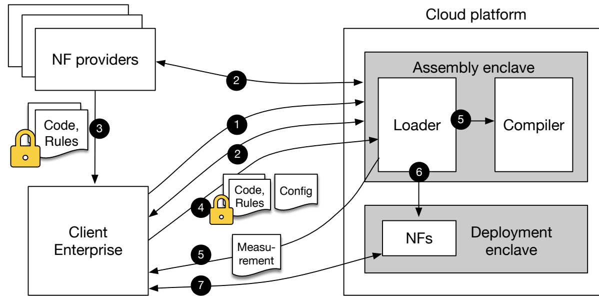


Figure 7: SafeBricks's NF assembly and deployment phases during bootstrap. Locks indicate that the data is encrypted.

piler. (In §7.2, we discuss how the enclave obtains the keys to decrypt this code.) In doing so, it adds the following additional nodes to the composite graph: (i) a `toEnclave` node at the root of the graph, (ii) a `toHost` node at the end of the graph, and (iii) `parse` nodes followed by a `wList` node before each NF. The loader infers the arguments to the `parse` and `wList` nodes automatically from the configuration file. Thus, `parse` is run by the trusted SafeBricks framework and not by an NF or the client enterprise, ensuring that the packets are not parsed in an unintended way.

Compiler. The compiler is a standard Rust compiler that implements a lint prohibiting unsafe code inside the enclave, as discussed in §6.3. Since launching the compiled binary requires OS support, the binary must be placed into main memory where the OS can access it post compilation. However, giving the OS access to the binary unencrypted would violate NF confidentiality.

In order to maintain the privacy of NF code while still allowing its execution by the OS, we take inspiration from VC3 [59]. Similarly to VC3, our compiler links the compiled NF code to a small amount of public code $\mathtt{NF}_{\mathtt{load}}$, and then encrypts the NF code because it will be placed in main memory for the OS to load in the deployment enclave. We refer to the encrypted code as $\mathtt{NF}_{\mathtt{priv}}$. Post compilation, $\mathtt{NF}_{\mathtt{load}} + \mathtt{NF}_{\mathtt{priv}}$ are loaded and run in a separate deployment enclave by the OS. $\mathtt{NF}_{\mathtt{load}}$ will be responsible for decrypting and interfacing with $\mathtt{NF}_{\mathtt{priv}}$ within the deployment enclave once it's initialized.

The loader and compiler are generic modules independent of the NFs. Hence, the NF providers need to audit them only once, across all customer deployments.

Assembly protocol. Figure 7 illustrates the assembly and deployment protocol. ❶ The cloud provisions an enclave with the SafeBricks loader and compiler modules. ❷ Next, the client as well as the NF providers verify that the loader and compiler have been securely provisioned into the enclave using the remote attestation feature of SGX, as described in §2.1. During the attestation, the enclave also returns a securely generated public key to each NF provider. ❸ Each provider then encrypts the NF source code and rulesets with the received public key and submits it to the client enterprise. ❹ The enterprise loads the encrypted source codes and rulesets along with configuration files into the enclave via APIs exposed by the loader module. ❺ The loader decrypts the source codes, stitches them together, and builds and encrypts the assembled code using the compiler, producing $\mathtt{NF}_{\mathtt{load}} + \mathtt{NF}_{\mathtt{priv}}$. It then returns to the client a hash measurement of the compiled code so that the client can later verify it once it's deployed in a separate enclave.

7.2 Phase 2: NF deployment

❻ The loader finally requests the OS to deploy $\mathtt{NF}_{\mathtt{load}} + \mathtt{NF}_{\mathtt{priv}}$ in a separate enclave on the cloud platform. It attests the deployed enclave, establishes a secure channel with $\mathtt{NF}_{\mathtt{load}}$, and transfers to it the decryption key for $\mathtt{NF}_{\mathtt{priv}}$. $\mathtt{NF}_{\mathtt{load}}$ decrypts the private code and starts execution. Note that since the assembly enclave attests the deployment enclave and the NF vendors attested the assembly enclave, the NF vendors are assured that the deployment enclave will not send the decrypted binary anywhere but merely run it. ❼ The client then attests the deployed enclave using the measurement it received at the end of the assembly phase, after which it establishes a secure channel of communication with the enclave.

8 Security Guarantees

We describe SafeBricks's guarantees assuming the threat model in §2, including the enclave assumption.

SafeBricks's main benefit to confidentiality is that it exposes only encrypted traffic to a cloud attacker, so the attacker does not see the contents of the packets and is limited to observing only packet sizes, timing, and NF access patterns to packets and data. SafeBricks protects in this manner the packet payload and, except in the direct architecture, the header as well.

As with any system with complex processing, encryption does not mean perfect confidentiality because of the existence of side-channels. In §2.2 we mentioned some categories of side-channels that SafeBricks, and SGX in general, does not protect against. In addition, there are a few other SafeBricks-specific side channels. First, an attacker in SafeBricks knows which (encrypted) packets belong to which flow because each flow is affinitized to an IPSec tunnel for scalability. If this issue is of concern, it can be fixed by using a single tunnel for all flows at the expense of performance. Second, an attacker can measure the time taken by NFs to process a batch of packets. This could leak information in some cases, *e.g.*, whether an expensive regular expression was triggered or not. This is a classical problem, already investigated by prior work [6, 13, 78] with common solutions involving padding, *i.e.*, bounding the running time of NFs by executing dummy cycles. Third, an attacker can learn

the action taken by an NF, *e.g.*, whether a connection was dropped simply by noticing that fewer packets were sent out. Like many other side-channels, this leakage can also be removed via padding—for example, the gateways could continue sending dummy traffic.

SafeBricks also protects the integrity of the traffic and of the NF processing. A cloud attacker cannot drop, insert, or modify packets, nor can it tamper with NF execution. Integrity of the NFs is guaranteed by SGX, while the integrity of the traffic is guaranteed by the IPSec tunnels between the enclave and the client.

Via the isolation and least privilege design, SafeBricks further ensures that each NF is confined to accessing only parts of the packet the enterprise desires. For the NF vendors, SafeBricks guarantees that NF source codes are hidden from all untrusted parties, including the client enterprise, a cloud attacker or other NF vendors.

8.1 Comparison to prior approaches

Prior approaches leak significantly more information about the traffic to the cloud provider than SafeBricks.

Cryptographic approaches. BlindBox [63] and Embark [38] encrypt the traffic in a special way that allows the cloud to match encrypted tokens against the traffic and detect if a match occurs. In these schemes, the cloud learns the offset at which any string from any rule in an NF occurs in the packet, regardless of whether or not the rule as a whole matched (rules often contain several such strings). If the rule is known (as in public rulesets), the attacker learns the exact string at that offset in the packet. Even if the rule string is not known, the attacker learns its frequency, which could lead to decryption via frequency analysis. Assuming an enclave employing side-channel protections as in §2.2, SafeBricks does not reveal this information. The attacker does not know which rule or part of a rule triggered on a packet. Moreover, BlindBox and Embark do not protect against *active attackers* who modify the traffic flow and, for example, drop packets.

We remark, however, that these prior approaches rely on cryptography alone, and not on trusted hardware as SafeBricks, which makes it much more challenging for them to achieve the properties SafeBricks achieves.

mcTLS [48] aims to provide least privilege in a setting where each NF is a separate hardware middlebox and belongs to a *different trust perimeter*. Running mcTLS in the cloud in software, however, removes essentially all its security guarantees: the cloud receives the *union* of the permissions of all NFs, which often, is everything.

9 Evaluation

We now measure the impact of SafeBricks on NF performance versus an insecure baseline. We also measure the reduction in TCB size as a result of our design. We do not discuss the performance of SafeBricks's gateway as the protocols it implements are well understood.

9.1 Setup

We evaluate the performance of SafeBricks using SGX hardware on a single-socket server provisioned with an Intel Xeon E3-1280 v5 CPU with 4 cores running at 3.7GHz. We disable hyperthreading for our experiments. The server has 64GB of memory, and runs Ubuntu 14.04.1 LTS with Linux kernel version 4.4. The hardware supports the SGX v1 instruction set which does not allow dynamic page allocation. Further, the total enclave page cache memory (EPC) available to all enclaves is limited to ∼94MB. For test traffic, we use another server that runs a DPDK-based traffic generator and is directly connected to the SGX machine via Intel XL710 40Gb NICs. The SGX machine acts as the cloud, and the traffic generator is both source and sink for the client traffic.

9.2 Performance

We evaluate the performance of SafeBricks using (i) synthetic traces of different packet sizes, from 64B to 1KB, and (ii) the ICTF 2010 trace [31], captured during a wide-area security competition and commonly used in academic research. We report throughput in millions of packets per second (Mpps). In all experiments, we exchange traffic between the traffic generator and the SGX machine over an encrypted tunnel (per §3). As a result, the size of each packet exchanged between the enterprise and the cloud increases by a fixed amount, equal to the headers added by the IPSec protocol.

We compare SafeBricks against an insecure baseline comprising vanilla NetBricks augmented with support for the encrypted tunnel. The baseline represents a setup in which traffic is sent to the cloud over an encrypted channel (hence safe from network attackers), but lacks the protection of SafeBricks at the cloud. Finally, we report the median of 10 iterations for each experiment.

9.2.1 Framework overheads

We first measure the overhead introduced by SafeBricks as a result of redesigning the core NetBricks framework. To illustrate the benefits of our architecture, we also compare the overheads of the strawman approach that performs packet I/O via enclave transitions (per §5).

The net overhead of both approaches varies with the complexity of NFs and the latency the NF introduces as a result of packet processing. In this experiment, we use CPU cycles as a proxy for NF complexity, and evaluate a simple NF that first modifies each batch of packets by interchanging the source and destination IP addresses, and then loops for a given number of cycles. We use packet batches of size 32 for both NetBricks and SafeBricks.

Figure 8 (left) presents the results with varying packet sizes when the NF is deployed on a single core, and Figure 8 (right) shows the performance for 64B packets when the deployment is scaled to two cores. In the worst

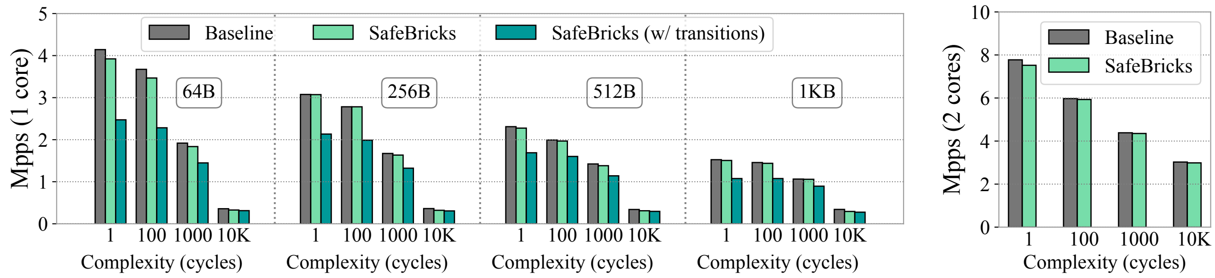


Figure 8: (**Left**) SafeBricks framework performance on 1 core compared to the baseline across different packet sizes, and with increasing NF complexity (*i.e.*, processing time in CPU cycles). (**Right**) Performance with 64B packets and NFs on 2 cores.

case with 64B packets and a delay of 1 cycle, the overhead introduced by SafeBricks is $< 5\%$. As the processing time begins to dominate (with increasing NF complexity), the overhead of SafeBricks becomes negligible.

The results also confirm that the design of SafeBricks outperforms the strawman approach, the overhead of which is ∼40% in the worst case. It's worth noting, however, that the relative overhead of the strawman approach decreases with larger packet sizes, as the rate of I/O falls.

9.2.2 Impact on real NFs

Unlike the simple NF in the previous experiment, real NFs have varying state requirements. Since the sizes of both the processor caches and enclave memory are limited, the overheads of SafeBricks are also governed by the memory access patterns of the NFs in addition to their complexity. In particular, L3 cache misses are more expensive for enclave applications because cache lines need to be encrypted/decrypted before being evicted/loaded. In this experiment, we characterize the effect of state on the performance of SafeBricks by evaluating the following sample applications:

- **Firewall**: We use a stateful firewall application that linearly scans a list of access control rules and drops connections if it finds a match. We evaluate it using a ruleset we obtained from our department (643 rules).
- **DPI**: We use a simple deep packet inspection (DPI) application that implements the Aho-Corasick pattern matching algorithm [1] on incoming packets, similar to the core signature matching component of the Snort IDS [58]. We evaluate the DPI using patterns extracted from the Snort Community ruleset [68].
- **NAT**: Our implementation is based on MazuNAT [42].
- **Load balancer**: We use a partial implementation of Google's Maglev [20], that spreads traffic between backends using a consistent hashing lookup table.

Figure 9 shows the normalized overhead of SafeBricks on application performance across different packet sizes with synthetic traffic. Figure 12 summarizes the worst-case results corresponding to 64B packets. Figure 12 also presents the performance results with the ICTF trace. Across applications, the overhead ranges between an acceptable ∼0–15% for both synthetic and real traffic, and is a result of page faults triggered by L3 cache misses.

Impact of larger memory footprint. In the previous experiment, the working sets of the applications exceeded the L3 cache but remained less than the size of the EPC (∼94MB). However, accessing memory beyond the EPC is doubly expensive because evicted EPC pages need to be encrypted and integrity-protected. We now assess the impact of a large memory footprint using the DPI application. The application builds a finite state machine over all the patterns in the ruleset, and as such has a significantly larger memory footprint than other NFs.

Figure 10 shows the results of our experiment using an increasing number of rules from the Emerging Threats ruleset [21] and the ICTF trace. At 18K rules, the working set of the DPI breached the ∼94MB EPC boundary causing its performance to sharply deteriorate thereafter.

This experiment indicates the limits of SafeBricks with regard to the nature of applications it can efficiently support. However, we note that the ∼94MB limit is only an artifact of existing hardware and isn't fundamental to SGX enclaves. The next generation of SGX machines is likely to support larger EPC sizes.

9.2.3 Cost of NF isolation

We now evaluate the overhead as a result of our mechanisms for enforcing least privilege. Given a chain of NFs, SafeBricks increases the overall depth of the NF graph by one node per NF (§6.3). In this experiment, we first measure this extra cost as a function of the length of the NF chain. We then compare our approach against an mcTLS-like strawman that relies on encryption for selectively exposing packet fields to NFs (§6.1).

Effect of chain length. For this part of the experiment, we use a simple NF that decrements the time-to-live (TTL) field in the IP header of each packet, composed together into chains of varying length. Before executing subsequent NFs in the chain, SafeBricks whitelists access to the TTL field in the permissions vector per packet.

Figure 11 compares the performance of SafeBricks with and without least privilege. Since the NF is stateless, in the absence of isolation SafeBricks does not introduce any discernible overhead against the baseline. With least privilege enforcement, the latency added

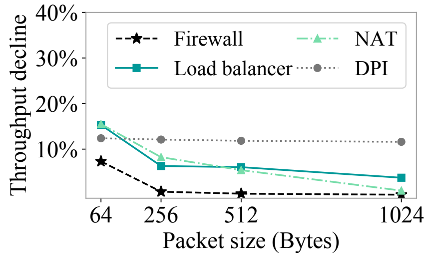


Figure 9: Normalized overhead (Mpps) across NFs for different packet sizes.

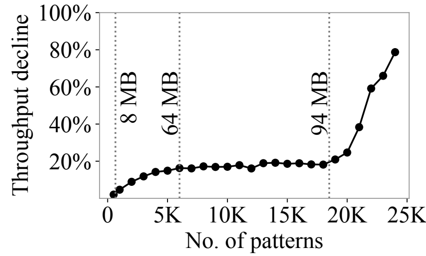


Figure 10: DPI performance (Mpps) using the ICTF trace, with increasing no. of rules.

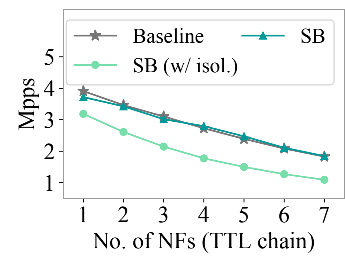


Figure 11: Cost of least privilege with increasing no. of NFs.

	Synthetic (64B packets)		ICTF trace	
NF	**Baseline**	**SB**	**Baseline**	**SB**
Firewall	3.86	3.58	1.96	1.93
DPI	1.10	0.96	0.29	0.25
NAT	3.80	3.21	1.97	1.80
Maglev	3.59	3.04	1.92	1.73

Figure 12: Performance of sample NFs (Mpps)

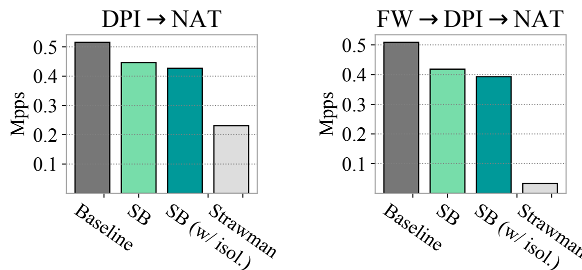


Figure 13: Cost of least privilege across NF chains (2 cores)

by the additional nodes increases as the length of the chain increases. Consequently, the overhead climbs from ∼14–40% as the chain grows to a size of seven NFs. We note that these numbers represent an upper bound on the overhead of SafeBricks. As we show in the next part of this experiment, the percentage overhead is much smaller for real, more complex NFs.

Comparison with encryption-based strawman. We now measure the performance of SafeBricks using a chain of real NFs each of which accesses different parts of packets—a firewall (given read permissions on packet headers), a DPI (with read permissions on both headers and payload), and a NAT (with read and write permissions on packet headers). The NF implementations are identical to the ones described in §9.2.2.

To quantify the benefit of our approach for enforcing least privilege, we also compare SafeBricks to an mcTLS-like strawman in which each NF in the chain is run in a separate enclave (as described in §6.1). In all setups (including the baseline), we allocate two cores for running the NFs, and reserve one core for I/O.

Figure 13 shows the results with two different chains: (i) a DPI followed by a NAT, and (ii) a firewall chained to a DPI and then a NAT. In the former scenario, SafeBricks results in an overhead of 15% in the absence of least privilege enforcement. With least privilege, the throughput declines by a further 3%, confirming that the cost of enforcing least privilege across real NFs is minimal. In contrast, an mcTLS-like approach (with each NF running in a separate enclave, affinitized to distinct cores) results in a sharper decline of 2.2× the performance being bottlenecked at the DPI along with the added encryption and copying of packets as they move across NFs in different enclaves. In the latter scenario with three NFs in a chain, the performance of the strawman approach falls further, by 16×. In this scenario, however, the NFs (and hence enclaves) outnumbered the available cores in our setup, leading to resource contention.

9.3 Comparison with BlindBox and Embark

Both SafeBricks and Embark tunnel packets to a third-party service in the cloud. For the ICTF trace, IPSec tunneling inflates the bandwidth by 16% due to both encryption and encapsulation. Embark introduces a further 20-byte overhead per IPv4 packet because it converts them to IPv6, resulting in a net overhead of 21%. BlindBox, in contrast, does not pay the cost of tunneling as it is targeted at in-network DPI applications. However, the BlindBox encryption protocol (also used by Embark for DPI processing) inflates bandwidth consumption by up to 5× in the worst case, unlike SafeBricks which only uses standard encryption schemes.

As regards throughput, both Embark and BlindBox are competitive with unencrypted baseline NFs and incur negligible overhead, whereas SafeBricks impacts performance by ∼0–15% across NFs due to its use of SGX enclaves (§9.2.2). At the same time, both BlindBox and Embark impact performance at the client considerably—with BlindBox, client endpoints need to implement its special encryption protocols over and above TLS and take 30× longer to encrypt a packet; Embark centralizes this overhead at the enterprise's gateway instead. Clients do not need to pay these costs with SafeBricks.

9.4 TCB size

SafeBricks involves the use of two types of enclaves: one for assembling the NFs during system bootstrap (per §7), and another for deploying the NFs. The assembly enclave primarily contains the Rust compiler, which is nec-

essarily part of the TCB of applications with or without SafeBricks. The deployment enclave, on the other hand, represents the TCB which we aim to reduce in redesigning the NetBricks framework.

To evaluate the reduction in TCB, we thus compare the size of the deployment enclave components in SafeBricks with that of NetBricks. The size of the enclave binary in SafeBricks is ∼1MB. In comparison, the aggregate size of NetBricks components is 21.3MB, representing a TCB reduction of over 20×. The reduction can largely be attributed to the exclusion of DPDK from the TCB as a result of partitioning NetBricks, which itself comprises ∼516K LoC. Furthermore, by designing for our specific use case, we avoid including a library OS within our trust perimeter, the size of which can be as large as 209MB (as in Haven [9]).

10 Limitations and Future Work

SafeBricks inherits three primary limitations owing to its use of Intel SGX.

First, enclave memory is limited to ∼94MB in existing hardware, making SafeBricks impractical for applications with larger working sets. Exploring alternate architectures that combine cryptographic approaches and SGX, thereby reducing the memory burden on the enclaves, is an interesting open problem in this context.

Second, SafeBricks is unsuitable for NFs relying on operations that are illegal within SGX enclaves, such as system calls and timestamps. Though SafeBricks supports timestamps, it can only ensure their monotonicity and not correctness.

Third, SGX enclaves, and consequently SafeBricks, are vulnerable to side-channel attacks (per §2.1). Though a number of potential solutions have been proposed in recent work [16, 18, 27, 65, 66], their impact on application performance is often non-trivial. Investigating the viability of these proposals in the NFV context, or developing targeted solutions for NFs is potential future work.

11 Related Work

We divide related work largely into two categories: (i) cryptographic approaches for securing NFs, and (ii) proposals based on trusted hardware. We do not discuss the mcTLS protocol [48] further as we have already compared SafeBricks with mcTLS in §6 and §8.

Cryptographic approaches. Recent systems propose the use of cryptographic schemes that enable NFs to operate directly over encrypted traffic [5, 38, 44, 63, 76]. When compared to SafeBricks, these approaches have the advantage that they do not rely on trusted hardware. However, this comes with two significant limitations. (1) Their functionality is severely constrained, as discussed in §1, and hence are not applicable to a wide range of NFs. To provide full functionality with cryptography, one needs schemes such as fully-homomorphic encryption [23], which is orders of magnitude too slow. (2) Regarding security, we explained in §8 how these systems leak more information to the cloud than SafeBricks.

Trusted hardware proposals for legacy applications. Other work has shown how to use hardware enclaves to run applications in the cloud without having to trust the cloud provider [3, 9, 30, 50, 67, 72]. The mandate of these systems is to support arbitrary, legacy applications instead of optimizing for any in particular. As a result, some of these systems inflate the size of the TCB by introducing a library OS within the enclave (to support illegal enclave instructions), or impact performance because of enclave transitions.

Trusted hardware proposals for network applications. Recent work has proposed the use of hardware enclaves for securing network applications. Kim *et al.* use SGX to enhance the security of Tor [61], and also identify NFs as a potential use case [36]. Other proposals develop prototypes for specific functions: Coughlin *et al.* [19] present a proof-of-concept Click element for pattern matching within enclaves; and Shih *et al.* [64] propose SGX for isolating the state of NFs, applying it to a subset of the Snort IDS. In contrast, SafeBricks is a *general*-purpose framework that additionally enforces least privilege across NFs. At the same time, SafeBricks balances the interests of NF vendors by maintaining the confidentiality of NF code and rulesets.

Concurrent to our work, SGX-Box [29] and ShieldBox [71] also propose frameworks for executing NFs within enclaves. SGX-Box [29] does not explicitly handle NF isolation or chaining; ShieldBox integrates SGX with Click and isolates each NF in a separate enclave. In such cases, ShieldBox reports a throughput decline of up to 3×. SafeBricks, in contrast, avoids this overhead by isolating NFs within the same enclave with the help of language-based enforcement. However, unlike SafeBricks, ShieldBox also supports NFs with system calls by leveraging the Scone framework [3]. Both SGX-Box and ShieldBox also allow NFs to access entire packets, while SafeBricks enforces least privilege.

Acknowledgments

We thank our shepherd, Emin Gün Sirer, and the anonymous reviewers for their helpful comments. Aurojit Panda helped us with NetBricks; Jethro Beekman helped us with his SGX SDK; Assaf Araki and Intel supplied the SGX cluster; Jon Kuroda helped us manage our testbed. We are also grateful to Chia-che Tsai, Mona Vij, Amin Tootoonchian, Mihai Christodorescu, Rohit Sinha, and Takeshi Mochida for valuable discussions and feedback. This work was supported by the Intel/NSF CPS-Security grants #1505773 and #20153754, the UC Berkeley Center for Long-Term Cybersecurity, and gifts to the RISELab from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

[1] AHO, A. V., AND CORASICK, M. J. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* (1975).

[2] ANATI, I., GUERON, S., JOHNSON, S. P., AND SCARLATA, V. R. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[3] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M. L., GOLTZSCHE, D., EYERS, D., KAPITZA, R., PIETZUCH, P., AND FETZER, C. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[4] Aryaka. http://www.aryaka.com/.

[5] ASGHAR, H. J., MELIS, L., SOLDANI, C., DE CRISTOFARO, E., KAAFAR, M. A., AND MATHY, L. SplitBox: Toward Efficient Private Network Function Virtualization. In *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization (HotMiddlebox)* (2016).

[6] ASKAROV, A., ZHANG, D., AND MYERS, A. C. Predictive Black-box Mitigation of Timing Channels. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2010).

[7] BALASUBRAMANIAN, A., BARANOWSKI, M. S., BURTSEV, A., PANDA, A., RAKAMARIĆ, Z., AND RYZHYK, L. System Programming in Rust: Beyond Safety. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)* (2017).

[8] Barracuda Networks. https://www.barracuda.com/.

[9] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[10] BEEBY, D. Rogue tax workers snooped on ex-spouses, family members, 2010. https://goo.gl/WNKoCS.

[11] Berkeley Extensible Software Switch (BESS). http://span.cs.berkeley.edu/bess.html.

[12] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)* (2017).

[13] BRAUN, B. A., JANA, S., AND BONEH, D. Robust and Efficient Elimination of Cache and Timing Side Channels. *arxiv:1506.00189* (2015).

[14] BULCK, J. V., WEICHBRODT, N., KAPITZA, R., PIESSENS, F., AND STRACKX, R. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2017).

[15] CHECKOWAY, S., AND SHACHAM, H. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[16] CHEN, S., ZHANG, X., REITER, M. K., AND ZHANG, Y. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Proceedings of the ACM Asia Conference on Computer & Communications Security (AsiaCCS)* (2017).

[17] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086* (2016). http://eprint.iacr.org/2016/086.

[18] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2016).

[19] COUGHLIN, M., KELLER, E., AND WUSTROW, E. Trusted Click: Overcoming Security Issues of NFV in the Cloud. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security)* (2017).

[20] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[21] Emerging Threats Open Rulesets. https://rules.emergingthreats.net/.

[22] Fortinet. https://www.fortinet.com/.

[23] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)* (2009).

[24] GOOGLE. Transparency Report. https://www.google.com/transparencyreport/userdatarequests/US/.

[25] GORDON, C. S., PARKINSON, M. J., PARSONS, J., BROMFIELD, A., AND DUFFY, J. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2012).

[26] GÖTZFRIED, J., ECKERT, M., SCHINZEL, S., AND MÜLLER, T. Cache Attacks on Intel SGX. In *Proceedings of the European Workshop on Systems Security (EuroSec)* (2017).

[27] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2017).

[28] HÄHNEL, M., CUI, W., AND PEINADO, M. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2017).

[29] HAN, J., KIM, S., HA, J., AND HAN, D. SGX-Box: Enabling Visibility on Encrypted Traffic Using a Secure Middlebox Module. In *Proceedings of the Asia-Pacific Workshop on Networking (APNet)* (2017).

[30] HUNT, T., ZHU, Z., XU, Y., PETER, S., AND WITCHEL, E. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[31] ICTF data. https://ictf.cs.ucsb.edu/.

[32] Intel Data Plane Development Kit (DPDK). http://dpdk.org/.

[33] JAMSHED, M. A., MOON, Y., KIM, D., HAN, D., AND PARK, K. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[34] JARMOC, J. SSL/TLS Interception Proxies and Transitive Trust. In *Black Hat* (2012).

[35] JEONG, E., WOOD, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[36] KIM, S., SHIN, Y., HA, J., KIM, T., AND HAN, D. A First Step Towards Leveraging Commodity Trusted Execution Environments for Network Applications. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)* (2015).

[37] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)* (2000).

[38] LAN, C., SHERRY, J., POPA, R. A., RATNASAMY, S., AND LIU, Z. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[39] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2017).

[40] LKL: Linux Kernel Library. https://lkl.github.io.

[41] lwIP: A lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip/.

[42] MazuNAT. https://github.com/kohler/click/blob/master/conf/mazu-nat.click.

[43] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C., SHAFI, H., SHANBHOGUE, V., AND SAVAGAONKAR, U. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[44] MELIS, L., ASGHAR, H. J., DE CRISTOFARO, E., AND KAAFAR, M. A. Private Processing of Outsourced Network Functions: Feasibility and Constructions. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security)* (2016).

[45] MICROSOFT. Law Enforcement Requests Report. https://www.microsoft.com/en-us/about/corporate-responsibility/lerr.

[46] Mirage TCP/IP stack. https://github.com/mirage/mirage-tcpip.

[47] MOGHIMI, A., IRAZOQUI, G., AND EISENBARTH, T. Cachezoom: How SGX amplifies the power of cache attacks. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)* (2017).

[48] NAYLOR, D., SCHOMP, K., VARVELLO, M., LEONTIADIS, I., BLACKBURN, J., LÓPEZ, D. R., PAPAGIANNAKI, K., RODRIGUEZ RODRIGUEZ, P., AND STEENKISTE, P. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (2015).

[49] NetBricks. http://netbricks.io.

[50] ORENBACH, M., LIFSHITS, P., MINKIN, M., AND SILBERSTEIN, M. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)* (2017).

[51] Palo Alto Networks. https://www.paloaltonetworks.com/.

[52] PALO ALTO NETWORKS. Virtualization Features. https://goo.gl/ezntv6.

[53] PANDA, A., HAN, S., JANG, K., WALLS, M., RATNASAMY, S., AND SHENKER, S. NetBricks: Taking the V out of NFV. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[54] POULSEN, K. Five IRS employees charged with snooping on tax returns, 2008. https://www.wired.com/2008/05/five-irs-employ/.

[55] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. NFV Whitepaper. https://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[56] PRIVACY RIGHTS CLEARINGHOUSE. Chronology of Data Breaches. http://www.privacyrights.org/data-breach.

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

[58] ROESCH, M. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the Large Installation System Administration Conference (LISA)* (1999).

[59] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)* (2015).

[60] SCHWARZ, M., WEISER, S., GRUSS, D., MAURICE, C., AND MANGARD, S. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2017).

[61] SEONGMIN KIM AND JUHYENG HAN AND JAEHYEONG HA AND TAESOO KIM AND DONGSU HAN. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[62] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (2012).

[63] SHERRY, J., LAN, C., POPA, R. A., AND RATNASAMY, S. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (2015).

[64] SHIH, M.-W., KUMAR, M., KIM, T., AND GAVRILOVSKA, A. S-NFV: Securing NFV States by Using SGX. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security)* (2016).

[65] SHIH, M.-W., LEE, S., KIM, T., AND PEINADO, M. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017).

[66] SHINDE, S., CHUA, Z. L., NARAYANAN, V., AND SAXENA, P. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the ACM Asia Conference on Computer & Communications Security (AsiaCCS)* (2016).

[67] SHINDE, S., TIEN, D. L., TOPLE, S., AND SAXENA, P. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017).

[68] Snort Community Rulesets. https://www.snort.org/downloads.

[69] STRACKX, R., AND PIESSENS, F. Ariadne: A Minimal Approach to State Continuity. In *Proceedings of the USENIX Security Symposium (USENIX Security)* (2016).

[70] THE FAST DATA PROJECT. Vector packet processing. https://www.fd.io/technology.

[71] TRACH, B., KROHMER, A., GREGOR, F., ARNAUTOV, S., BHATOTIA, P., AND FETZER, C. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the ACM Symposium on SDN Research (SOSR)* (2018).

[72] TSAI, C.-C., PORTER, D. E., AND VIJ, M. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2017).

[73] WANG, W., CHEN, G., PAN, X., ZHANG, Y., WANG, X., BINDSCHAEDLER, V., TANG, H., AND GUNTER, C. A. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2017).

[74] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)* (2015).

[75] YAHOO! Transparency Report. https://transparency.yahoo.com/.

[76] YUAN, X., WANG, X., LIN, J., AND WANG, C. Privacy-preserving Deep Packet Inspection in Outsourced Middleboxes. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)* (2016).

[77] ZETTER, K. Ex-Googler allegedly spied on user emails, chats, 2010. https://www.wired.com/2010/09/google-spy/.

[78] ZHANG, D., ASKAROV, A., AND MYERS, A. C. Predictive Mitigation of Timing Channels in Interactive Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).

[79] Zscaler. https://www.zscaler.com/.

Vesper: Measuring Time-to-Interactivity for Web Pages

Ravi Netravali[†*], Vikram Nathan[†*], James Mickens[‡], Hari Balakrishnan[†]
[†]MIT CSAIL, [‡]Harvard University

Abstract

Everyone agrees that web pages should load more quickly. However, a good definition for "page load time" is elusive. We argue that, for pages that care about user interaction, load times should be defined with respect to *interactivity*: a page is "loaded" when above-the-fold content is visible, and the associated JavaScript event handling state is functional. We define a new load time metric, called *Ready Index*, which explicitly captures our proposed notion of load time. Defining the metric is straightforward, but actually measuring it is not, since web developers do not explicitly annotate the JavaScript state and the DOM elements which support interactivity. To solve this problem, we introduce Vesper, a tool that rewrites a page's JavaScript and HTML to automatically discover the page's interactive state. Armed with Vesper, we compare Ready Index to prior load time metrics like Speed Index; across a variety of network conditions, prior metrics underestimate or overestimate the true load time for a page by 24%–64%. We introduce a tool that optimizes a page for Ready Index, decreasing the median time to page interactivity by 29%–32%.

1 INTRODUCTION

Users want web pages to load quickly [31, 40, 42]. Thus, a vast array of techniques have been invented to decrease load times. For example, browser caches try to satisfy network requests using local storage. CDNs [9, 27, 36] push servers near clients, so that cache misses can be handled with minimal network latency. Cloud browsers [4, 29, 34, 38] resolve a page's dependency graph on a proxy that has low-latency links to web servers; this allows a client to download all objects in a page using a single HTTP round-trip to the proxy.

All of these approaches try to reduce page load time. However, an inconvenient truth remains: none of these techniques directly optimize the speed with which a page becomes *interactive*. Modern web pages have sophisticated, dynamic GUIs that contain both visual and programmatic aspects. For example, many sites provide a search feature via a text input with autocompletion support. From a user's perspective, such a text input is worthless if the associated HTML tags have not been rendered; however, the text input is also crippled if the JavaScript code that implements autocompletion has not been fetched and evaluated. JavaScript code can also implement animations or other visual effects that do not receive GUI inputs directly, but which are still necessary

[*] These authors contributed equally to this work.

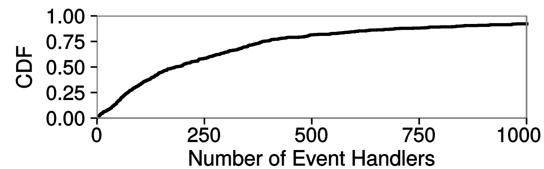


Figure 1: For the Alexa US Top 500 sites, we observed the median number of GUI event handlers to be 182.

for a page to be ready for user interaction. As shown in Figure 1, pages often contain *hundreds* of event handlers that drive interactivity.

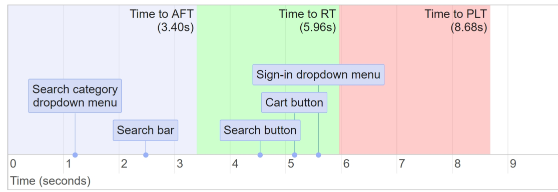
In this paper, we propose a new definition for load time that directly captures page interactivity. We define a page to be fully loaded when:

(1) the visual content in the initial browser viewport[1] has completely rendered, and
(2) for each interactive element in the initial viewport, the browser has fetched and evaluated the JavaScript and DOM state that supports the element's interactive functionality.

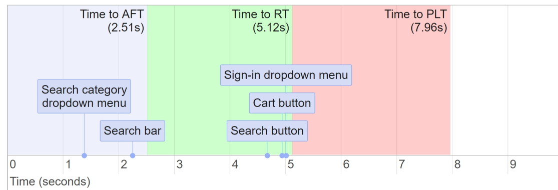
Prior definitions for page load time overdetermine or underdetermine one or both of those conditions (§2), leading to inaccurate measurements of page interactivity. For example, the traditional definition of page load time, as represented by the JavaScript `onload` event, captures when *all* of a page's HTML, JavaScript, CSS, and images have been fetched and evaluated; however, this definition is overly conservative, since only a subset of that state may be needed to allow a user to interact with the content in the initial viewport. Newer metrics like above-the-fold time [21] and Speed Index [14] measure the time that a page needs to render the initial viewport. However, these metrics do not capture whether the page has loaded critical JavaScript state (e.g., event handlers that respond to GUI interactions, or timers that implement animations).

To accurately measure page interactivity, we must determine when conditions (1) and (2) are satisfied. Determining when condition (1) has been satisfied is relatively straightforward, since rendering progress can be measured using screenshots or the paint events that are emitted by the browser's debugger interface. However, determining when condition (2) has been satisfied is challenging. How does one precisely enumerate the JavaScript state that supports interactivity? How does one determine when this state is ready? To answer these questions, we introduce a new measurement framework called *Vesper*.

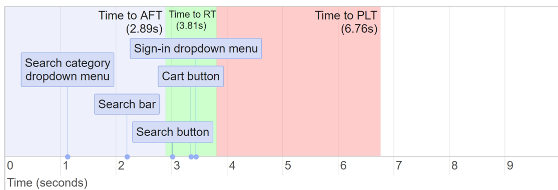
[1]The viewport is the region of a page that the browser is currently displaying. Content in the initial viewport is often called "above-the-fold" content.



(a) Loading the normal version of the page.



(b) Loading a version that optimizes for above-the-fold time.



(c) Loading a version that optimizes for Ready Time.

Figure 2: Timelines for loading amazon.com, indicating when critical interactive components become fully interactive. Note that Ready Time best captures when the site is interactive; furthermore, optimizing for Ready Time is the best way to decrease the page's time-to-interactivity. The client used a 12 Mbit/s link with a 100 ms RTT (§5.1).

RTT	PLT	RT	AFT
25 ms	1.5 (3.9)	1.1 (2.9)	0.8 (1.9)
50 ms	3.4 (7.2)	2.5 (5.8)	1.9 (4.7)
100 ms	6.1 (12.5)	3.9 (9.1)	2.9 (7.0)
200 ms	9.2 (20.6)	5.6 (12.8)	3.8 (8.9)

Figure 3: Median (95th percentile) load time estimates in units of seconds. Each page in our 350 site corpus was loaded over a 12 Mbit/s link.

Vesper rewrites a page's JavaScript and HTML; when the rewritten page loads, the page automatically logs paint events as well as reads and writes to individual JavaScript variables and DOM elements.[2] By analyzing these logs, Vesper generates a progressive load metric, called *Ready Index*, which quantifies the fraction of the initial viewport that is interactive (i.e., visible and functional) at a given moment. Vesper also outputs a derived metric, called Ready Time, which represents the exact time at which *all* of the above-the-fold state is interactive.

Using a test corpus of 350 popular sites, we compared our new load metrics to traditional ones. Figure 2(a) provides a concrete example of the results, showing the differences between page load time (PLT), above-the-fold time (AFT), and Ready Time (RT) for the amazon.com homepage when loaded over a 12 Mbit/s link with a 100 ms RTT. AFT underestimates time-to-full-interactivity by 2.56 seconds; PLT overestimates the time-to-full-interactivity by 2.72 seconds. Web developers celebrate the elimination of *milliseconds* of "load time," claiming that a slight decrease can result in millions of dollars of extra income for a large site [6, 8, 41]. However, our results suggest that developers may be optimizing for the wrong definition of load time. As shown in Figure 3, prior metrics inaccurately forecast time-to-full-interactivity under a variety of network conditions, with median inaccuracies of 24%–39%; as shown in our user study (§6), users with interactive goals prefer websites that actually prioritize the loading of interactive content.

The differences between load metrics are particularly stark if a page's dependency graph [25, 37] is deep, or if a page's clients are stuck behind high-latency links. In these scenarios, the *incremental interactivity* of a slowly-loading page is important: as the page trickles down the wire, interactive HTML tags should become visible and functional as soon as possible. This allows users to meaningfully engage with the site, even if some content is missing; incremental interactivity also minimizes the time window for race conditions in which user inputs are generated at the same time that JavaScript event handling state is being loaded [30]. To enable developers to build incrementally-interactive pages with low Ready Indices, we extended Polaris [25], a JavaScript framework that allows a page to explicitly schedule the order in which objects are fetched and evaluated. We created a new Polaris scheduler that optimizes for Ready Index; the resulting scheduler improves RI by a median of 29%, and RT by a median of 32%. Figure 2(c) demonstrates the scheduler's performance on the amazon.com homepage. Importantly, Figure 2(b) shows that optimizing for above-the-fold time does *not* optimize for time-to-interactivity.

Of course, not all sites have interactive content, and even interactive sites can be loaded by users who only look at the content. In these situations, pages should optimize for the rendering speed of above-the-fold content. Fortunately, our user study shows that pages which optimize for Ready Index will substantially reduce user-perceived rendering delays too (§6). If desired, Vesper enables developers to automatically optimize their pages solely for rendering speed instead of Ready Index.

In summary, this paper has four contributions. First, we define a new load metric called Ready Index which quantifies a page's interactive status (§3). Determining how interactivity evolves over time is challenging. Thus, our second contribution is a tool called Vesper that automates the measurement of Ready Index (§4). Our third contribution is a study of Ready Index in 350 real pages.

[2] Each HTML tag in a web page has a corresponding DOM element. The DOM element is a special JavaScript object that JavaScript code can use to manipulate the properties of the underlying HTML tag.

By loading those pages in a variety of network conditions, we explain the page characteristics that lead to faster interactivity times (§5). Our fourth contribution is an automated framework for optimizing a page's Ready Index or pure rendering speed; both optimizations are enabled by Vesper-collected data. User studies demonstrate that pages which optimize for Ready Index provide better support for immediate interactivity (§6).

2 Background

In this section, we describe prior attempts to define "page load time." Each metric tracks a different set of page behaviors; thus, for a given page load, different metrics may provide radically different estimates of the load time.

The Original Definition: The oldest metric is defined with respect to the JavaScript `onload` event. A browser fires that event when all of the external content in a page's static HTML file has been fetched and evaluated. All image data must be present and rendered; all JavaScript must be parsed and executed; all style files must be processed and applied to the relevant HTML tags; and so on. The load time for a page is defined as the elapsed time between the `navigationStart` event and the `onload` event. In the rest of the paper, we refer to this load metric as PLT ("page load time").

PLT was a useful metric in the early days of the web, but modern web pages often dynamically fetch content after the `onload` event has fired [12, 13]. PLT also penalizes web pages that have large amounts of statically-declared below-the-fold content. Below-the-fold content resides beneath the initial browser viewport, and can only be revealed by user scrolling. PLT requires static below-the-fold content to be fetched and evaluated before a page load is considered done. However, from a user's perspective, a page can be ready even if its below-the-fold content is initially missing: the interactivity of the initial viewport content is the primary desideratum.

Time to First Paint: Time to First Paint (TTFP) measures when the browser has received enough page data to render the first pixels in the viewport. Time to First Meaningful Paint [33], or TTFMP, measures the time until the biggest layout change, using the intuition that the associated paint event is the one that matters most. TTFP and TTFMP try to capture the earliest time that a human could usefully interact with a page. For a given PLT, a lower TTFP or TTFMP is better. However, decreasing a page's PLT is not guaranteed to lower the other metrics, and vice versa [1]. For example, when the HTML parser (which generates input for the rendering pipeline) hits a `<script>` tag, the parser may need to synchronously fetch and evaluate the JavaScript file before continuing the HTML parse [25]. By pushing `<script>` tags to the end of a page's HTML, render times may improve; however, careless deferral of JavaScript evaluation may hurt interactivity, since event handlers will be registered later, animation callbacks will start firing later, and so on.

Above-the-fold Time: This metric represents the time that the browser needs to render the final state of *all* pixels in the initial browser viewport. Like TTFP, above-the-fold time (AFT) is not guaranteed to move in lockstep with PLT. Measuring AFT and TTFP requires a mechanism for tracking on-screen events. WebKit-derived browsers like Chrome and Opera expose paint events via their debugging interfaces. Rendering progress can also be tracked using screenshots [16, 19].

If a web page contains animations, or videos that automatically start playing, a naïve measurement of AFT would conclude that the page never fully loaded. Thus, AFT algorithms must distinguish between *static pixels* that are expected to change a few times at most, and *dynamic pixels* that are expected to change frequently, even once the page has fully loaded. To differentiate between static and dynamic pixels, AFT algorithms use a threshold number of pixel updates; a pixel which is updated more often than the threshold is considered to be dynamic. AFT is defined as the time that elapses until the last change to a static pixel.

Speed Index: AFT fails to capture the progressive nature of the rendering process. Consider two hypothetical pages which have the same AFT, but different rendering behavior: the first page updates the screen incrementally, while the second page displays nothing until the very end of the page load. Most users will prefer the first page, even though both pages have the same AFT.

Speed Index [14] captures this preference by explicitly logging the progressive nature of page rendering. Intuitively speaking, Speed Index tracks the fraction of a page which has not been rendered at any given time. By integrating that function over time, Speed Index can penalize sites that leave large portions of the screen unrendered for long periods of time. More formally, a page's Speed Index is $\int_0^{end} 1 - \frac{p(t)}{100}\,dt$, where *end* is the AFT time, and $p(t)$ is the percentage of static pixels at time t that are set to their final value. A lower Speed Index is better than a higher one.

Strictly speaking, a page's Speed Index has units of "percentage-of-visual-content-that-is-not-displayed milliseconds." For brevity, we abuse nomenclature and report Speed Index results in units of just "milliseconds." However, a Speed Index cannot be directly compared to a metric like AFT that is actually measured in units of time. Also note that TTFP, AFT, and Speed Index do not consider the load status of JavaScript state. As a result, these metrics cannot determine (for example) when a button that has been rendered has actually gone live as result of the associated event handlers being registered.

User-perceived PLT: This metric captures when a user believes that a page render has finished [20, 35]. Unlike Speed Index, User-perceived PLT is not defined programmatically; instead, it is defined via user studies which empirically observe when humans think that enough of a page has rendered for the page load to be "finished." Like Speed Index, User-perceived PLT ignores page functionality (and thus page interactivity). User-perceived PLT also cannot be automatically measured, which prevents developers from easily optimizing for the metric.

TTI: Several commercial products claim to measure a page's time-to-interactivity (TTI) [28, 32]; however, these products do not explicitly state how interactivity is defined or measured. In contrast, Google is currently working on an open standard for defining TTI [15]. The standard's definition of TTI is still in flux. The current definition expresses interactivity in terms of time-to-first-meaningful-paint, the number of in-flight network requests, and the utilization of the browser's main thread (which is used to dispatch GUI events, execute JavaScript event handlers, and render content). TTI defines an "interactive window" as a period in which the main thread runs no tasks that require more than 50 ms; in other words, during an interactive window, the browser can respond to user input in at most 50 ms. A page's TTI is the maximum of:

(1) the time when the `DOMContentLoaded` event has fired, and
(2) the start time of the first interactive window that has at most two network requests in flight for 5 consecutive seconds.

This definition for load time has several problems. First, it could declare a page to be loaded even if the page has not rendered all of the content in the initial viewport. Second, condition (2) does not consider whether a network request is for above-the-fold, interactive content; a window with many outstanding network requests may represent an interactive page if those network requests are for below-the-fold state. Similarly, this TTI definition makes no explicit reference to the JavaScript state that supports above-the-fold event handlers, and the JavaScript state that does not. User-perceived interactivity requires the former state to be loaded, but not the latter.

Summary: Traditional metrics for load time fail to capture important aspects of user-perceived page readiness. PLT does not explicitly track rendering behavior, and implicitly assumes that all JavaScript state is necessary to make above-the-fold content usable. AFT, Speed Index, User-perceived PLT, and TTFP/TTFMP consider visual content, but are largely oblivious to the status of JavaScript code—the code is important only to the extent that it might update a pixel using DOM methods [23]. However, AFT, Speed Index, User-perceived PLT, and TTFP/TTFMP completely ignore event handlers (and the program state that event handlers manipulate). Consequently, these metrics fail to capture the interactive component of page usability. Google's TTI also imprecisely captures above-the-fold, interactive state.

3 Ready Index

In this section, we formally define Ready Index (RI). Like Speed Index, RI is a progressive metric that captures incremental rendering updates. Unlike Speed Index, RI also captures the progressive loading of JavaScript state that supports interactivity.

Defining Functionality: Let T be an upper-bound on the time that a browser needs to load a page's above-the-fold state, and make that state interactive. This upper-bound does not need to be tight; in practice (§5), we use a static value of 30 seconds.

Let E be the set of DOM elements that are visible in the viewport at T. For each $e \in E$, let $h(e)$ be the set of all event handlers that are attached to e at or before T. Let t_e be the earliest time at which, for all handlers $h \in h(e)$, h's JavaScript function has been declared, and all JavaScript state and DOM state that would be accessed by h's execution has been loaded. Given those definitions, we express the *functionality progress* of e as

$$F(e,t) = \begin{cases} 0 & t < t_e \\ 1 & t \geq t_e \end{cases} \quad (1)$$

Intuitively speaking, Equation 1 states that a DOM node is not functional until all of the necessary event handlers have been attached to the node, and the browser has loaded all of the state that the handlers would touch if executed.

Defining Visibility: An element e may be the target of multiple paint events, e.g., as the browser parses additional HTML and recalculates e's position in the layout. We assume that e is not fully visible until its last paint completes. Let $P(e)$ be the set of all paint events that update e, and let $P_t(e) \subseteq P(e)$ be the paint events that have occurred by time t. The *visibility progress* of e is

$$V(e,t) = \frac{|P_t(e)|}{|P(e)|} \quad (2)$$

Similar to how Speed Index computes progressive rendering scores for pixels [14], Equation 2 assumes that each paint of e contributes equally to e's visibility score. Note that $0 \leq V(e,t) \leq 1$.

Defining Readiness: Given the preceding definitions for functionality and visibility, we define the readiness

of an element e as

$$R(e,t) = \frac{1}{2}F(e,t) + \frac{1}{2}V(e,t) \quad (3)$$

such that the functionality and visibility of e are equally weighed,[3] and $0 \leq R(e,t) \leq 1$. The readiness of the entire page is then defined as

$$R(t) = \sum_{e \in E} A(e)R(e,t) \quad (4)$$

where $A(e)$ is the area (in pixels) that e has at time T.

Putting It All Together: An element e is *fully ready* at time t if $R(e,t) = 1$, i.e., if e is both fully visible and fully functional. A page's Ready Time (RT) is thus the smallest time at which all of the above-the-fold elements are ready. A page's Ready Index (RI) is the area above the curve of the readiness progress function. Thus, RI is equal to

$$RI = \int_0^T 1 - \frac{R(t)}{R(T)}\, dt \quad (5)$$

4 VESPER

Vesper is a tool that allows a web developer to determine the RI and RT for a specific page. Vesper must satisfy three design goals. First, Vesper must produce *high coverage*, i.e., Vesper must identify all of a page's interactive, above-the-fold state. Second, Vesper's instrumentation must have *minimal overhead*, such that instrumented pages have RI and RT scores that are close to those of unmodified pages. Ideally, Vesper would also be *browser-agnostic*, i.e., capable of measuring a page's RI and RT without requiring changes to the underlying browser.

These design goals are in tension. To make Vesper browser-agnostic, Vesper should be implemented by rewriting a page's JavaScript code and HTML files, not through modification of a browser's JavaScript engine and rendering pipeline; unfortunately, the most direct way to track interactive state is via heavyweight instrumentation of all reads and writes that a page makes to the JavaScript heap, the DOM, and the rendering bitmap. Vesper resolves the design tension by splitting instrumentation and log analysis across two separate page loads. Each load uses a differently-rewritten version of a page, with the first version using heavyweight instrumentation, and the second version using lightweight instrumentation. As a result, the second page load injects minimal timing distortion into the page's true RI and RT scores. Figure 4 provides an overview of Vesper's two-phase workflow. We provide more details in the remainder of this section.

[3] The use of equal weights reflects our assumption that functionality and visibility are equally important. However, future empirical research may suggest better weighting schemes.

4.1 Phase 1

The goal of this phase is to identify the subset of DOM nodes and JavaScript state that support above-the-fold interactivity.

Element Visibility: For most pages, only a subset of all DOM nodes will have bounding boxes that overlap with the initial viewport. Even if a node is above-the-fold, it may not be visible, e.g., due to CSS styling which hides the node. Vesper injects a JavaScript timer into the page which runs at time T. When the timer function executes, it traverses the DOM tree and records which nodes are visible. In the rest of the section, we refer to this timer as the Vesper timer.

Event Handlers: Developers make a DOM element interactive by attaching one or more event handlers to that element. For example, a `<button>` element does nothing in response to clicks until JavaScript code registers `onclick` handlers for the element. To detect when such handlers are added, Vesper shims the event registration interfaces [22]. There are two types of registration mechanisms:

- DOM elements define JavaScript-accessible properties and methods that support event handler registration. For example, assigning a function `f` to a property like `DOMnode.onclick` will make `f` an event handler for clicks on that DOM node. Invoking `DOMnode.addEventListener("click", f)` has similar semantics. Vesper interposes on registration mechanisms by injecting new JavaScript into a page that modifies the DOM prototypes [22]; the modified prototypes insert logging code into the registration interfaces, such that each registered handler is added to a Vesper-maintained, in-memory list of the page's handlers.
- Event handlers can also be defined via HTML, e.g., `<img src=... onload=handler()/>`. At T, the Vesper timer iterates through the page's DOM tree, identifying event handlers that were not registered via a JavaScript-level interface, and adding those handlers to Vesper's list.

The Vesper timer only adds a handler if the handler is attached to a visible DOM element that resides within the initial viewport.

Event Handler State: When a handler fires, it issues reads and writes to program state. That state may belong to JavaScript variables, or to DOM state like the contents of a `<b>` tag. As the handler executes, it may invoke other functions, each of which may touch an additional set of state. The aggregate set of state that the call chain may touch is the *functional state* for the handler. Given a DOM element e, we define e's functional state as the

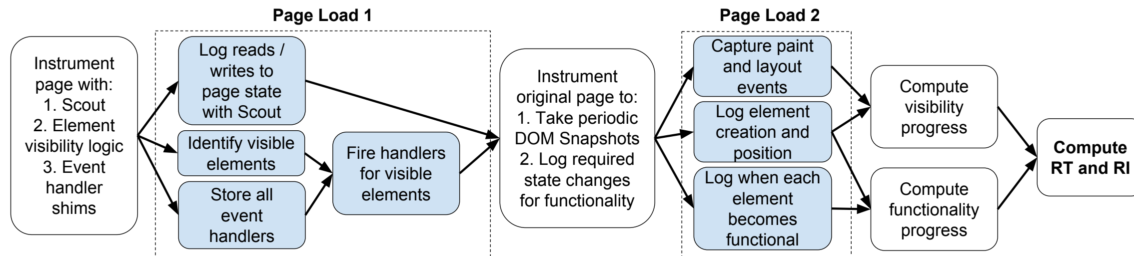


Figure 4: Vesper's two-phase approach for measuring RI and RT. Shaded boxes indicate steps that occur during a page load. Clear boxes represent pre- and post-processing steps.

union of the functional state that belongs to each of *e*'s event handlers.

If *e* resides within the initial viewport, then *e* is not functional until two conditions have been satisfied:

1. all of *e*'s event handlers must be registered, and
2. all of *e*'s functional state must be loaded.

At any given moment during the page load, none, either, or both of these conditions may be satisfied. For example, if *e*'s event handlers are defined in a `<script>` tag, but key functional state is defined by downstream HTML or `<script>` tags, then after evaluation of the first `<script>` tag, condition (1) is true, but condition (2) is not.

To identify a page's functional state, Vesper instruments the HTML and JavaScript in a page, such that, when the instrumented page loads, the page will log all reads and writes to JavaScript variables and DOM state. When the Vesper timer runs, it actively invokes the event handlers that were captured by event registration shimming. As those handlers fire, their call chains touch functional state. By post-processing the page's logs, and looking for reads and writes that occurred after the Vesper timer began execution, Vesper can identify a page's functional state. In particular, Vesper can associate each handler with its functional state, and each DOM element with the union of the functional states of its handlers.

To fire the handlers for a specific event type like `click`, the Vesper timer determines the minimally-sized DOM subtree that contains all handlers for the `click` event. Vesper then constructs a synthetic `click` event, and invokes the built-in `DOMnode.dispatchEvent()` method for each leaf of the subtree. This approach ensures that synthetic events follow the same dispatch path used by real events.

Some event types are logically related to a single, high-level user interaction. For example, when a user clicks a mouse button, her browser generates `mousedown`, `click`, and `mouseup` events, in that order. Vesper is aware of these semantic relationships, and uses them to guide the generation of synthetic events, ensuring a realistic sequence of handler firings.

Implementation: To instrument a page, Vesper could modify the browser's renderer and JavaScript engine to track reads and writes to DOM objects and JavaScript variables. However, our Vesper prototype leverages Scout [25] instead. Scout is a browser-agnostic rewriting framework that instruments a page's JavaScript and HTML to log reads and writes. A browser-agnostic approach is useful because it allows Vesper to compare a page's Ready Index across different browser types (§5.4).

The instrumentation that tracks element visibility and handler registration adds negligible overhead to the page load process. However, tracking all reads and writes to page state is more costly. Across the 350 pages in our test corpus, we measured a Scout-induced load time increase of 4.5% at the median, and 7.6% at the 95th percentile. Thus, trying to calculate RI and RT directly in Phase 1 would lead to inflated estimates. To avoid this problem, we use the outputs of Phase 1 as the inputs to a second phase of instrumentation. This second phase is more lightweight, and directly calculates RI and RT.

4.2 Phase 2

In Phase 1, Vesper discovers the DOM nodes and JavaScript variables that support above-the-fold interactivity. In Phase 2, Vesper tracks the rendering progress of the above-the-fold DOM elements that were identified in Phase 1. Vesper also tracks the rate at which functional JavaScript state is created. This information is sufficient to derive RI and RT.

4.2.1 Measuring Functionality Progress

A DOM element becomes functional when all of its event handlers have been registered, and all of the functional state for those handlers has been created. An element's functional state may span both the JavaScript heap and the DOM. Vesper uses different techniques to detect when the two types of state become ready.

JavaScript state: By analyzing Scout logs from Phase 1, Vesper can determine when the last write to each JavaScript variable occurs. The "last write" is defined as a source code line and an execution count for that line. The execution count represents the fact that a source code

line can be run multiple times, e.g., if it resides within a loop body.

At the beginning of Phase 2, Vesper rewrites a page's original JavaScript code, injecting a logging statement after each source code line that generates a final write to functional JavaScript state. The logging statement updates the execution count for the line, and only outputs a log entry if the final write has been generated.

DOM state: An event handler's functional state may also contain DOM nodes. For example, a `keypress` handler may assume the existence of a specific DOM node whose properties will be modified by the handler. At the beginning of Phase 2, Vesper rewrites a page's original HTML to output the creation time for each DOM node. The rewriting is complicated by the fact that, when a browser parses HTML, it does not trigger a synchronous, JavaScript-visible event upon the creation of a DOM node. Thus, Vesper rewrites a page's HTML to include a new `<script>` tag after *every* original HTML tag. The new `<script>` tag logs two things: the creation of the preceding DOM node, and the bounding boxes of all DOM nodes which exist at that moment in the HTML parse. The `<script>` tag then removes itself from the DOM tree (so that at any point in the HTML parse, non-Vesper code that inspects the DOM tree will see the original DOM tree which does not contain Vesper's self-destructing tags). *DOM snapshots* using self-destructing JavaScript tags are by far the most expensive part of the Phase 2 instrumentation; however, they only increase page load times by 1.9% at the median, and 3.9% at the 95th percentile. Thus, we believe that the overhead is acceptable.

After the initial HTML parse, DOM nodes may be created by asynchronous event handlers. Vesper logs such creations by interposing on DOM methods like `DOMnode.appendChild()`. This interpositioning has negligible overhead and ensures that Vesper has DOM snapshots after the initial HTML parse.

4.2.2 Measuring Visibility Progress

DOM snapshots allow Vesper to detect when elements are created. However, a newly-created element will not become *visible* until some point in the future, because the construction of the DOM tree is earlier in the rendering pipeline than the paint engine. Browsers do not expose layout or paint events to JavaScript code. Fortunately, Vesper can extract those events from the browser's debugging output [11]. Each layout or paint message contains the bounding box and timestamp for the activity. Unfortunately, the message does not identify which DOM nodes were affected by the paint; thus, Vesper must derive the identities of those nodes.

After the Phase 2 page load is complete, Vesper collates the DOM snapshots and the layout+paint debugging events, using the following algorithm to determine the layout and paint events that rendered a specific DOM element e:

1. Vesper finds the first DOM snapshot that contains a bounding box for e. Let that snapshot have a timestamp of t_d. Vesper searches for the layout event that immediately precedes t_d and has a bounding box that contains e's bounding box. Vesper defines that layout event L_{first} to be the one which added e to the layout tree.
2. Vesper then rolls forward through the log of paint and layout events, starting at L_{first}, and tracking all paint events to e's bounding box. That bounding box may change during the page load process, but any changes will be captured in the page's DOM snapshots. Thus, Vesper can determine the appropriate bounding box for e at any given time.

As described in Equation 2, each paint event contributes equally towards e's visibility score. For example, if e is updated by four different paints, then e is 25% visible after the first one, 50% visible after the second one, and so on.

In summary, the output of the Phase 2 page load is a trace of a page's functionality progress and visibility progress. Using that trace, and Equations 4 and 5, Vesper determines the page's RT and RI. Note that, for a given version of a page (i.e., for a particular set of HTML, CSS, and JavaScript files), Phase 1 only needs to run once, on the server-side, with Phase 2 running during the live page loads on clients in the wild.

4.3 Discussion

The PLT metric is natively supported by commodity browsers, meaning that a page can measure its own PLT simply by registering a handler for the `onload` event. Newer metrics that lack native browser support require 1) browsers to install a special plugin (the SI approach [10]), or 2) page developers to rewrite content (the approach used by our Vesper prototype). Vesper is amenable to implementation via plugins or native support; either option would enable lower instrumentation overhead, possibly allowing Vesper to collapse its two phases into one.

As a practical concern, a rewriting-based implementation of Vesper must deal with the fact that a single page often links to objects from multiple origins. For example, a developer for `foo.com` will lack control over the bytes in linked objects from `bar.com`. As described in Section 5, our Vesper prototype uses Mahimahi [26], a web replay tool, to record *all* of the content in a page; Vesper rewrites the recorded content, and then replays the modified content to a browser that runs on a machine controlled by the `foo.com` developer. In this manner, as with the browser plugin approach, a developer can mea-

sure RI and RT for any page, regardless of whether the developer owns all, some, or none of the page content.

All load metrics are sensitive to nondeterministic page behavior. In the context of Vesper, such behavior may result in a page having different interactive state across different page loads. For example, an event handler that branches on the return value of `Math.random()` might access five different DOM nodes across five different loads of the page. Even if a page's state is deterministic, Vesper's synthetic event generation (§4.1) is not guaranteed to exhaustively explore all *possible* event handler interleavings—instead, Vesper tests the *most likely* event sequences based on how a realistic human user would generate GUI events. Vesper could use symbolic execution [7] to increase path coverage, but we believe that Vesper's current level of coverage is sufficiently high for two reasons. First, from the empirical perspective, the pages in our large test corpus do not exhibit nondeterminism that results in different functional state across different loads. Second, the Vesper timer does not fire synthetic events until a page is fully loaded; thus, "unexpected" event-level race conditions arising from partially-loaded content [30] should not arise.

5 Evaluation

In this section, we compare RI and RT to three prior metrics for page load time (PLT, AFT, and Speed Index). We do not evaluate Google's TTI because the metric's definition is still evolving.

Across a variety of network conditions, we find that PLT overestimates the time that a page requires to become interactive; in contrast, AFT and Speed Index underestimate the time-to-interactivity (§5.2 and A.1.1). These biases persist when browser caches are warm (§A.1.2). Furthermore, the discrepancies between prior metrics and our interactive metrics are large, with median and 95th percentile load time estimates often differing by multiple *seconds* (Figures 3 and 6). Thus, Ready Index and Ready Time provide a fundamentally new way of understanding how pages load.

5.1 Methodology

We evaluated the various load metrics using a test corpus of 350 pages. The pages were selected from the Alexa US Top 500 list [2]. We filtered out sites using deprecated JavaScript statements that Scout [25] does not rewrite. We also filtered sites that caused errors with Speedline [19], a preexisting tool for capturing SI.

To measure PLT, we recorded the time between the JavaScript `navigationStart` and `onload` events (§2). RT and RI were measured with Vesper; we set T to 30 seconds. We also used Vesper to measure AFT and SI.[4] Calibration experiments showed that Vesper's estimates of SI were within 2.1% of Speedline's estimates at the median, and within 3.9% at the 95th percentile.

Measuring PLT is non-invasive, since unmodified pages will naturally fire the `navigationStart` and `onload` events. Capturing the other metrics requires new instrumentation, like DOM snapshots (§4.2.1). To avoid measurement biases due to varying instrumentation overheads, each experimental trial loaded each page five times, and in each of the five loads, we enabled all of Vesper's Phase 2 instrumentation, such that each load metric could be calculated. Enabling all of the instrumentation increased PLT by 1.9% at the median, and 3.9% at the 95th percentile.

We used Mahimahi [26] to record the content in each test page, and later replay the content via emulated network links. With the exception of the mobile experiments (§A.1.1), all experiments were performed on Amazon EC2 instances running Ubuntu 14.04. Unless otherwise specified, each page load used Google Chrome (v53) with a cold browser cache and remote debugging enabled so that we could track layout and paint events.

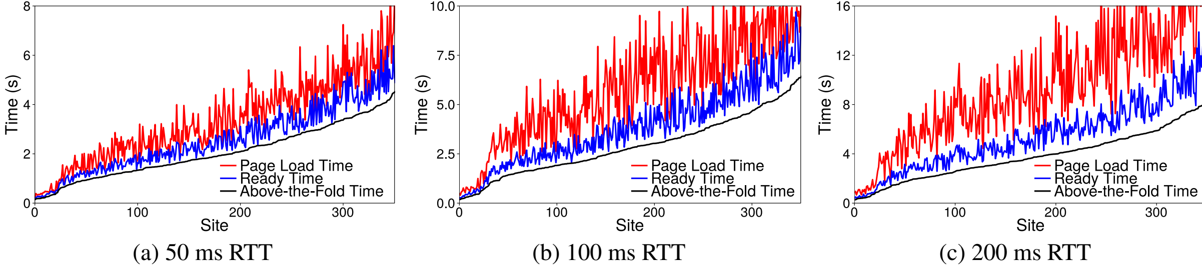
5.2 Cross-metric Comparisons

On computationally-powerful devices like desktops and laptops, network latency (not bandwidth) is the primary determinant of how quickly a page loads [1, 5, 25, 34]. So, our first set of tests used a t2.large EC2 VM with a fixed bandwidth of 12 Mbit/s, but a minimum round-trip latency that was drawn from the set {25 ms, 50 ms, 100 ms, 200 ms}. These emulated network conditions were enforced by the Mahimahi web replay tool.

Figure 3 summarizes the results for PLT, RT, and AFT. Recall that these metrics are non-progressive, i.e., they express a page's load time as a single number that represents when the browser has "completely" loaded the page (for some definition of "completely"). As expected, PLT is higher than RT because PLT requires all page state, including below-the-fold state, to be loaded before a page load is finished. Also as expected, AFT is lower than RT, because AFT ignores the load status of JavaScript code that is necessary to make visible elements functional.

The surprising aspect of the results is that the differences between the metrics are so noticeable. As shown in Figures 2(a) and 3, the differences are large in terms of percentage (24.0%–64.3%); more importantly, the differences are large in terms of absolute magnitude, equating to hundreds or thousands of milliseconds. For example, with a round-trip latency of 50 ms, RT and PLT differ by roughly 900 ms at the median, and by 1.4 seconds at the 95th percentile. For the same round-trip latency, RT and

[4]To compute SI, Vesper only considers element visibility, assigning zero weight to functionality.



(a) 50 ms RTT (b) 100 ms RTT (c) 200 ms RTT

Figure 5: Comparing RT, PLT, and AFT. Results used emulated links with a bandwidth of 12 Mbit/s.

RTT	Ready Index	Speed Index
25 ms	714 (1522)	568 (1027)
50 ms	1759 (3846)	1325 (3183)
100 ms	2737 (6174)	2054 (4549)
200 ms	4252 (9719)	3071 (6913)

Figure 6: Median (95th percentile) load time estimates (see Section 2 for a discussion of the units). Results used our entire 350 page corpus. Content was loaded over a 12 Mbit/s link.

AFT differ by approximately 600 ms at the median, and by 1.1 seconds at the 95th percentile.

The discrepancies increase as RTTs increase. This observation is important, because cellular and residential networks often have RTTs that exceed 100 ms [3, 18]. For example, in our emulated network with an RTT of 100 ms, RT differed from PLT by *2.2 seconds* at the median; RT differed from AFT by 1 second at the median. From the perspective of a web developer, the differences between RT and AFT are particularly important. Users frequently assume that a visible element is also functional. However, visibility does not necessarily imply functionality, and interactions with partially-functional elements can lead to race conditions and broken page behavior [30]. In Section 6, we describe how developers can create incrementally-interactive pages that minimize the window in which a visual element is not interactive.

Figure 5 compares the RT, PLT, and AFT values for each page in our 350 site corpus. Pages are sorted along the x-axis in ascending AFT order. Figure 5 vividly demonstrates that PLT is an overly conservative definition for user-perceived notions of page readiness. The spikiness of the RT line also demonstrates that pages with similar AFT values often have very different RT scores. For example, consider an emulated link with a 100 ms round-trip time. Sites 200 (`mashable.com`) and 201 (`overdrive.com`) have AFT values of 3099 ms and 3129 ms, respectively. However, the sites have RT values of 4418 ms and 3970 ms, a difference of over 400 ms. In Section 5.3, we explain how the relationships between a page's HTML, CSS, and JavaScript cause divergences in RT and AFT.

Figures 6 and 7 compare the two progressive metrics. The results mirror those for the non-progressive metrics. A page's SI is lower than its RI, because SI does not consider the load status of JavaScript code that supports interactivity. Furthermore, pages with similar SIs often have much different RIs.

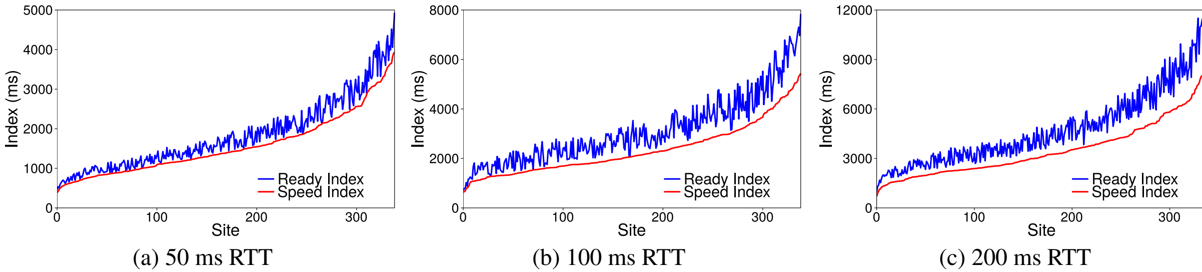
5.3 Case Studies

Figure 8 uses two randomly-selected pages to demonstrate how interactivity evolves. Figure 8(a) describes the homepage for Bank of America, whereas Figure 8(b) describes the homepage for WebMD. Using the terminology from Section 3, each graph plots the visual progression of the page ($\sum_{e \in E} V(e,t)A(e)$) and the readiness progression of the page ($R(t)$); in the graphs, each data point is normalized to the range [0.0,1.0]. At any given moment, a page's readiness progression is less than or equal to its visual progression, since visual progression does not consider the status of functional state.

The gaps between the red and blue curves indicate the existence of visible, interactive DOM elements that are not yet functional. If users try to interact with such elements, then at best, nothing will happen; at worst, an incomplete set of event handlers will interact with incomplete JavaScript and DOM state, leading to erroneous page behavior. For example, the Bank of America site contains a text input that supports autocompletion. With RTTs of 100 ms and above, we encountered scenarios in which the input was visible but not functional. In these situations, we manually verified that a human user could type into the text box, have no autosuggestions appear, and then experience the text disappear and reappear with autosuggestions as the page load completed.

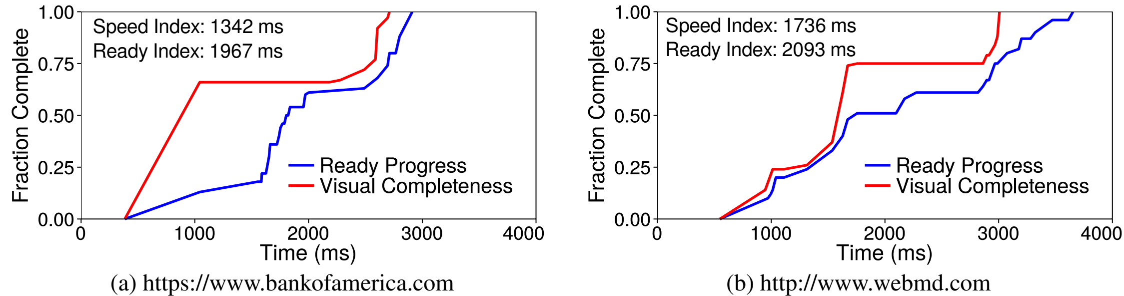
Both the red and blue curves contain stalls, i.e., time periods in which no progress is made. For example, both pages exhibit a lengthy stall in their visual progression—for roughly a second, neither page updates the screen. Both pages also contain stretches that lack visual progress or readiness progress. During these windows, a page is not executing any JavaScript code that creates interactive state.

Functionality progression stalls when the `<script>` tags supporting functionality have not been fetched, or have been fetched but not evaluated. Visual progression may stall for a variety of reasons. For example, the browser might be blocked on network fetches, waiting on



(a) 50 ms RTT (b) 100 ms RTT (c) 200 ms RTT

Figure 7: Comparing the progressive metrics (Ready Index versus Speed Index). Results used emulated links with a bandwidth of 12 Mbit/s.



(a) https://www.bankofamerica.com (b) http://www.webmd.com

Figure 8: Exploring how visibility and functionality evolve for two different pages. The client had a 12 Mbit/s link with an RTT of 100 ms. Remember that a progressive metric like Ready Index is calculated by examining the area that is *above* a curve.

HTML data so that new tags can be parsed and rendered. Browsers also use a single thread for HTML parsing, DOM node rendering, and JavaScript execution; thus, executing a `<script>` tag blocks parsing and rendering of downstream HTML. As described in Section 6, developers can use automated tools to minimize these stalls and improve a page's Ready Time and Ready Index.

5.4 Other Page Load Scenarios

In Section A.1, we analyze how Ready Index evolves in three additional scenarios: mobile page loads, page loads that use a warm browser cache, and page loads on two different browsers (namely, Chrome versus Opera). Due to space restrictions, we merely provide a summary here:

Mobile page loads: Mobile page loads exhibit the same trends that we observed on more powerful client devices. For example, on a Nexus 5 phone running on an emulated Verizon LTE cellular link, the median PLT is 35.2% larger than the median RT; the median RI is 29.7% larger than the median Speed Index.

Warm cache loads: The results from earlier in this section used cold caches. However, clients sometimes have a warm cache for objects in a page to load. As expected, pages load faster (for all metrics) when caches are warm. However, the general trends from Section 5.2 still hold. For example, on a desktop browser with a 12 Mbit/s, 100 ms RTT link, the median warm-cache PLT is 38.2% larger than the median RT. The median RT is 26.0% larger than the median AFT.

Chrome vs. Opera: Since our Vesper implementation is browser-agnostic, it can measure a single page's load metrics across different browser types. For example, we compared RI on Chrome and Opera. With cold browser caches and a 12 Mbit/s, 100 ms RTT link, Chrome's RI values were 6.5% lower at the median, and 11.9% lower at the 95th percentile. Since Vesper's logs contain low-level information about reads and writes to interactive state, browser vendors can use these logs to help optimize the internal browser code that handles page loading.

6 Optimizing for Interactivity

To minimize a page's Ready Time and Ready Index, browsers must fetch and evaluate objects in a way that prioritizes interactivity. In particular, a browser should:

1. maximize utilization of the client's network connection;
2. prioritize the fetching and evaluating of HTML files that define above-the-fold DOM elements;
3. prioritize the fetching and evaluating of `<script>` tags that generate interactive, above-the-fold state; and
4. respect the semantic dependencies between a page's objects.

By maximizing network utilization (Goal (1)), a browser minimizes the number of CPU stalls that occur due to synchronous network fetches; ideally, a browser would

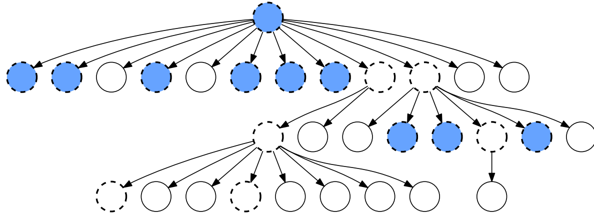


Figure 9: The dependency graph for `priceline.com`. OPT-PLT assigns equal weights to all nodes. OPT-SI prioritizes the shaded objects. OPT-RI prioritizes the objects with dashed outlines.

fetch each piece of content *before* that content is desired by a parsing/evaluation engine. Goals (2) and (3) directly follow from the definitions for page readiness in Section 3. However, Goal (4) is in tension with the others: fetching and evaluating objects in a way that satisfies Goals (1), (2), and (3) may break page functionality. For example, two JavaScript libraries may have shared state, like a variable that is written by the first library and read by the second. Invalid reads and other problems will arise if a browser evaluates the two libraries "out-of-order" with respect to the lexical order of their `<script>` tags in the page's HTML.

Web pages contain a variety of additional dependencies that constrain the order in which objects can be fetched and evaluated. Polaris [25] is a load optimizer that uses Scout to extract all of these dependencies and generate an explicit dependency graph (i.e., a partial ordering that specifies how certain objects must be loaded before others). Polaris then rewrites the page so that the page is *self-assembling*. The rewritten page uses a custom JavaScript library to schedule the fetching and evaluating of objects in a way that satisfies Goals (1) and (4).

At any given moment in a page load, the *dynamic critical path* is the path in the dependency graph that has the largest number of unfetched objects. The default Polaris scheduler prioritizes the fetching of objects along the dynamic critical path. This policy minimizes PLT, but may increase or decrease RT, depending on whether interactive, above-the-fold state is created by objects along the dynamic critical path.

We created a new scheduling policy, called OPT-RI ("optimize RI"), which prioritizes the loading of interactive content. Let $O_{interactive}$ be the objects (e.g., HTML files, JavaScript files) that Vesper identifies as generating interactive, above-the-fold state. Given $O_{interactive}$ and the dependency graph from Scout, OPT-RI assigns node weights of zero to nodes that do not reside in $O_{interactive}$; for a node in $O_{interactive}$, OPT-RI finds all of the above-the-fold elements that the node affects, and then weights the node by the fraction of the initial viewport area that those elements cover. During the actual page load, the OPT-RI scheduler prioritizes objects along the *weighted* dynamic critical path.

We also defined OPT-SI, which only considers visual progress. Nodes that do not lead to the creation of visible, above-the-fold DOM elements receive a weight of zero. For each remaining node, OPT-SI finds the DOM elements that the node influences, and assigns a node weight that is proportional to the fraction of the viewport that the elements cover. OPT-SI will not prioritize JavaScript files that only define event handler state; however, OPT-SI will prioritize JavaScript files that dynamically create above-the-fold content via DOM methods like `document.appendChild()`. Figure 9 provides an example of a real dependency graph, and the nodes that are prioritized by the various schedulers.

Figure 10 compares the performance of the schedulers. OPT-RI and OPT-SI reduce all load metrics, *but the targeted metrics decrease the most*. Thus, sites that want to decrease time-to-interactivity must explicitly target RI and RT, not preexisting metrics like SI and PLT. For example, consider the search button in Figures 2(b) and 2(c). OPT-RI makes the button interactive 1.5 seconds earlier than OPT-SI. Differences of that magnitude have significant impacts on user satisfaction and site revenue [6, 8, 41].

As shown in Figure 10, OPT-RI reduces RI by a median of 29%, and RT by a median of 32%; PLT, AFT, and SI also drop, but not as much (by 23%, 15%, and 12%, respectively). Interestingly, the default Polaris scheduler (OPT-PLT) improves PLT, RT, and RI, but actually *hurts* AFT and SI by -4% and -7% at the median. The reason is that JavaScript files often form long dependency chains; evaluating one JavaScript file in the chain leads to the fetching and evaluation of additional JavaScript files. These long dependency chains tend to lie along the dynamic critical paths that are preferentially explored by OPT-PLT. By focusing on those chains, OPT-PLT increases the speed at which event handling state is loaded. However, this approach defers the loading of content in short chains. Short chains often contain images, since images (unlike HTML, CSS, and JavaScript) cannot trigger new object fetches. Deferring image loading hurts AFT and SI, though RT and RI improve, and the likelihood of broken user interactions (§5.2 and §5.3) decreases.

User Study 1: Do User-perceived Rendering Times Actually Change? The results from Figure 10 programmatically compare OPT-PLT, OPT-SI, and OPT-RI. We now evaluate how the differences between these optimization strategies are perceived by real users. We performed a user study in which 73 people judged the load times of 15 randomly-selected sites from our corpus, each of which had three versions (one for each optimization strategy). We used a standard methodology for evaluating user-perceived load times [20, 35]. We pre-

Weights	PLT	RT	AFT	SI	RI
OPT-PLT	**36% (51%)**	13% (22%)	-4% (5%)	-7% (4%)	8% (17%)
OPT-RI	23% (34%)	**32% (48%)**	15% (26%)	12% (20%)	**29% (35%)**
OPT-SI	10% (19%)	18% (31%)	**27% (39%)**	**18% (28%)**	14% (23%)

Figure 10: Median (95th percentile) load time improvements using our custom Polaris schedulers and the default one (OPT-PLT). Results used our entire 350-page corpus. Loads were performed on a desktop Chrome browser that had a 12 Mbit/s link with an RTT of 100 ms; the performance baseline was a regular (i.e., non-Polaris) page load. The best scheduler for each load metric is highlighted.

sented each user with 10 randomly-selected pages that employed a randomly-selected optimization target; we injected a JavaScript `keypress` handler into each page, so that users could press a key to log the time when they believed the page to be fully loaded. In all of the user studies, content was served from Mahimahi on a Macbook Pro, using an emulated 12 Mbit/s link with a 100 ms RTT.

Unsurprisingly, users believed that OPT-PLT resulted in the slowest loads for all 15 pages. However, OPT-SI did not categorically produce the lowest user-perceived rendering times; users thought that OPT-RI was the fastest for 4 pages, and OPT-SI was the fastest for 11. Across the study, median (95th percentile) user-perceived rendering times with OPT-RI were within 4.7% (10.9%) of those with OPT-SI. Furthermore, the performance of OPT-RI and OPT-SI were closer to each other than to that of OPT-PLT. At the median (95th percentile), OPT-RI was 14.3% (25.3%) faster than OPT-PLT, whereas OPT-SI was 17.4% (32.9%) faster.

These results indicate that a page that only wants to decrease rendering delays should optimize for SI. However, optimizing for RI results in comparable decreases in rendering time. Our next user study shows that optimizing for RI also decreases user-perceived time-to-interactivity.

User Study 2: Does OPT-RI Help Interactive Sites? Unlike the first user study, our second one asked users to interact with five well-known landing pages: Amazon, Macy's, Food Network, Zillow, and Walmart. For each site, users completed a site-specific task that normal users would be likely to perform. For example, on the Macy's page, users were asked to hover over the "shopping bag" icon until the page displayed a pop-up icon that listed the items in the shopping bag. On the Walmart site, users were asked to search for "towels" using the autocompleting text input at the top of the page; they then had to select the autocompleted suggestion. To avoid orientation delays, users were shown all five pages and the location of the relevant interactive elements at the beginning of the study. This setup emulated users who were returning to frequently-visited sites.

The study had 85 users interact with three different versions of each page: a default page load, a load that was optimized with OPT-SI, and one that was optimized with OPT-RI. For each page, users were presented with the three variations in a random order and were unaware of which variant they were seeing. Users were asked to select the variant that enabled them to complete the given task the fastest; if users felt that there was no perceivable difference between the loads, users could report "none."

Load method	Preference %
OPT-RI	83%
OPT-SI	4%
Default load	7%
None	6%

Figure 11: The results of our second user study. OPT-RI leads to human-perceivable reductions in the completion times for interactive tasks.

As shown in Figure 11, OPT-RI was overwhelmingly preferred, with 83% of users believing that OPT-RI led to the fastest time-to-interactivity. For example, on the Macy's page, OPT-RI made the shopping bag icon fully interactive *1.6 seconds faster* than the default page load, and *2.1 seconds faster* than the OPT-SI load. Time-to-interactivity differences of these magnitudes are easily perceived by humans. Thus, for pages with interactive, high-priority content, OPT-RI is a valuable tool for reducing time-to-interactivity (as well as the time needed to fully render the page). Optimizing for interactivity is particularly important for web browsing atop mobile devices with poor network connectivity. In these scenarios, users often desire to interact with pages as soon as relevant content becomes visible [17].

7 Conclusion

A web page is not usable until its above-the-fold content is both visible and functional. In this paper, we define Ready Index, the first load time metric that explicitly quantifies page interactivity. We introduce a new tool, called Vesper, that automates the measurement of Ready Index. Using a corpus of 350 pages, we show that Ready Index captures interactivity better than prior metrics like PLT and SI. We also present an automated page-rewriting framework that uses Vesper to optimize a page for Ready Index or pure rendering speed. User studies show that pages which optimize for Ready Index support more immediate user interactions with less user frustration.

Acknowledgments

We thank the anonymous reviewers and our shepherd Jon Howell for their helpful feedback. We also thank Tim Dresser at Google for his insights about how Google's TTI metric is currently defined. This research was partially supported by NSF grant 1407470 and by a Google Research Award.

References

[1] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proceedings of NSDI*, 2015.

[2] Alexa. Top Sites in United States. http://www.alexa.com/topsites/countries/US, 2018.

[3] M. Allman. Comments on Bufferbloat. *SIGCOMM Comput. Commun. Rev.*, 43(1):30–37, January 2012.

[4] Amazon. Silk Web Browser. https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html, 2018.

[5] M. Belshe. More Bandwidth Doesn't Matter (Much). Google. https://goo.gl/PFDGMi, April 8, 2010.

[6] J. Brutlag. Speed Matters. Google Research Blog. https://research.googleblog.com/2009/06/speed-matters.html, June 23, 2009.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of OSDI*, 2008.

[8] K. Eaton. How One Second Could Cost Amazon $1.6 Billion In Sales. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales, March 15, 2012.

[9] M. J. Freedman. Experiences with CoralCDN: A Five-year Operational View. In *Proceedings of NSDI*, 2010.

[10] Google. Perceptual Speed Index. https://developers.google.com/web/tools/lighthouse/audits/speed-index, December 14, 2017.

[11] Google. Chrome Debugging Protocol. https://chromedevtools.github.io/devtools-protocol/, 2018.

[12] Google. Reduce the size of the above-the-fold content. PageSpeed Tools Documentation. https://developers.google.com/speed/docs/insights/PrioritizeVisibleContent, January 9, 2018.

[13] Google. Remove Render-Blocking JavaScript. PageSpeed Tools Documentation. https://developers.google.com/speed/docs/insights/BlockingJS, January 25, 2018.

[14] Google. Speed Index: WebPagetest Documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2018.

[15] Google. Time to Interactive (TTI). https://github.com/WPO-Foundation/webpagetest/blob/master/docs/Metrics/TimeToInteractive.md, January 12, 2018.

[16] Google. WebPagetest: Website Performance and Optimization Test. https://www.webpagetest.org/, 2018.

[17] B. Greenstein. Delivering the Mobile Web to the Next Billion Users. Keynote speech: Workshop on Mobile Computing Systems and Applications (HotMobile), 2018.

[18] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proceedings of MobiSys*, 2012.

[19] P. Irish. Speedline. https://github.com/paulirish/speedline, November 21, 2017.

[20] C. Kelton, J. Ryoo, A. Balasubramanian, and S. Das. Improving User Perceived Page Load Times Using Gaze. In *Proceedings of NSDI*, 2017.

[21] P. Meenan. How Fast is Your Web Site? *ACM Queue*, 11(2), March 2013.

[22] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of NSDI*, 2010.

[23] Mozilla Developer Network. Document Object Model (DOM). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model, August 29, 2017.

[24] Mozilla Developer Network. HTTP Caching FAQ. https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching_FAQ, January 4, 2018.

[25] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of NSDI*, 2016.

[26] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of USENIX ATC*, 2015.

[27] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A Platform for High-performance Internet Applications. *SIGOPS Oper. Syst. Rev.*, 44(3), August 2010.

[28] D. Oksnevad. Time to Interact: A New Metric for Measuring User Experience. https://blog.dotcom-monitor.com/web-performance-tech-tips/time-to-interact-new-metric-measuring-user-experience/, February 3, 2014.

[29] Opera. Opera Mini. http://www.opera.com/mobile/mini, 2018.

[30] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race Detection for Web Applications. In *Proceedings of PLDI*, 2012.

[31] T. Poss. How Does Load Speed Affect Conversion Rate? Oracle: Modern Marketing Blog. https://blogs.oracle.com/marketingcloud/how-does-load-speed-affect-conversion-rate, January 14, 2016.

[32] T. Russo. Why Your Website Dev Team Should Care About Revenue. https://www.bluetriangletech.com/performance-insider/your-website-dev-team-should-care-about-revenue/, June 10, 2016.

[33] K. Sakamoto. Time to First Meaningful Paint. Chromium draft document. https://bit.ly/ttfmp-doc, June 6, 2016.

[34] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. PARCEL: Proxy Assisted BRowsing in Cellular Networks for Energy and Latency Reduction. In *Proceedings of CoNEXT*, 2014.

[35] M. Varvello, J. Blackburn, D. Naylor, and K. Papagiannaki. EYEORG: A Platform For Crowdsourcing Web Quality Of Experience Measurements. In *Proceedings of CoNEXT*, 2016.

[36] L. Wang, K. S. Park, R. Pang, V. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Proceedings of USENIX ATC*, 2004.

[37] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of NSDI*, 2013.

[38] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of NSDI*, 2016.

[39] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of NSDI*, 2013.

[40] S. Work. How Loading Time Affects Your Bottom Line. Kissmetrics Blog. https://blog.kissmetrics.com/loading-time/, April 28, 2011.

[41] Z. Yang. Every Millisecond Counts. https://www.facebook.com/note.php?note_id=122869103919, August 28, 2009.

[42] W. Young. The Need For Speed: 7 Observations On The Impact Of Page Speed To The Future Of Local Mobile Search. Search Engine Land. http://searchengineland.com/need-speed-7-observations-impact-page-speed-future-local-mobile-search-243128, February 29, 2016.

A Appendix

A.1 Additional Evaluation Results

In this section, we elaborate on the experimental results from Section 5.4, discussing how Ready Index applies to mobile page loads (§A.1.1), warm cache page loads (§A.1.2), and loads on different browser types (§A.1.3).

A.1.1 Mobile Page Loads

Mobile browsers run on devices with limited computational resources. As a result, mobile page loads are typically compute-bound, with less sensitivity to network latency [5, 34]. To explore RI and RT on mobile devices, we USB-tethered a Nexus 5 phone running Android 5.1.1 to a Linux desktop machine that ran Mahimahi. Mahimahi emulated a Verizon LTE cellular link [39] with a 100 ms RTT. The phone used Google Chrome v53 to load pages from a test corpus. The corpus had the same 350 sites from our standard corpus, but used the mobile version of each site if such a version was available. Mobile sites are reformatted to fit within smaller screens, and to contain fewer bytes to avoid expensive fetches over cellular networks.

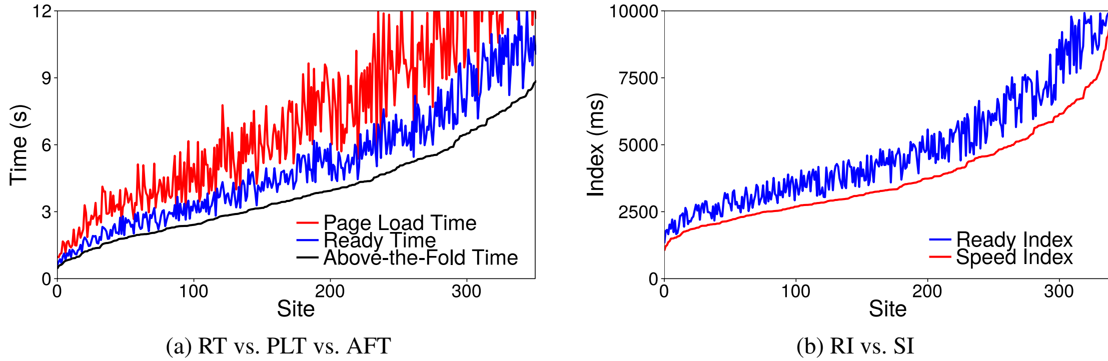
As shown in Figure 12, mobile page loads exhibit the same trends that we observed on more powerful client devices. For example, the median PLT is 35.2% larger than the median RT; the median RI is 29.7% larger than the median Speed Index. These differences persist even when considering only the mobile-optimized pages in our corpus. For that subset of pages, the median PLT is 27.4% larger than the median RT, and the median RI is 25.3% larger than the median Speed Index.

A.1.2 Browser Caching

Our prior experiments used cold browser caches, meaning that, to load a particular site, a browser had to fetch each of the constituent objects over the network. However, users often visit the same page multiple times; different sites also share objects. Thus, in practice, browsers often have warm caches that allow some object fetches to be satisfied locally.

To determine how warm caches affect page loads, we examined the HTTP caching headers [24] for each object in our corpus. For each object that was marked as cacheable, we rewrote the headers to indicate that the object would be cacheable forever. We then loaded each page in our corpus twice, back to back; the first load populated the cache, and the second one leveraged the prewarmed cache. Figure 13 shows the results for a desktop browser which used a 12 Mbit/s link with a 100 ms RTT.

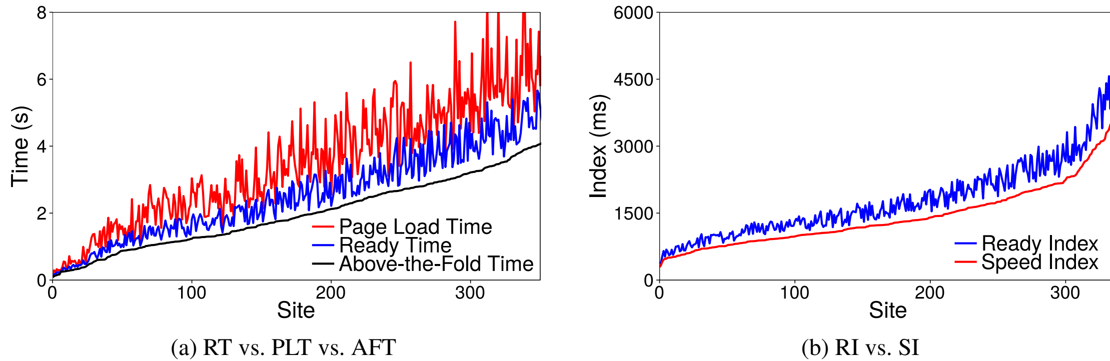
As expected, pages load faster when caches are warm. However, the general trends from Section 5.2 still hold. For example, the median PLT is 38.2% larger than the median RT, which is 26.0% larger than the median AFT. The correlations between various metrics also continue



(a) RT vs. PLT vs. AFT

(b) RI vs. SI

Figure 12: Comparing the load metrics for mobile pages loaded on a Nexus 5 phone. The network used an emulated Verizon LTE link with a 100 ms RTT.



(a) RT vs. PLT vs. AFT

(b) RI vs. SI

Figure 13: Page loads with warm browser caches. The desktop browser used a 12 Mbit/s link with a 100 ms RTT.

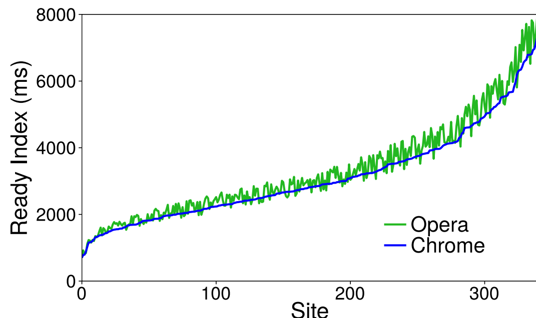


Figure 14: The Ready Index for each page in our corpus, as measured on Chrome and Opera. Pages are sorted on the x-axis by increasing Ready Index on Chrome. The results were collected using cold browser caches and a 12 Mbit/s link with an RTT of 100 ms.

to be noisy. For example, SI increases from 1147 ms to 1168 ms between sites 134 (`duckduckgo.com`) and 135 (`nexusmods.com`); however, RI decreases from 1601 ms to 1228 ms.

A.1.3 Cross-Browser Comparisons

Different browsers are built in different ways. As shown in Figure 14, those architectural variations impact page load times. Figure 14 compares Ready Index on Chrome v53 and Opera v42. Chrome and Opera both use the WebKit rendering engine and the V8 JavaScript runtime. However, the browsers' code is sufficiently different to produce noticeable biases in RI values: Chrome's RI values are 6.5% lower at the median, and 11.9% lower at the 95th percentile.

To understand the causes for such discrepancies, developers must analyze the steps that a browser takes to load a page. Tools like WProf [37] and the built-in Chrome debugger allow developers to examine coarse-grained interactions between high-level activities like HTML parsing, screen painting, and JavaScript execution. However, Vesper's logs describe how interactive state loads *at the granularity of individual JavaScript variables and DOM nodes.* For example, Vesper allows a developer to associate a dynamically-created text input with the specific code that creates the input and registers event handlers for the input; Vesper also tracks the JavaScript variables that are manipulated by the execution of the event handlers. None of this information is explicitly annotated by developers, nor should it be: for a large, frequently-changing site, humans should focus on the correct implementation of desired features, not the construction of low-level bookkeeping details about data and code dependencies. Thus, automatic extraction of these dependencies is crucial, since, as we demonstrate in Section 6, a fine-grained understanding of those dependencies is necessary to minimize a page's time-to-interactivity.

Towards Battery-Free HD Video Streaming

Saman Naderiparizi, Mehrdad Hessar, Vamsi Talla, Shyamnath Gollakota and Joshua R. Smith
University of Washington

Abstract – Video streaming has traditionally been considered an extremely power-hungry operation. Existing approaches optimize the camera and communication modules individually to minimize their power consumption. However, designing a video streaming device requires power-consuming hardware components and computationally intensive video codec algorithms that interface the camera and the communication modules. For example, monochrome HD video streaming at 60 fps requires an ADC operating at a sampling rate of 55.3 MHz and a video codec that can handle uncompressed data being generated at 442 Mbps.

We present a novel architecture that enables HD video streaming from a low-power, wearable camera to a nearby mobile device. To achieve this, we present an "analog" video backscatter technique that feeds analog pixels from the photo-diodes directly to the backscatter hardware, thereby eliminating power-consuming hardware components, such as ADCs and codecs. To evaluate our design, we simulate an ASIC, which achieves 60 fps 720p and 1080p HD video streaming for 321 μW and 806 μW, respectively. This translates to 1000x to 10,000x lower power than it used for existing digital video streaming approaches. Our empirical results also show that we can harvest sufficient energy to enable battery-free 30 fps 1080p video streaming at up to 8 feet. Finally, we design and implement a proof-of-concept prototype with off-the-shelf hardware components that successfully backscatters 720p HD video at 10 fps up to 16 feet.

1 Introduction

There has been recent interest in wearable cameras like Snap Spectacles [16] for applications ranging from life-casting, video blogging and live streaming concerts, political events and even surgeries [18]. Unlike smartphones, these wearable cameras have a spectacle form-factor and hence must be both ultra-lightweight and cause no overheating during continuous operation. This has resulted in a trade-off between the usability of the device and its streaming abilities since higher resolution video streaming requires a bigger (and heavier) battery as well as power-consuming communication and processing units. For example, Snap Spectacle, while lightweight and usable, cannot stream live video [16] and can record only up to one hundred 10-second videos (effectively less than 20 minutes) [16, 13] on a single charge.

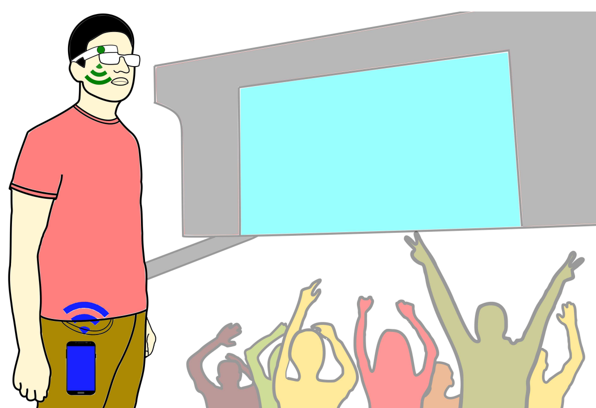
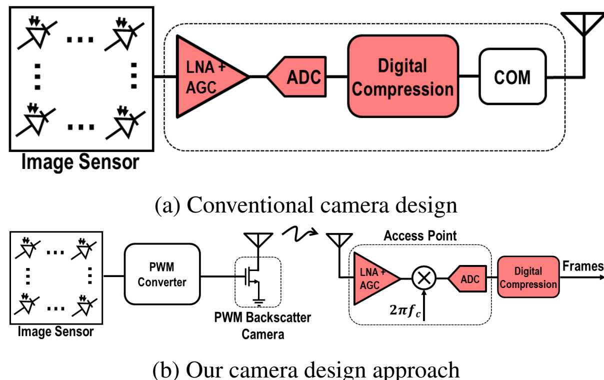


Figure 1: **Target application.** Our ultra-low-power HD streaming architecture targets wearable cameras. We achieve this by performing analog video backscatter from the wearable camera to a nearby mobile device (e.g., smartphone).

In this paper, we ask the following question: Can we design a low-power camera that can perform HD video streaming to a nearby mobile device such as a smartphone? A positive answer would enable a wearable camera that is lightweight, streams high quality video, and is safe and comfortable to wear. Specifically, reducing power consumption would reduce battery size, which in turn addresses the key challenges of weight, battery life and overheating. Finally, since users typically carry mobile devices like smartphones that are comparatively not as weight and power constrained, they can relay the video from camera to the cloud infrastructure.

To understand this challenge, let us look at the different components of a video-streaming device: image



(a) Conventional camera design

(b) Our camera design approach

Figure 2: The amplifier, AGC, ADC and compression module consume orders of magnitude more power than is available on a low-power device. In our design, these power hungry modules have been delegated to the reader, eliminating their power consumption overhead from the wireless camera.

sensor, video compression and communication. Image sensors have an optical lens, an array of photo-diodes connected to amplifiers, and an ADC to translate analog pixels into digital values. A video codec then performs compression in the digital domain to produce compressed video, which is then transmitted on the wireless medium. Existing approaches optimize the camera and communication modules individually to minimize power consumption. However, designing an efficient video streaming device requires power-consuming hardware components and video codec algorithms that interface to the camera and communication modules. Specifically, optical lens and photo-diode arrays can be designed to consume as little as 1.2 μW [22]. Similarly, recent work on backscatter can significantly lower the power consumption of communication to a few microwatts [26, 23] using custom ICs. Interfacing camera hardware with backscatter, however, requires ADCs and video codecs that significantly add to power consumption.

Table 1 shows the sampling rate and data rate requirements for the ADC and video codec, respectively. HD video streaming requires an ADC operating at a high sampling rate of more than at least 10 MHz. While the analog community has reduced ADC power consumption at much lower sampling rates [46, 20], state-of-the-art ADCs in the research community consume at least a few milliwatts at the high sampling rates [34]. Additionally, the high data rate requires the oscillator and video codec to run at high clock frequencies, which proportionally increases power consumption. Specifically, video codecs at these data rates consume 100s of milliwatts to a few watts of power [2].

We present a novel architecture that enables video streaming on low-power devices. Instead of independently optimizing the imaging and the communication modules, we jointly design these components to significantly reduce system power consumption. Our architecture, shown in Fig. 2b, takes its inspiration from the Great Seal Bug [5], which uses changes in a flexible metallic membrane to reflect sound waves. Building on this idea, we create the first "analog" video backscatter system. At a high level, we feed analog pixels from the photo-diodes directly to the backscatter antenna; we thus eliminate power-hungry ADCs, amplifiers, AGCs and codecs.

Figure 3: **Sample HD video frame streamed with our analog video backscatter design.** Our prototype was placed 4 feet from the reader.

Our intuition for an analog video backscatter approach is to shift most of the power-hungry, analog-to-digital conversion operations to the reader. Because analog signals are more susceptible to noise than digital ones, we split the ADC conversion process into two phases, one performed at the video camera, and one accomplished at the reader. At the video camera, we convert analog pixel voltage into a pulse that is discrete in amplitude but continuous in time. This signal is sent via backscatter from the camera to the reader. Avoiding the amplitude representation in the wireless link provides better noise immunity. The reader measures the continuous length pulse it receives to produce a binary digital value. Philosophically, the continuous-time, discrete amplitude pulse representation used in the backscatter link resembles that used in an extremely power-efficient biological nervous system, which encodes information in spikes that do not vary in amplitude but are continuous in time [45].

Specifically, our design synthesizes three key techniques. First, we show how to interface pixels directly to the backscatter hardware without using ADCs. To do, this we transform analog pixel values into different pulse widths using a passive ramp circuit and map these pulses into pixels at the reader. Second, we achieve intra-frame compression by leveraging the redundancy inherent in typical images. Intuitively, the signal bandwidth is proportional to the rate of change across adjacent pixels; since videos tend to be redundant, the bandwidth of the analog signal is inversely proportional to the redundancy in the frame. Thus, by transmitting pixels consecutively, we can implicitly perform compression and significantly reduce wireless bandwidth. Finally, to achieve inter-frame compression, we design a distributed algorithm that reduces the data the camera transmits while delegating most

Table 1: Raw digital video sampling and bitrate requirement

	Frame Rate: 60 fps		Frame Rate: 30 fps		Frame Rate: 10 fps	
Video Quality	**Sampling Rate (MHz)**	**Data Rate (Mbps)**	**Sampling Rate (MHz)**	**Data Rate (Mbps)**	**Sampling Rate (MHz)**	**Data Rate (Mbps)**
1080p (1920x1080)	124.4	995.3	62.2	497.7	20.7	165.9
720p (1280x720)	55.3	442.4	18.4	221.2	9.2	73.7
480p (640x480)	18.4	147.4	9.2	73.7	3.1	24.58
360p (480x360)	10.4	82.9	5.2	41.5	1.7	13.8

inter-frame compression functionality to the reader. At a high level, the camera performs averaging over blocks of nearby pixels *in the analog domain* and transmits these averaged values using our backscatter hardware. The reader compares these averages with those from the previous frame and requests only the blocks that have seen a significant change in the average pixel value, thus reducing transmission between subsequent video frames.

We implement a proof-of-concept prototype of our analog backscatter design on an ultra-low-power FPGA platform and a custom implementation of the backscatter module. Because no HD camera currently provides access to its raw pixel voltages, we connect the output of a DAC converter to our backscatter hardware to emulate the analog camera and stream raw analog video voltages to our backscatter prototype. Fig. 3 shows a sample frame from an HD video streamed with our backscatter camera. More specifically, our findings are:

- We stream 720p HD video at 10 frames per second up to 16 feet from the reader. The Effective Number of Bits (ENOB) received for each pixel at distances below six feet exceeds 7 bits. For all practical purposes, these results are identical to the quality of the source HD video.
- Our inter and intra-frame compression algorithms reduces total bandwidth requirements by up to two orders of magnitude compared to raw video. For example, for 720p HD video at 10 fps, our design uses a wireless bandwidth of only 0.98 MHz and 2.8 MHz in an average-case and worst-case scenario video, respectively.

We design and simulate an ASIC implementation for our system, taking into account power consumption for the pixel array. Our results show that power consumption for video streaming at 720p HD is 321 μW and 252 μW for 60 and 30 fps, respectively. Power consumption at 1080p full-HD is 806 μW and 561 μW at 60 and 30 fps, respectively. We run experiments with RF power harvesting from the reader which shows that we can support 1080p full-HD video streaming at 30 fps up to distances of 8 feet from the reader. *Our results demonstrate that we can eliminate batteries and achieve HD video streaming on battery-free devices using our analog video backscatter approach.*

2 A Case for Our Architecture

Fig. 2a shows the architecture of a traditional wireless camera. Photodiodes' output is first amplified by a low noise amplifier (LNA) with automatic gain control (AGC). The AGC adjusts the amplifier gain to ensure that the output falls within the dynamic range of the analog to digital converter (ADC). Next, the ADC converts the analog voltage into discrete digital values. The video codec then compresses these digital values, which are transmitted on the wireless communication link.

Unfortunately, this architecture cannot be translated to an ultra-low-power device. Although camera sensors consisting of an array of photodiodes have been shown to operate on as low as 1.2 μW of power [22] at 128×128 resolution, amplifiers, AGC, ADC and the compression block require orders of magnitude higher power. Furthermore, power consumption increases as we scale the camera's resolution and/or frame rate. A low resolution 144p video recorded at 10 fps requires an ADC sampling at 368 KSPS to generate uncompressed data at 2.95 Mbps. With the latest advancements in ADCs [33] and backscatter communication [26], uncompressed data can be digitally recorded using low-power ADCs and transmitted using digital backscatter while consuming below 100 μW of power, which is within the power budget of energy harvesting platforms. However, as we scale the resolution to HD quality and higher, such as 1080p and 1440p, the ADC sampling rate increases to 10-100 MHz, and uncompressed data is generated at 100 Mbps to Gbps. ADCs operating at such high sampling rates consume at least a few mW [34]. Further, a compression block that operates in real time on 100 Mbps to 1 Gbps of uncompressed video consumes up to one Watt of power [2]. This power budget exceeds by orders of magnitude that available on harvesting platforms. Thus, while existing architectures might operate on harvested power for low-resolution video by leveraging recent advancements in low-power ADCs and digital backscatter, as we scale the resolution and/or the frame rate of the wireless camera, these architectures' use of ADCs and compression block drives up power consumption to levels beyond the realm of battery-free devices.

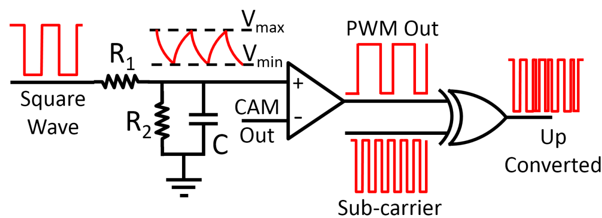


Figure 4: **Architecture of the PWM converter**.

3 System Design

In the rest of this section, we first describe our analog video transmission scheme followed by the intra-frame compression and finally the interactive distributed inter-frame compression technique.

3.1 Analog Video Backscatter

Problem. At a high level, our design eliminates the ADC on the camera to significantly reduce power consumption. This limits us to working with analog voltage output from the photodiodes. A *naive* approach would leverage the existing analog backscatter technique used in wireless microphones [48] to implement an analog backscatter camera. In an analog backscatter system, sensor output directly controls the gate of a field-effect transistor (FET) connected to an antenna. As the output voltage of the sensor varies, it changes the impedance of the FET which amplitude modulates the RF signal backscattered by the antenna. The reader decodes sensor information by demodulating the amplitude-modulated backscattered RF signal. However, this approach cannot be translated to a camera sensor. Photodiode output has a very limited dynamic range (less than 100 mV under indoor lighting conditions). These small voltage changes map to a very small subset of radar cross-sections at the antenna [19]. As a result, the antenna backscatters a very weak signal. Since wireless channel and receivers add noise, this approach results in a low SNR signal at the reader, which limits the system to both poor signal quality and limited operating range. One can potentially surmount this constraint by introducing a power-hungry amplifier and AGC, but this would negate the power-saving potential of analog backscatter.

Our solution. Instead of using amplitude modulation, which is typical in existing backscatter systems [52, 26], we use pulse width modulation (PWM) [24] to convert the camera sensor's analog output into the digital domain. At a high level, PWM modulation converts analog input voltage to different pulse widths. Specifically, the output of PWM modulation is a square wave, where the duty cycle of the output square wave is proportional to the analog voltage of the input signal. PWM signal harmonics do not encode any sensor information that is not already present in the fundamental. The harmonics can be considered a redundant representation of the fundamental. While they may add robustness, they contain no additional information. Thus, without causing any information loss, higher order components can be eliminated via harmonic cancellation techniques introduced in prior work [47].

As we show next, a PWM converter can implemented with passive RC components and a comparator, thereby consuming very low power. Fig. 4 shows the PWM converter architecture. The input is a square wave operating at frequency f, which is determined by the camera's frame rate and resolution. First, the square wave is low-pass filtered by an RC network to generate a triangular waveform, as shown in the figure. This waveform is then compared to the pixel value using a comparator. The comparator outputs a zero when the triangular signal is less than the pixel value and a one otherwise. Thus, the pulse width generated changes with the pixel value: lower pixel values have a larger pulse duration, while higher pixel values have a smaller pulse duration. We choose the minimum and maximum voltages for the triangular signal to ensure that the camera's pixel output is always within this limits.

The final component in our hardware design is sub-carrier modulation, which addresses the problem of self-interference. Specifically, in addition to receiving the backscatter signal, the receiver also receives a strong interference from the transmitter. Since the sensor data is centered at the carrier frequency, the receiver cannot decode the backscattered data in the presence of a strong, in-band interferer. Existing backscatter communication systems, such as RFID and Passive Wi-Fi, address this problem using sub-carrier modulation. These systems use a sub-carrier to shift the signal from the carrier to a frequency offset Δf from the carrier frequency. The receiver can then decode the backscattered signal by filtering out of band interference. Another consideration in choosing sub-carrier frequency is to avoid aliasing; sub-carrier frequency should not be smaller than the effective bandwidth of the analog signal.

Our PWM-based design integrates subcarrier modulation. We implement this modulation with a simple XOR gate. The sub-carrier can be approximated by a square wave operating at Δf frequency. We input sub-carrier and PWM output to an XOR gate to up-convert the PWM signal to a frequency offset Δf. Sub-carrier modulation addresses the problem of self-interference at the reader, and as a result, the PWM backscattering wireless camera can now operate at a high SNR and achieve broad operating ranges. We show in §5.1 that our PWM backscatter wireless camera can operate at up to 16 feet for a 2.76 MHz bandwidth 10 fps monochrome video signal in HD resolution. We also show in §6 that our camera system can work at up to 150 feet for a 50 KHz video signal in 112×112 resolution, over a 4× improvement relative to the 3 kHz bandwidth analog backscatter wireless microphone [50].

Table 2: Intra-frame compression for average/worst-case scenarios across 100 videos.

	Frame Rate: 60 fps		Frame Rate: 30 fps		Frame Rate: 10 fps	
Video Resolution	**Raw Data Rate(Mbps)**	**Analog BW (MHz)**	**Raw Data Rate(Mbps)**	**Analog BW (MHz)**	**Raw Data Rate(Mbps)**	**Analog BW (MHz)**
1080p	995.3	10.8/30.5	497.6	5.6/18.7	165.8	1.9/9.3
720p	442.3	6.3/15.6	221.1	8/3.1	73.7	0.98/2.8
480p	147.4	3/6.7	73.7	1.6/3.3	24.5	0.53/1.4
360p	82.9	1.9/3.9	41.4	0.94/2.3	13.8	0.32/0.78

3.2 Intra-Frame Compression

There is significant redundancy in the pixel values of each frame of uncompressed video. Redundancy occurs because natural video frames usually include objects larger than a single pixel, which means that the colors of nearby pixels are highly correlated. At the boundaries of objects (edges), larger pixel variations can occur; conversely, in the interior of an object, the amount of pixel variation is considerably less than the theoretical maximum. The net result of pixel correlations is a reduction in the information needed to represent natural images to levels far below the worst-case maximum. Traditional digital systems use a video codec to reduce pixel value redundancy. Specifically, the ADC first digitizes the camera's raw output at the Nyquist rate determined by the resolution and frame rate. The raw digital data stream is then fed to the video codec, which implements compression algorithms.

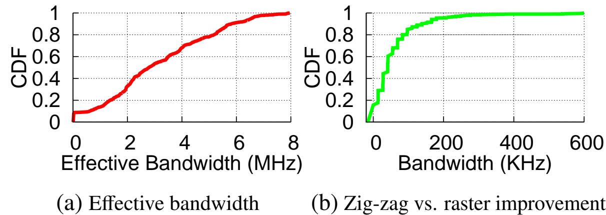
In the absence of the ADC, our wireless camera transmits analog video directly. However, we note that the bandwidth of any analog signal is a function of the new information contained in the signal. Inspired by analog TV broadcast, which transmits pixel information in a raster scan (left to right), we introduce and implement a zig-zag pixel scanning technique: pixels are scanned from left to right in odd rows and from right to left in even rows. The intuition here is that neighboring pixels have less variation, and the resulting signal would thus occupy less bandwidth.

We evaluate how well zig-zag transmission performs in terms of bandwidth reduction. We download one hundred 60 fps Full-HD (1080p) videos from [17] to use as our baseline. These videos range from slow to fast moving and contain movie action stunts, running animals, racing, etc. To create video baselines at different resolutions and frame rates, we resize and subsample these Full-HD video streams. For each of the resulting video resolutions and frame rates, we create an analog video by zig-zagging the pixels as described above. We then apply low-pass filters with different bandwidths on this analog signal and report the minimum bandwidth at which the PSNR of the resulting signal exceeds 30 dB. The bandwidth requirement reported in Table 2 shows the average/worst-case scenario, i.e., the bandwidth that ensures the recovery of average/worst-case videos with minimal quality degradation. We outline the uncompressed digital data rate for reference. Compared to the digital uncompressed data sampled at the Nyquist rate, the analog signal occupies 17.7–32.6x less bandwidth for the worst-case and 43–92x for the average-case, demonstrating our intra-frame compression technique's capability. Fig. 5a shows the CDF of effective bandwidth for our 30 fps 720p video dataset and demonstrates that an average-case 30 fps 720p video acheives up to a 71× improvement compared to raw digital video transmission.

Finally, we note that, compared to raster, a zig-zag scan faces less discontinuity in pixel values: instead of jumping from the last pixel in a row to the first pixel in the next row, it continues at the same column in the next row, thereby taking greater advantage of the video's inherent redundancy. This further lowers bandwidth utilization in the wireless medium. As an example, on average, zig-zag pixel scanning occupies ~120KHz and ~60KHz less bandwidth than raster scanning in a 60 fps and 30 fps 720p video stream, respectively. Fig. 5b shows the CDF of bandwidth improvement for zig-zag scanning over raster in our one hundred 30 fps 720p videos dataset. The plot makes clear that use of zig-zag pixel scanning provides greater bandwidth efficiency than its raster counterpart.

3.3 Distributed Inter-Frame Compression

In addition to the redundancy within a single frame, raw video output also has significant redundancy between consecutive frames. Existing digital architectures use a video codec to compress the raw video prior to wireless transmission to remove inter-frame redundancy and reduce power and bandwidth. Our approach to address this challenge is to once again leverage the reader. Like backscatter communication, where we move power-hungry components (such as ADCs) to the reader, we also move compression functionality to the reader. Specifically, our design distributes the compression algorithm between the reader and the camera. We delegate power-hungry computation to the reader and leverage the fact that our camera system can transmit *super-pixels*. A super-pixel value is the average of a set of adjacent pixel values. A



(a) Effective bandwidth (b) Zig-zag vs. raster improvement

Figure 5: **CDF of effective bandwidth for zig-zag pixel scanning (a), and improvement it provides over raster scanning (b).**

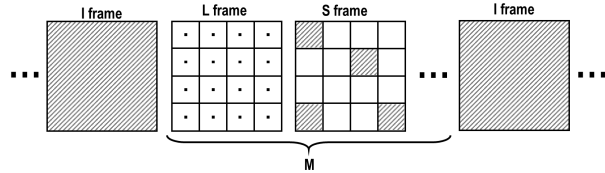


Figure 6: **Distributed compression.** The sequence of frames and pixels transmitted by the camera to the reader.

camera frame consists of an $N \times N$ array of pixels, which can be divided into smaller sections of $n \times n$ pixels. A super-pixel corresponding to each section is the average of all the pixels in the $n \times n$ frame section. The camera sensor, a photodiode, outputs a current proportional to the intensity of light. The camera uses a buffer stage at the output to convert the current into an output voltage. To compute the super-pixel, the camera first combines the current from the set of pixels and then converts the combined current into a voltage output, all in the analog domain. We note that the averaging of close-by pixels is supported by commercial cameras including the CentEye Stonyman camera [4].

Instead of transmitting the entire $N \times N$ pixel frame, the camera transmits a lower resolution frame consisting of $n \times n$ sized super-pixels (the average value of the pixels in the $n \times n$ block), called the low-resolution frame or $\boldsymbol{L}$ frame. Doing so reduces the data transmitted by the camera by a factor of $\frac{N^2}{n^2}$. The reader performs computation on $\boldsymbol{L}$ frames and implements a change-driven compression technique. At a high level, the reader compares in real time the incoming $\boldsymbol{L}$ frame with the previous $\boldsymbol{L}$ frame. If a super-pixel value differs by more than a predetermined threshold between frames, then the super-pixel has sufficiently changed, and the reader asks the camera to transmit all pixels corresponding to it. If the difference does not cross the threshold, then the reader uses the pixel values corresponding to the previous reconstructed frame to synthesize the new frame and does not request new pixel values. We call the frame that contains the pixel values corresponding to the sufficiently changed super-pixels, the super-pixel frame or $\boldsymbol{S}$ frame. In addition to transmitting the $\boldsymbol{S}$ and $\boldsymbol{L}$ frames, the camera periodically transmits uncompressed frames ($\boldsymbol{I}$) to correct for potential artifacts and errors that the compression process may have accumulated.

In streaming camera applications, the communication overhead of the reader requesting specific pixel values is minimal and is implemented using a downlink similar to prior 100-500 kbps designs [26, 44], where the receiver at the camera uses a simple envelope detector to decode the amplitude modulated signal from the reader. We note that prior designs can achieve Mbps downlink transmissions [42] as well as full-duplex backscatter [31], which can be used to remove the downlink as a bottleneck.

Fig. 6 shows the sequence of frames and pixels transmitted by our camera. Between two $\boldsymbol{I}$ frames, the camera transmits M low-resolution $\boldsymbol{L}$ frames and M super-pixel $\boldsymbol{S}$ frames which contain pixel values corresponding to super-pixels whose value differences exceed the threshold between consecutive frames. The number of $\boldsymbol{L}$ and $\boldsymbol{S}$ frames (M) transmitted between consecutive $\boldsymbol{I}$ frames trades off between the overhead associated with transmission of full resolution frames and the artifacts and errors the compression algorithm introduced. In our implementation of 10 fps HD video streaming, we transmit an $\boldsymbol{I}$ frame after a transmission of every 80 $\boldsymbol{L}$ and $\boldsymbol{S}$ frames.

4 Implementation

We built our wireless camera using off-the-shelf components on a custom-designed Printed Circuit Board (PCB). We use the COTS prototype to evaluate the performance of the wireless camera in various deployments. We then present the application-specific integrated circuit (ASIC) design of the wireless camera that we used to quantify the power consumption for a range of video resolutions and frame rates.

COTS implementation. Our wireless camera design eliminates the power-hungry ADCs and video codecs and consists only of the image sensor, PWM converter, a digital block for camera control and sub-carrier modulation, a backscatter switch and an antenna. We built two hardware prototypes, one for the high definition (HD) and another for the low-resolution version of the camera.

We built the low-resolution wireless camera using the 112×112 grayscale random pixel access camera from CentEye [4], which provides readout access to individual analog pixels. We implement the digital control block on a low-power Igloo Nano FPGA by Microsemi [7]. The analog output of the image sensor is fed to the PWM converter built using passive RC components and a Maxim NCX2200 comparator [10]. We set $R_1 = 83K\Omega$, $R_2 = 213K\Omega$ and $C = 78pF$ in our PWM converter design to support video frame rates of up to 13 fps. PWM converter's output acts as input to the FPGA. The FPGA performs sub-carrier modulation at 1.024 MHz using an XOR gate and outputs the sub-carrier modulated PWM signal to the Analog Devices ADG919 switch which switches a 2 dBi dipole antenna between open and short impedance states. The FPGA injects frame and line synchronization

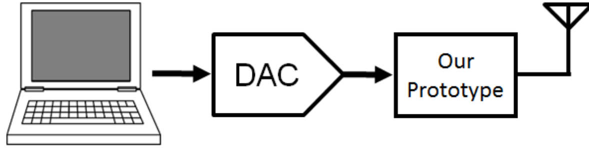


Figure 7: **HD wireless camera prototype.** A laptop emulating an analog output camera feeds raw pixel data into our backscatter prototype. Then the PWM encoded pixels are backscattered to the reader.

patterns into the frames data before backscattering. We use Barker codes [3] of length 11 and 13 for our frame and line synchronization patterns, respectively. Barker codes have a high-autocorrelation property that helps the reader more efficiently detect the beginning of the frame in the presence of noise.

We use Verilog to implement the digital state machine for camera control and sub-carrier modulation. Verilog design can be easily translated into ASIC using industry standard EDA tools. We can further reduce our system's power consumption by using the distributed compression technique. As described in §6, a camera deployed in a normal lab space can achieve an additional compression ratio of around 30×, which proportionately reduces wireless transmissions.

To develop an HD-resolution wireless camera, we need access to the raw analog pixel outputs of an HD camera. Currently, no camera on the market provides that access. To circumvent this constraint, we download from YouTube HD-resolution sample videos lasting 1 minute each. We output the recorded digital images using a USB interface to an analog converter (DAC) to simulate voltage levels corresponding to an HD quality image sensor operating at 10 fps. Given the USB speeds, we achieve the maximum frame rate of 10 fps. We feed the voltage output to our PWM converter. Fig. 7 shows the high-level block diagram of this implementation. For the high-resolution version of the wireless camera, we set $R_1 = 10K\Omega$, $R_2 = 100K\Omega$ and $C = 10pF$ and use an LMV7219 comparator by Texas Instruments [8] in our PWM converter. The digital block and other system components were exactly the same as for the low-resolution wireless camera described above except that sub-carrier frequency is set to ~10 MHz here to avoid aliasing.

ASIC Design. As noted, our design eliminates the power-hungry LNA, AGC and ADC at the wireless camera by delegating them to the reader to reduce wireless camera power consumption by orders of magnitude. However, since commercially available Stonyman cameras (like CentEye) and components (such as FPGA) are designed for flexibility and ease of prototyping and are not optimized for power, our COTS implementation cannot achieve the full power savings from our design. Therefore, we analyze the power consumption of an application-specific integrated circuit (ASIC) implementation of our

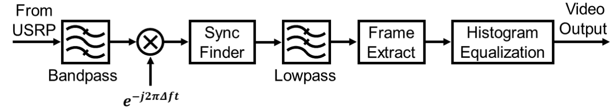


Figure 8: **Signal processing steps at the reader.** Recovering video from a PWM backscatter signal.

design for a range of video resolutions and frame rates. An ASIC can integrate the image sensor, PWM converter, digital core, oscillator and backscatter modulator onto a small silicon chip. We implement our design in a TSMC 65 nm LP CMOS process.

We use Design Compiler by Synopsis [14] to synthesize transistor level from the behavioral model of our digital core, which is written in Verilog. We custom-design the PWM converter, oscillator and backscatter modulator described in §3 in Cadence software and use industry standard simulation tools to estimate power. To support higher resolution and higher frame rate video, we simply increase the operating frequency of the oscillator, PWM converter and digital core. As an example, 360p at 60 fps requires a 10.4 MHz input clock, which consumes a total of 42.4 μW in the digital core, PWM converter and backscatter switch; a 1080p video at 60 fps requires an ~124.4 MHz input clock, which consumes 408 μW in the digital core, PWM converter and backscatter switch. To eliminate aliasing in all cases, we choose a sub-carrier frequency equal to the input clock of each scenario. Note that sub-carrier frequency cannot be lower than the effective bandwidth of the signal reported in Table 2.

We use existing designs to estimate the power consumption of the image sensor for different video resolutions. State-of-the-art image sensors consume 3.2pW/($frame \times pixel$) [51] for the pixels-only image sensor, which results in 33.2 μW for 360p resolution; this increases to 398 μW for 1080p resolution video at 60fps. Table 3 shows the power consumption of the ASIC version of our wireless camera for different video resolution and frame rates. Note that these results show power consumption before inter-frame compression distributed across the reader and camera, which could further reduce wireless bandwidth and power consumption.

Reader Implementation. We implement the reader on the X-300 USRP software-defined radio platform by Ettus Research [15]. The reader uses a bi-static radar configuration with two 6 dBi circularly polarized antennas [1]. Its transmit antenna is connected to a UBX-160 daughterboard, which transmits a single-tone signal. USRP output power is set to 30 dBm using the RF5110 RF power amplifier [11]. The receive antenna is connected to another UBX-160 daughter board configured as a receiver, which down-converts the PWM modulated backscattered RF signal to baseband and samples it at 10 Msps. The digital samples are transmitted to the PC via Ethernet.

Table 3: Power consumption

	Frame Rate: 60 fps	Frame Rate: 30 fps	Frame Rate: 10 fps
Video Quality	**Power (µW)**	**Power (µW)**	**Power (µW)**
1080p (1920x1080)	806.50	560.63	167.77
720p (1280x720)	320.94	252.10	78.31
480p (640x480)	126.88	106.78	36.71
360p (480x360)	75.63	65.68	25.11

Fig. 8 shows block diagram of the signal processing steps required to recover the transmitted video. For example, for low-resolution video, the received data is centered at an offset frequency of the 1.024 MHz; therefore, we first filter the received data using a 500 order bandpass filter centered at 1.024 MHz. Then, we down-convert the signal to baseband using a quadrature down-conversion mixer. Next, we correlate the received data with 13 and 11 bit Barker codes to determine the frame sync and line sync. After locating frame and line sync pulses, we extract the time periods corresponding to a row of PWM-modulated pixel values and low-pass filter the signal with a 500 order filter to remove out of band noise. We divide the row in evenly spaced time intervals that corresponded to the number of pixels in a single row of the image sensor. We recover the pixel value by calculating the average voltage of the signal, which corresponds to the duty cycle of the PWM-modulated signal. We sequentially arrange recovered pixel values into rows and columns to create video frames. Finally, we run a histogram equalization algorithm on the video to adjust frames intensity and enhance output video contrast [21].

5 Evaluation

We now evaluate various aspects of our wireless camera system. We start by characterizing the received video quality from the camera as a function of its distance to the reader. Next, we evaluate the performance of our wireless camera while it is worn by a user under different head positions and orientations. We then evaluate the power available by harvesting energy from RF signals transmitted by the reader to demonstrate the feasibility of a battery-free wireless camera. Finally, we evaluate the distributed interactive compression algorithm under two different scenarios.

5.1 Operational Range

We deploy our high-definition wireless camera device in a regular lab space. We use the USRP-based reader implementation (§4), which we set to transmit 23 dBm into a 6 dBi patch antenna. This is well below the 30 dBm maximum transmit power permitted by FCC in the 900 MHz ISM band. We vary the distance between the reader and

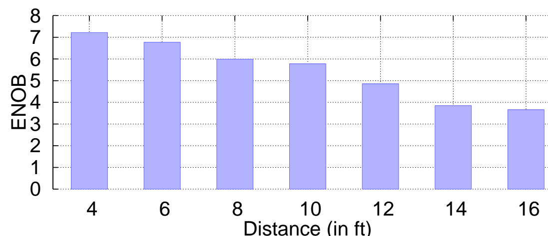


Figure 9: **ENOB of the Received Video.** ENOB of the received video versus distance of our wireless camera prototype from the reader.

the wireless camera prototype from 4 to 16 feet and configure the camera to repeatedly stream a 15-second video using PWM backscatter communication. The 720p resolution video is streamed at 10 fps, and the pixel values are encoded as 8-bit values in monochrome format. We record the wirelessly received video at the reader and measure the Signal to Noise Ratio (SNR). From that, we calculate the Effective Number of Bits (ENOB) [27] at the receiver.

We plot the ENOB of the video digitized by the reader as a function of distance between the reader and the wireless camera in Fig. 9. The plot shows that up to 6 feet from the reader, we achieve an ENOB greater than 7, which indicates negligible degradation in quality of the video, streamed using PWM backscatter. As the distance between the reader and the camera increase the SNR degrades, indicates a decrease in ENOB. Beyond the distance of 16 feet, we stop reliably receiving video frames. A separation of 16 feet between the reader and wearable camera is more than sufficient for wearable cameras, typically located a few feet away from readers such as smartphones. For reference, Fig. 10 shows frames from a 720p HD color video backscattered with our prototype at different distances from the reader; this resulted in different ENOB values.

We conclude the following: our analog video backscatter approach is ideal for a wearable camera scenario since the video is streamed to a nearby mobile device such as a smartphone. In this scenario, the SNR is high; hence, quality degradation due to an analog approach is not severe. Further, we gain significant power reduction benefits.

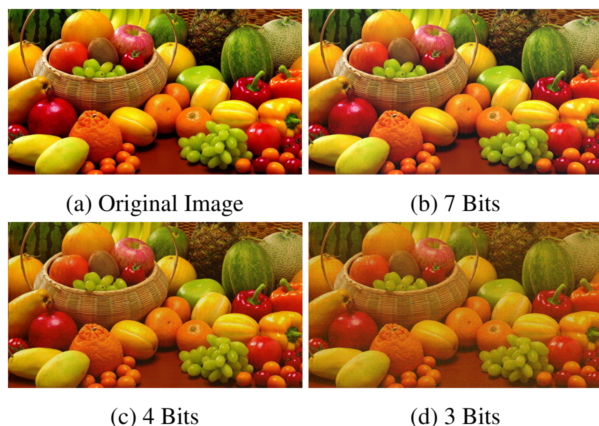


Figure 10: **Effective Number of Bits (ENOB) versus video quality.** The frame corresponding to an ENOB of 3 (d) shows video quality degradation.

5.2 Effect of Head Pose and Motion

We ask a participant to wear the antenna of our wireless camera on his head and perform different head movements and poses while standing around five feet from a reader with fixed location on a table. The poses included standing still, moving head left and right, rotating head to the side, moving head up and down, and talking. Hence, our evaluation includes scenarios with relative mobility between the reader and camera; in fact, it also includes cases where no line of sight communication exists between the reader and camera. We next, evaluate our wireless camera for in-situ applications and assess how movements and antenna contact with a body affect video quality. We record the streaming video with the reader and measure the SNR and ENOB as we did in §5.1. Fig. 11 plots the ENOB of the received video for five different poses and movements. This plot shows that we can achieve sufficient ENOB while performing most of these gestures, resulting in a high-quality video compared to the original source video.

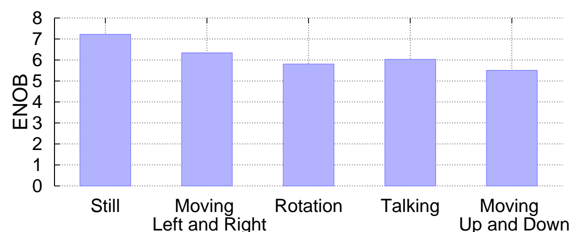


Figure 11: **ENOB of received video under different head motions.**

5.3 RF Power Harvesting

Next, we evaluate the feasibility of developing a battery-free HD streaming camera that operates by harvesting RF signals transmitted by the reader. We build an RF harvester for the 900 MHz ISM band based on a state-of-the-art RF harvester design [43]. The harvester consists of a 2 dBi dipole antenna and a rectifier that converts incoming RF signals into low-voltage DC output. The low-voltage DC is amplified by a DC-DC converter to generate the voltage levels to operate the image sensor and digital control block. We measure the power available at the output of the DC-DC converter.

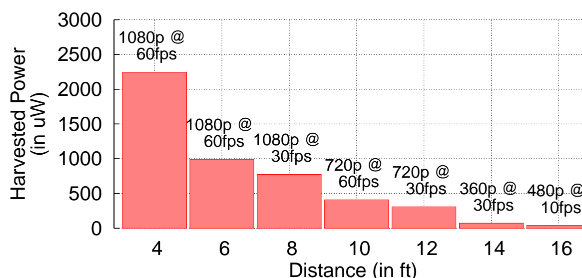


Figure 12: **Power harvesting.** We plot the average power harvested by the battery-free hardware over different distances from the reader.

We configure the reader to transmit a single tone at 30 dBm into a 6 dBi patch antenna and move the RF harvester away from the reader. Fig. 12 plots available power at the harvester as a function of distance. Based on available power, we also plot in Fig. 12 the maximum resolution and frame rate of the video that could be transmitted by an RF-powered ASIC version of our wireless camera. At close distances of 4 and 6 feet, we see sufficient power available from RF energy harvesting to operate the wireless camera at 60 fps 1080p resolution. As the distance increases, available power reduces, which lowers resolution of video being continuously streamed from the wireless camera. At 16 feet, the wireless camera continuously streams video at 10 fps 480p resolution; beyond this distance, the harvester does not provide sufficient power to continuously power the wireless camera. Note that Fig. 12 shows camera performance without using the distributed inter-frame compression algorithm described in §3.3. That algorithm, distributed across the wireless camera and reader, reduces camera transmissions, which lowers power consumption and consequently increases operating distances.

5.4 Distributed Inter-Frame Compression Evaluation

We consider two scenarios to evaluate our distributed inter-frame compression (subsection 3.3). We analyze HD video streamed from a fixed highway monitoring camera and from an action camera mounted on a user riding a motorcycle [9]. We evaluate the trade-off between the compression ratio and PSNR under both static and dynamic video feeds using our design. We measure the Peak Signal to Noise Ratio (PSNR) for different compression ratios by varying the threshold at which we consider the super-pixel (20 by 20 pixels) to have significantly changed.

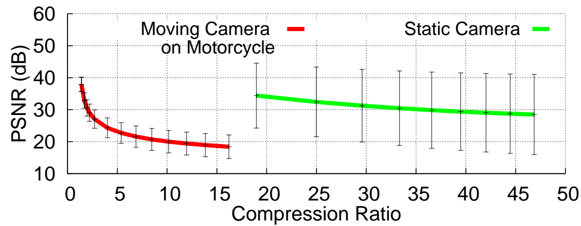


Figure 13: **Evaluation of distributed inter-frame compression algorithm for HD video.** The plots show PSNR at different compression ratio for two HD videos.

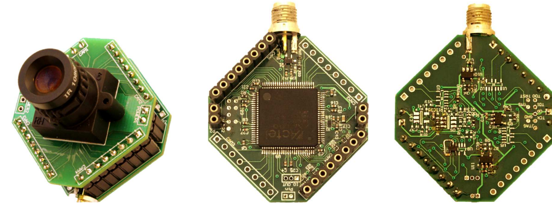


Figure 14: **Prototype of our low-resolution, video streaming home security camera.** Image of the analog camera, FPGA digital core and pulse width modulated (PWM) backscatter, all implemented using COTS components. The overall board measures 3.5 cm by 3.5 cm by 3.5 cm.

A higher threshold would result in higher compression ratios but at the cost of a degraded PSNR. Fig. 13 shows PSNR of the compressed video as a function of the compression ratio for our distributed inter-frame compression technique. For static cameras, we achieve a compression ratio of at least 35× while maintaining a PSNR above 30 dB. For dynamic videos recorded from a motorcycle, we achieve a compression ratio of 2× for a PSNR greater than 30 dB. This is expected since mobile scenarios significantly change majority of pixel values between frames, resulting in lower compression using our approach. We could address this by implementing a more complex compression algorithm at the reader to track moving objects in the frame and request specific pixel locations, achieving compression levels similar to video codecs. Implementing such complex compression algorithms, however, is beyond the scope of this paper.

6 Low-Resolution Security Camera

So far, we have demonstrated how high-resolution video offers a useful paradigm for a wearable camera system. However, various other applications, such as security systems and smart homes, do not require high-resolution video; lower resolution video would suffice for applications such as face detection. Specifically, wireless cameras are increasingly popular for security and smart home applications. In contrast to wearable camera applications, these cameras require low-resolution but much longer operating distances. To show that our design can extend to such applications, we evaluate the operation range at 13 fps 112×112 resolution. Our IC design at this resolution consume less than 20 μW without accounting for any distributed inter-frame compression saving.

To evaluate the range, we use a 13 fps 112×112 resolution, gray-scale random pixel access camera from CentEye [4] as our image sensor. The camera has a photodiode image sensor array, a trans-impedance buffer stage (to convert photodiode current to voltage) and a low-noise amplifier. It is extremely flexible, and the user can modify various settings, such as gain of the amplifier stage and, if desired, completely bypass the amplifier. We use this unique camera feature to prototype our wireless camera, which directly transmits analog values from the image sensor (sans amplification) using PWM backscatter. Fig. 14 shows photographs of our wireless camera prototype. The camera allows random access, i.e., any pixel on the camera can be accessed at random by setting the corresponding value in the row and column address registers. It can also be configured to output a single pixel, two adjacent pixels, or a super-pixel with sizes ranging from 2×2 to 7×7. We use the camera's random access and super-pixel functionality to implement our distributed inter-frame compression algorithm. The power consumption of our low-resolution, off-the-shelf analog video streaming prototype is 2.36 mW. We emphasize that this off-the-shelf camera is used only to demonstrate operational range; to achieve the tens of microwatts power budget, we need to use our ASIC design.

Deployment results. We deployed our wireless camera system in the parking lot of an apartment complex. We use the USRP-based reader implementation set to transmit 30 dBm into a 6 dBi patch antenna. We vary the distance between the wireless camera prototype, and, the reader and at each separation, we stream 20 seconds of video from the camera to the reader. Simultaneously, we record camera output using a high input impedance National Instrument USB-6361 DAQ as the ground truth. We choose the popular PSNR metric commonly used in video applications to compare the video wirelessly streamed to the reader using PWM backscatter to the ground truth video recorded at the camera using an NI DAQ. PSNR computes the difference between the ground truth and wirelessly received video.

We measure the PSNR of the received video to evaluate the performance of the wireless camera under normal lighting conditions (below 300 lux) at a frame rate of 13 fps. To evaluate how much our sole wireless communication method affects the quality of received video, we consider PWM converter output as the ground truth for PSNR measurement. Also, to isolate the impact of AGC, which occurs at the reader, unaltered video received by the reader prior to applying any AGC is compared to the ground truth for PSNR measurement. Fig. 15 plots the PSNR of the received video at the reader as a function of

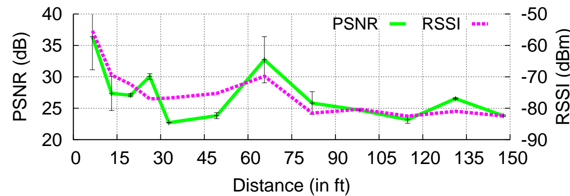


Figure 15: **Operational range of low-resolution security camera.**

the separation between the reader and the wireless camera. The plot shows that wireless camera streamed video at an average PSNR greater than 24 dB to a reader up to a distance of 150 feet away. Beyond 150 feet, the reader does not reliably decode the sync pulses, which limits the operating range of our wireless camera system. Thus, our analog backscatter approach achieves significantly longer range for low-resolution video compared to the HD version of the camera due to the trade-off between bandwidth/data rate and operating distances.

Applying our distributed inter-frame compression algorithm to low resolution videos. We deploy this security camera in a normal lab space and then implement our distributed inter-frame compression technique on the videos received. We evaluate the performance of our algorithm for three different super pixels sizes, 3×3, 5×5 and 7×7 pixels, and plot results in Fig. 16. We achieve a 29.4× data reduction using our distributed inter-frame compression technique while maintaining a PSNR greater than 30 dB.

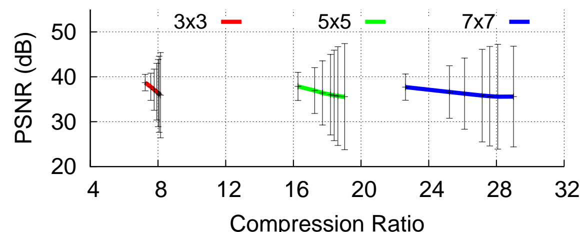


Figure 16: **Distributed interactive compression with our low-resolution security camera.** PSNR of the compressed video as a function of compression parameters in a typical lab setting.

Face detection accuracy. Next, we demonstrate that the quality of the video streamed from our low-resolution COTS implementation is sufficient for detecting human faces. Such a system can be used to detect human occupancy, grant access (such as Ring [12]), or set off an alarm in case of an intruder. To evaluate the system, we place the wireless camera at five different distances ranging from 16 to 100 feet from the reader. We ask ten users to walk around and perform gestures within 5 feet of the camera. We stream a 2 minutes video at each location at 13 fps and use the MATLAB implementation of the Viola-Jones algorithm to analyze the approximately four thousand video frames. Fig. 17 shows the accuracy of

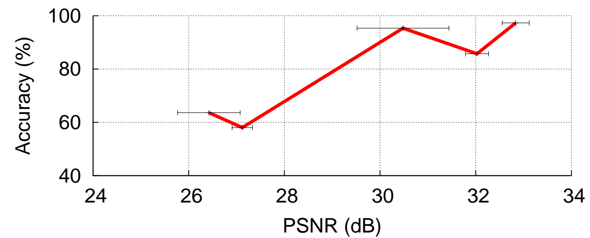


Figure 17: **Face detection with our low-resolution security camera.** We show the accuracy of face detection on the video streamed from our wireless camera.

face detection as a function of the PSNR of the received video: as the quality (PSNR) of the video improves, the accuracy of face detection increases. We accurately detect up to 95% of human faces when the PSNR exceeds 30 dB.

7 Related Work

Prior work falls in two different categories.

Backscatter communication. An early example of analog backscatter was a gift by the Russians to the US embassy in Moscow, which included a passive listening device. This spy device consisted of a sound-modulated resonant cavity. The voice moved the diaphragm to modulate the cavity's resonance frequency, which could be detected by analyzing the RF signals reflected by the cavity. [5]. A more recent example of analog backscatter is a microphone-enabled, battery-free tag that amplitude-modulates its antenna impedance using microphone output [50, 48]. In contrast, we design the first analog video backscatter system. Further, prior microphone designs had a low data-rate compared to video streaming. Our camera at 10 fps transmits about $9.2M$ pixels per second; for a microphone, a few kilo-samples of audio transmission is sufficient to fully recover the voice. In addition, our 13 fps $163K$ pixels per second camera operates at more than four times the range of the microphone in [50] due to pulse width modulation.

Ekhonet [53] optimizes the computational blocks between the sensor and the backscatter module to reduce the power consumption of backscatter-based wireless sensors. Our design builds on this work but differs from it in multiple ways: 1) prior work still uses ADCs and amplifiers on the cameras to transform pixels into the digital domain and hence cannot achieve streaming video on the limited harvesting power budget. In contrast, we provide the first architecture for battery-free video streaming by designing an analog video backscatter solution.

Recent work on Wi-Fi and TV-based backscatter systems [25, 26, 23, 30, 42] can achieve megabits per second of communication speed using a backscatter technique. Integrating these designs with our video backscatter approach would prove a worthwhile engineering effort.

Low-Power cameras. [41] introduces a self-powered camera that can switch its photo diodes between energy harvesting and photo capture mode. Despite being self-powered, these cameras do not have wireless data transmission capabilities. [49, 38, 40, 35, 36, 37] show that using off-the-shelf, low-resolution camera modules, one can build battery-free wireless cameras that will capture still images using the energy they harvested from RF waves, including Wi-Fi and 900 MHz transmissions. Despite their ability to transmit data wirelessly, they are heavily duty cycled and cannot stream video. In particular, these designs can send a new frame at most every ten seconds when they are very close to the RF power source (within about a foot) and once every few tens of minutes at longer distances [49].

[32] presents a 90×90 pixels image sensor with pixels that are sensitive to changes in the environment. If a pixel receives sufficient illumination variation, the pixel address will be stored in a FIFO, thus compressing the image to the pixels that have significantly changed. Despite enabling image compression to occur at the image sensor, this system does not stream live video. In addition, at this low resolution, it burns about 3 mW of power when running at 30 fps. [28] introduces a 128×128 pixel event-driven image sensor that emphasizes low latency for detecting very fast moving scenes, so its power consumption is orders of magnitude higher than our system's.

[29] addresses the problem of conventional image sensors' power consumption not scaling well as their resolution and frame rate increases. In particular, the authors propose to change the camera input clock and aggressively switch the camera to standby mode based on desired image quality. However, streaming video requires addressing the power consumption of multiple components, including camera, communication, and compression. Our work jointly integrates all these components to achieve the first battery-free video streaming design.

Finally, [39] shows that a regular camera on wearable devices burns more than 1200 mW, which limits the camera's operation time to less than two hours on a wearable device. They instead design a low-power wearable vision system that looks for certain events to occur in the field of view and turns on the primary imaging pipeline when those events happen. The power and bandwidth savings, however, are limited to the application and do not address communication. In contrast, we present the first battery-free video streaming application by jointly optimizing both backscatter communication and camera design and by eliminating power-consuming interfaces such as ADCs and amplifiers.

8 Limitations and Conclusion

This paper takes a significant first step in designing video streaming for battery-free devices. In this section, we discuss limitations and a few avenues for future research.

Security. Our current implementation does not account for security. However, to secure the wireless link between the camera and reader, we can leverage the fact that our digital core processes the PWM signal. Each wireless camera can be assigned a unique pseudo random security key. Based on this key, the camera's digital core can modulate the width of the PWM-encoded pixel value using an XOR gate. The reader, which knows the security key, can map the received data to the desired pixel values by performing the analogous operation.

ASIC versus off-the-shelf. While existing works on backscatter cameras focus on using off-the-shelf components, they treat cameras and backscatter independently and just interface the two. Thus, these works cannot achieve video streaming and the low-power demonstrated in this paper. Our key contributions are to make a case for a joint camera and backscatter architecture and to design the first analog video backscatter solution. However, this holistic architecture cannot be achieved with off-the-shelf components. The cost of building ICs in a research environment is prohibitively high. We believe that we spec out the IC design (in §4) in sufficient detail using industry standard EDA tools to take it from the lab to industry.

Mobile device as reader. To support our design for HD video streaming from wearable cameras, the smartphone must support a backscatter reader. We can use RFID readers that can be plugged into the headjack of the smartphone [6] to achieve this. In the future, we believe that backscatter readers would be integrated into smartphones to support video and other applications [23].

Enabling concurrent streaming. Our analog backscatter camera design can use frequency division multiplexing techniques to share the wireless medium across multiple devices. The reader can coordinate communication by assigning different frequency channels to each camera. Multiple cameras can simultaneously backscatter on different channels by using different (assigned) frequency offsets in our sideband modulation. However, evaluating this approach is beyond the scope of this paper.

9 Acknowledgments.

We thank Kyle Jamieson and the anonymous reviewers for their helpful feedback on the paper. Also, we thank Ye Wang for his efforts in building a real-time demonstration of our low-resolution security camera. This work was funded in part by NSF awards CNS-1407583, CNS-1305072, CNS-1452494, CNS-1420654, Google Faculty Research Awards and Sloan Fellowship.

References

[1] 6-dBi gain patch antenna. https://www.arcantenna.com/index.php/product_documents/get/document/id/492/. Accessed: March 2017.

[2] Ambarella a9 ultra hd 4k camera soc product brief. http://www.ambarella.com/uploads/docs/A9-product-brief.pdf.

[3] Barker code. http://mathworld.wolfram.com/BarkerCode.html. Accessed: September 2017.

[4] CentEye Stonyman image sensor datasheet. http://www.centeye.com/products/current-centeye-vision-chips/. Accessed: March 2017.

[5] The great seal bug. http://counterespionage.com/the-great-seal-bug-part-1.html. Accessed: March 2017.

[6] Head jack pluggable rfid reader for smartphones. http://www.rfidtagworld.com/products/Ear-Jack-reader-UHF_1056.html.

[7] IGLOO Nano FPGA datasheet. https://www.microsemi.com/document-portal/doc_view/130695-ds0110-igloo-nano-low-power-flash-fpgas-datasheet. Accessed: March 2017.

[8] Lmv7219 comparator. http://www.ti.com/lit/ds/symlink/lmv7219.pdfl. Accessed: September 2017.

[9] Moving camera on a motorcycle video clip. https://www.youtube.com/watch?v=sHj3xSG-R_E&t=376s.

[10] NCX2200I low power comparator datasheet. http://www.nxp.com/documents/data_sheet/NCX2200.pdf. Accessed: March 2017.

[11] RF5110 amplifier. http://www.rfmd.com/store/downloads/dl/file/id/30508/5110g_product_data_sheet.pdf. Accessed: March 2017.

[12] Ring video doorbell. https://ring.com/. Accessed: March 2017.

[13] Spectacles battery life. http://www.barrons.com/articles/snapchat-spectacles-review-the-good-the-bad-the-revolutionary-1487846715.

[14] Synopsis design complier. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx.

[15] USRP X-300. https://www.ettus.com/product/details/X300-KIT. Accessed: March 2017.

[16] Wearable spectacles. http://www.techradar.com/reviews/snap-spectacles.

[17] Youtube website. http://www.youtube.com.

[18] H. R. Beard, A. J. Marquez-Lara, and K. S. Hamid. Using wearable video technology to build a point-of-view surgical education library. *JAMA surgery*, 151(8):771–772, 2016.

[19] S. R. Best and B. C. Kaanta. A tutorial on the receiving and scattering properties of antennas. *IEEE Antennas and Propagation Magazine*, 51(5), 2009.

[20] L. Chen, X. Tang, A. Sanyal, Y. Yoon, J. Cong, and N. Sun. A 0.7-v 0.6-uw $100-ks/s$ low-power sar adc with statistical estimation-based noise reduction. *IEEE Journal of Solid-State Circuits*, 52(5):1388–1398, 2017.

[21] R. C. Gonzalez and R. E. Woods. *Image processing*, volume 2. 2007.

[22] S. Hanson, Z. Foo, D. Blaauw, and D. Sylvester. A 0.5 v sub-microwatt cmos image sensor with pulse-width modulation read-out. *IEEE Journal of Solid-State Circuits*, 45(4):759–767, 2010.

[23] V. Iyer, V. Talla, B. Kellogg, S. Gollakota, and J. Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 356–369, New York, NY, USA, 2016. ACM.

[24] K. Kapucu and C. Dehollain. A passive uhf rfid system with a low-power capacitive sensor interface. In *RFID Technology and Applications Conference (RFID-TA), 2014 IEEE*, pages 301–305. IEEE, 2014.

[25] B. Kellogg, A. Parks, S. Gollakota, J. R. Smith, and D. Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[26] B. Kellogg, V. Talla, S. Gollakota, and J. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *Usenix NSDI*, 2016.

[27] W. Kester. Understand sinad, enob, snr, thd, thd+ n, and sfdr so you don't get lost in the noise floor. *MT-003 Tutorial, www. analog. com/static/importedfiles/tutorials/MT-003. pdf*, 2009.

[28] J. A. Leñero-Bardallo, T. Serrano-Gotarredona, and B. Linares-Barranco. A 3.6us latency asynchronous frame-free event-driven dynamic-vision-sensor. *IEEE Journal of Solid-State Circuits*, 46(6):1443–1455, 2011.

[29] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl. Energy characterization and optimization of image sensing toward continuous mobile vision. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 69–82. ACM, 2013.

[30] V. Liu, V. Talla, and S. Gollakota. Enabling instantaneous feedback with full-duplex backscatter. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14.

[31] V. Liu, V. Talla, and S. Gollakota. Enabling instantaneous feedback with full-duplex backscatter. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 67–78. ACM, 2014.

[32] U. Mallik, M. Clapp, E. Choi, G. Cauwenberghs, and R. Etienne-Cummings. Temporal change threshold detection imager. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 362–603. IEEE, 2005.

[33] F. Michel and M. Steyaert. A 250mv 7.5 μw 61db sndr cmos sc $\delta\sigma$ modulator using a near-threshold-voltage-biased cmos inverter technique. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 476–478. IEEE, 2011.

[34] B. Murmann. Adc performance survey 1997-2017. http://web.stanford.edu/~murmann/adcsurvey.html.

[35] S. Naderiparizi, Z. Kapetanovic, and J. R. Smith. Battery-free connected machine vision with wispcam. *GetMobile: Mobile Computing and Communications*, 20(1):10–13, 2016.

[36] S. Naderiparizi, Z. Kapetanovic, and J. R. Smith. Wispcam: An rf-powered smart camera for machine vision applications. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems*, pages 19–22. ACM, 2016.

[37] S. Naderiparizi, Z. Kapetanovic, and J. R. Smith. Rf-powered, backscatter-based cameras. In *Antennas and Propagation (EUCAP), 2017 11th European Conference on*, pages 346–349. IEEE, 2017.

[38] S. Naderiparizi, A. N. Parks, Z. Kapetanovic, B. Ransford, and J. R. Smith. Wispcam: A battery-free rfid camera. In *RFID (RFID), 2015 IEEE International Conference on*, pages 166–173. IEEE, 2015.

[39] S. Naderiparizi, P. Zhang, M. Philipose, B. Priyantha, J. Liu, and D. Ganesan. Glimpse: A programmable early-discard camera architecture for continuous mobile vision. In *Proceeding of the 15th annual international conference on Mobile systems, applications, and services*. ACM, 2017.

[40] S. Naderiparizi, Y. Zhao, J. Youngquist, A. P. Sample, and J. R. Smith. Self-localizing battery-free cameras. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 445–449. ACM, 2015.

[41] S. K. Nayar, D. C. Sims, and M. Fridberg. Towards self-powered cameras. In *Computational Photography (ICCP), 2015 IEEE International Conference on*, pages 1–10. IEEE, 2015.

[42] A. N. Parks, A. Liu, S. Gollakota, and J. R. Smith. Turbocharging ambient backscatter communication. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.

[43] A. N. Parks, A. P. Sample, Y. Zhao, and J. R. Smith. A wireless sensing platform utilizing ambient rf energy. In *Biomedical Wireless Technologies, Networks, and Sensing Systems (BioWireleSS), 2013 IEEE Topical Conference on*, pages 154–156. IEEE, 2013.

[44] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, 2008.

[45] R. Sarpeshkar. Analog versus digital: extrapolating from electronics to neurobiology. *Neural computation*, 10(7):1601–1638, 1998.

[46] M. D. Scott, B. E. Boser, and K. S. Pister. An ultra–low power adc for distributed sensor networks. In *Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European*, pages 255–258. IEEE, 2002.

[47] V. Talla, M. Hessar, B. Kellogg, A. Najafi, J. R. Smith, and S. Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):105, 2017.

[48] V. Talla, B. Kellogg, S. Gollakota, and J. R. Smith. Battery-free cell phone. ACM UBIComp, 2017.

[49] V. Talla, B. Kellogg, B. Ransford, S. Naderiparizi, S. Gollakota, and J. R. Smith. Powering the next billion devices with wi-fi. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 4:1–4:13, New York, NY, USA, 2015. ACM.

[50] V. Talla and J. R. Smith. Hybrid analog-digital backscatter: A new approach for battery-free sensing. In *RFID (RFID), 2013 IEEE International Conference on*, pages 74–81. IEEE, 2013.

[51] F. Tang and A. Bermak. An 84 pw/frame per pixel current-mode cmos image sensor with energy harvesting capability. *IEEE Sensors Journal*, 12(4):720–726, 2012.

[52] A. Wang, V. Iyer, V. Talla, J. R. Smith, and S. Gollakota. Fm backscatter: Enabling connected cities and smart fabrics. *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*.

[53] P. Zhang, P. Hu, V. Pasikanti, and D. Ganesan. Ekhonet: High speed ultra low-power backscatter for next generation sensors. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 557–568. ACM, 2014.

Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs

Ravi Netravali[*], James Mickens[†]
[*]MIT CSAIL, [†]Harvard University

ABSTRACT

Web browsing on mobile devices is expensive in terms of battery drainage and bandwidth consumption. Mobile pages also frequently suffer from long load times due to high-latency cellular connections. In this paper, we introduce Prophecy, a new acceleration technology for mobile pages. Prophecy simultaneously reduces energy costs, bandwidth consumption, and page load times. In Prophecy, web servers precompute the JavaScript heap and the DOM tree for a page; when a mobile browser requests the page, the server returns a write log that contains a single write per JavaScript variable or DOM node. The mobile browser replays the writes to quickly reconstruct the final page state, eliding unnecessary intermediate computations. Prophecy's server-side component generates write logs by tracking low-level data flows between the JavaScript heap and the DOM. Using knowledge of these flows, Prophecy enables optimizations that are impossible for prior web accelerators; for example, Prophecy can generate write logs that interleave DOM construction and JavaScript heap construction, allowing interactive page elements to become functional immediately after they become visible to the mobile user. Experiments with real pages and real phones show that Prophecy reduces median page load time by 53%, energy expenditure by 36%, and bandwidth costs by 21%.

1 INTRODUCTION

Mobile browsing now generates more HTTP traffic than desktop browsing [18]. On a smartphone, 63% of user focus time, and 54% of overall CPU time, involves a web browser [56]; mobile browsing is particularly important in developing nations, where smartphones are often a user's sole access mechanism for web content [12, 23]. So, mobile page loads are important to optimize along multiple axes: bandwidth consumption, energy consumption, and page load time. Reducing bandwidth overhead allows users to browse more pages without violating data plan limits. Reducing energy consumption improves the overall lifetime of the device, because web browsing is a significant drain on battery power [8, 11, 48, 55, 56]. Improving page load time is important because users are frustrated by pages that take more than a few seconds to load [15, 17, 29, 50].

In this paper, we describe Prophecy, a new system for improving all three aspects of a mobile page load. A Prophecy web server *precomputes* much of the information that a mobile browser would generate during a traditional page load. In particular, a Prophecy server precomputes the JavaScript state and the DOM state that belongs to a loaded version of a frame. The precomputed JavaScript heap and DOM tree represent graphs of objects; however, one of Prophecy's key insights is that this state should be transmitted to clients in the form of *write logs*, not serialized graphs. At a high level, a write log contains one write operation per variable in the frame's load-time state. By returning write logs for each variable's final state, instead of returning traditional, unprocessed HTML, CSS, and JavaScript, the browser can elide slow, energy-intensive computations involving JavaScript execution and graphical layout/rendering. Conveniently, Prophecy's write logs for a frame are smaller than the frame's original content, and can be fetched in a single HTTP-level RTT. Thus, Prophecy's precomputation also decreases bandwidth consumption and the number of round trips needed to build a frame.

Earlier attempts at applying precomputation to web sites have suffered from significant practical limitations (§6), in part because these systems used serialized graphs instead of write logs. Serialized graphs hide data flows that write logs capture; analyzing these data flows is necessary to perform many optimizations. For example, Prepack [16] cannot handle DOM state, and is unable to elide computation for some kinds of common JavaScript patterns. Shandian [51] does not support caching for the majority of a page's content, does not support immediate page interactivity (§3.5), and does not work on unmodified commodity browsers; furthermore, Shandian exposes all of a user's cookies to a single proxy, raising significant privacy concerns. In contrast, Prophecy works on commodity browsers, handles both DOM and JavaScript state, preserves traditional same-origin policies about cookie security, and supports byte-granularity caching (which is *better* than HTTP's standard file-level caching scheme). Prophecy can also prioritize the loading of interactive state; this feature is important for sites that load over high-latency links, and would otherwise present users with rendered GUIs that may not actually be functional. Many of Prophecy's advantages are enabled by having fine-grained, variable-level understanding of how a page load unfolds.

Experiments with a Nexus 6 phone, loading 350 web pages on real WiFi and LTE networks, reveal Prophecy's significant benefits: median energy usage drops by 36%, median bandwidth consumption decreases by 21%, and median page load time decreases by 53% (2.8 seconds). Prophecy also helps page loads on desktop browsers, reducing median bandwidth usage by 18%, and median

page load time by 38% (0.8 seconds). These benefits are 2.2×–4.6× better than those enabled by Polaris [36], another state-of-the-art web accelerator. Thus, Prophecy represents a significant advance in web optimization.

2 Background

A single web page consists of one or more frames. Each frame is defined by an associated HTML file. The HTML describes a tree structure that connects individual HTML tags like `<title>` and `<div>`. Most kinds of tags can embed visual style attributes directly in their HTML text (e.g., `<h1 style='color:blue;'>`). However, best practice is for developers to place style information in separate CSS files, so that a frame's basic visual structure (as defined by HTML) can be defined separately from a particular styling approach for that structure.

Some tags, like `<script>` and `<img>`, support a `src` property which indicates the URL from which the tag's content should be downloaded. Alternatively, the content for the tag can be inlined. For example, a `<script>` tag can directly embed the associated JavaScript code. CSS information can also be inlined using a `<style>` tag. Inlining a tag's content eliminates an HTTP-level RTT to fetch the associated data. However, inlining prevents a browser from caching the associated object, since the browser cache interposes on explicit HTTP requests and responses, using the URL in the HTTP request as the key for storing and retrieving the associated object.

A frame uses JavaScript code to perform computation. JavaScript code is single-threaded and event-driven, with managed memory allocation; so, between the execution of event handlers, a frame's only JavaScript state resides in the managed heap. There are two kinds of JavaScript objects: application-defined and native. Application-defined objects are composed of pure JavaScript-level state. In contrast, native objects are JavaScript wrappers around native code functionality defined by the JavaScript engine, the HTML renderer, or the network engine. Examples of native objects include `RegExps` (which implement regular expressions) and `XMLHttpRequests` (which expose HTTP network connections).

DOM nodes [34] are another important type of native object. As the HTML parser scans a frame's HTML, the parser builds a native code representation of the HTML tree; this tree is reflected into the JavaScript runtime as the DOM tree. There is a 1-1 correspondence between HTML tags and DOM nodes. Using the DOM interface, JavaScript code can programmatically add, remove, or update DOM nodes, changing the visual content which is shown to a user. DOM changes often require the browser to recalculate the layout and styles of DOM nodes, and then repaint the DOM nodes. These calculations are computationally expensive (and therefore energy-intensive as well) [8, 25, 27, 56].

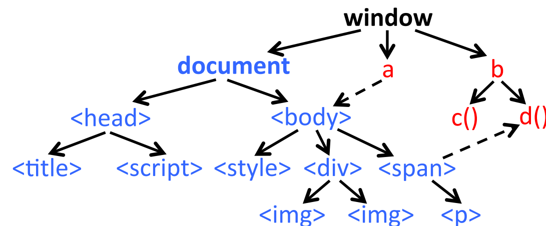


Figure 1: A simple frame's JavaScript heap and DOM tree. The JavaScript heap is red; the DOM tree is blue.

As shown in Figure 1, native code objects like the DOM tree can reference application-defined objects, and vice versa. For example, a DOM node becomes interactive via JavaScript calls like `DOMnode.addEventListener(eventType, callback)`, where `callback` is an application-defined JavaScript function which the browser will invoke upon the reception of an event.

Browsers define two main types of client-side storage. A cookie [5] is a small, per-origin file that can store up to 4 KB of data. When a browser issues an HTTP request to origin X, the browser includes any cookie that the browser stores on behalf of X. When the server receives the cookie, the server can generate personalized content for the HTTP response. The server can also use special HTTP response headers to modify the client-side cookie. Cookies are often used to hold personal user information, so cookie sharing has privacy implications.

DOM storage is the other primary type of client-side storage. DOM storage is also siloed per origin, but allows each origin to store MBs of key/value data. DOM storage can only be read and written by JavaScript code, and is separate from the browser cache (which is automatically managed by the browser itself).

3 Design

Figure 2 shows the high-level design of Prophecy. Users employ an unmodified browser to fetch and evaluate a Prophecy page. A single page consists of one or more frames; content providers who wish to accelerate their frame loads must run server-side Prophecy code that handles incoming HTTP requests for the relevant frames. The server-side Prophecy code uses a headless browser[1] to load the requested frame. The frame consists of individual objects like HTML files, JavaScript files, and images; Prophecy rewrites HTML and JavaScript before it is passed to the headless browser, injecting instru-

[1] A headless browser lacks a GUI, but otherwise performs the normal duties of a browser, parsing and rendering HTML, executing JavaScript, and so on.

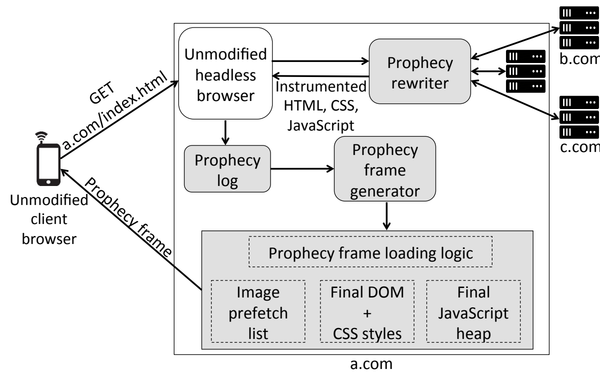


Figure 2: The Prophecy architecture.

mentation which tracks how the frame manipulates the JavaScript heap and the DOM tree.

After the headless browser has loaded the frame, Prophecy uses the resulting log to create a post-processed version of the frame. The post-processed version contains four items:

- a **write log for the JavaScript heap**, containing an ordered set of writes that a client executes to recreate the frame's final heap state;
- a **write log for the DOM**, containing HTML tags with precomputed styles, such that the client can immediately resurrect the DOM with minimal layout and rendering overheads;
- an **image prefetch log**, describing the images that the browser should fetch in parallel with the construction of the JavaScript heap and the DOM; and
- the **Prophecy resurrection library**, a small piece of JavaScript code which orchestrates client-side reconstruction of the frame (§3.2), optimizing the reconstruction for a particular load metric (§3.5).

During a warm cache load (§3.3), the three logs are diffs with respect to the client's cached logs. By applying the diffs and then executing the patched logs, a client fast-forwards its view of the frame to the latest version.

3.1 Generating a Prophecy Frame

Prophecy enables web acceleration at the granularity of a frame. However, web developers create the content for a particular frame in a Prophecy-agnostic way, using a normal workflow to determine which objects (e.g., HTML, CSS, JavaScript, and images) should belong in a frame. The process of transforming the normal frame into a Prophecy variant is handled automatically by Prophecy. The transformation can happen online (i.e., at the time of an HTTP request for the frame), or offline (i.e., before such a request has arrived). In this section, we describe the transformation process; later, we describe the tradeoffs between online and offline transformation (§3.4).

After fetching the frame's HTML, Prophecy's server-side component loads the frame in a headless browser. As the frame loads, Prophecy tracks the reads and writes that the frame makes to the JavaScript heap and to the DOM. Prophecy's design is agnostic as to how this tracking is implemented. Our concrete Prophecy prototype uses Scout [36], a frame rewriting framework, to inject logging instrumentation into the loaded frame, but Prophecy is compatible with in-browser solutions that use a modified JavaScript engine and renderer to log the necessary information. Regardless, once the frame has loaded, Prophecy analyzes the reads and writes to create the three logs which represent the Prophecy version of a frame.

The JavaScript write log: This log, expressed as a series of JavaScript statements, contains a single `lhs = rhs;` statement for each JavaScript variable that was live at the end of the frame load. The set of operations in the write log is a subset of all writes observed in the original log—only the final write to each variable in the original log is preserved. The write log first creates top-level global variables that are attached to the `window` object (see Figure 1); then, the log iteratively builds objects at greater depths from the `window` object. The final write log for the JavaScript heap does not create DOM nodes, so any JavaScript object properties that refer to DOM state are initially set to `undefined`.

The write log must pay special attention to functions. In JavaScript, a function definition can be nested within an outer function definition. The inner function becomes a closure, capturing the variable scope of the outer function. To properly handle these functions, Prophecy rewrites functions to explicitly expose their closure scope [30, 32, 51]. At frame load time on the server, this allows Prophecy's write tracking to explicitly detect which writes involve a function's closure state. Later, when a mobile browser needs to recreate a closure function, the replayed write log can simply create the function, then create the scope object, and then write to the scope object's variables.

The write log for the JavaScript heap does not contain entries for native objects that belong to the DOM tree. However, the write log does contain entries for the other native objects in a frame. For example, the log will contain entries for regular expressions (`RegExps`) and timestamps (`Dates`). Generally speaking, the write log creates native objects in the same way that it creates normal objects, i.e., by calling `lhs = new ObjClass()` and then assigning to the relevant properties via one or more statements of the form `lhs.prop = rhs`. However, Prophecy does not attempt to capture state for in-flight network requests associated with objects like `XMLHttpRequests`; instead, Prophecy waits for such connections to terminate before initiating the frame transformation process.

The DOM write log: Once Prophecy's server-side component has loaded a frame, Prophecy generates an HTML string representation for the frame using

the browser's predefined `XMLSerializer` interface. Importantly, the HTML string that is returned by `XMLSerializer` does not contain styling information for individual tags; the string merely describes the hierarchical tag structure. To extract the style information, Prophecy iterates over the DOM tree, and uses `window.getComputedStyle(domNode)` to calculate each node's style information.[2] Prophecy then augments the frame's HTML string with explicit style information for each tag. For example, a tag in the augmented HTML string might look like `<div style='border-bottom-color: rgb(255, 0, 0);border-left-color: rgb(255, 0, 0);'>`. Prophecy modifies all CSS-related tags in the augmented HTML string, deleting the bodies of inline `<style>` tags, and setting the `href` attributes in `<link rel='stylesheet'>` tags to point to the empty string (preventing a network fetch). Prophecy also modifies the `src` attribute of `<script>` tags to point to the empty string (since all JavaScript state will be resurrected using the JavaScript write log).

The augmented HTML string is the write log for the DOM, containing precomputed style information for each DOM node. Note that the style data may have been set by CSS rules, or by JavaScript code via the DOM interface. Also, some of the DOM nodes in the write log may have been dynamically created by JavaScript (instead of being statically created by the frame's original HTML). Prophecy's server-side component represents the DOM write log as a JavaScript string literal.

The image prefetch log: This log is a JavaScript array that contains the URLs for the images in the loaded frame. The associated `<img>` tags may have been statically declared in the frame's HTML, or dynamically injected via JavaScript. Note that the write log for the DOM tree contains the associated `<img>` tags; however, as we explain in Section 3.2, the image prefetch list allows the mobile browser to keep its network pipe busy as the CPU is parsing HTML and evaluating JavaScript.

The Prophecy frame consists of the three logs from above, and a small JavaScript library which uses the logs to resurrect the frame (§3.2). Since the three logs are expressed as JavaScript variables, the Prophecy server can just add those variables to the beginning of the resurrection library. So, the Prophecy frame only contains one HTML tag—a single JavaScript tag with inline content.

3.2 Loading a Prophecy Frame

A mobile browser receives the Prophecy frame as an HTTP response, and starts to execute the resurrection library. The library first issues asynchronous `Image()` requests for the URLs in the image prefetch log. As the browser fetches those images in the background, the resurrection library builds the frame in three phases.

Phase 1 (DOM Reconstruction): The resurrection library passes the DOM write log to the browser's preexisting `DOMParser` interface. `DOMParser` returns a `document` object, which is a special type of DOM node that represents an entire DOM tree. The resurrection library updates the frame's live DOM tree by splicing in the `<head>` and `<body>` DOM subtrees from the newly created `document`. After these splice operations complete, the entire DOM tree has been updated; note that the browser has avoided many of the traditional computational overheads associated with layout and rendering, since the resurrection library injected a pre-styled DOM tree which already contains the side effects of load-time JavaScript calls to the DOM interface. As the browser receives the asynchronously prefetched image data, the browser injects the pixels into the live DOM tree as normal, without assistance from the resurrection library; note that the browser will not "double-fetch" an image if, at DOM reconstruction time, the browser encounters an `<img>` tag whose prefetch is still in-flight.

Phase 2 (JavaScript Heap Reconstruction): Next, the resurrection library executes the assignments in the write log for the JavaScript heap. Each write operation is just a regular JavaScript assignment statement in the resurrection library's code. Thus, the mobile browser naturally recreates the heap as the browser executes the middle section of the library.

Phase 3 (Fixing Cross-references): At this point, the DOM tree and the JavaScript heap are largely complete. However, DOM objects can refer to JavaScript heap objects, and vice versa. For example, an application-defined JavaScript object might have a property that refers to a specific DOM node. As another example, the event handler for (say) a mouse click is an application-defined JavaScript function that must be attached to a DOM node via `DOMnode.addEventLister(evtType, func)`. In Phase 3, the resurrection library fixes these dangling references using information in the JavaScript write log. During the initial logging of reads and writes in the frame load (§3.1), Prophecy assigned a unique id to each JavaScript object and DOM node that the frame created. Now, at frame reconstruction time on the mobile browser, the resurrection library uses object ids to determine which object should be used to resolve each dangling reference. As hinted above, the library must resolve some dangling references in DOM nodes by calling specific DOM functions like `addEventListener()`. The library also needs to

[2] Prophecy uses additional logic to ensure that the extracted style information includes any default tag styles that apply to the DOM node. These default styles are not returned by `getComputedStyle()`.

invoke the relevant timer registration functions (e.g., `setTimeout(delay, callback)`) so that timers are properly resurrected.[3]

At the end of Phase 3, the frame load is complete, having skipped intermediate JavaScript computations, as well as intermediate styling and layout computations for the DOM tree. A final complication remains: what happens if, post-load, the frame dynamically injects a new DOM node into the DOM tree? Remember that Prophecy's write log for the DOM tree contains no inline `<style>` data, nor does it contain `href` attributes for `<link rel='stylesheet'>` tags (§3.1). So, as currently described, a Prophecy frame will not assign the proper styles to a dynamically created DOM node.

To avoid this problem, the resurrection library contains a string which stores all of the frame's original CSS data. The resurrection code also shims [31] DOM interfaces like `DOMnode.appendChild(c)` which are used to dynamically inject new DOM content. Upon the invocation of such a method, the Prophecy shim examines the frame's CSS rules (and the live style characteristics of the DOM tree) to apply the appropriate inline styles to the new DOM node. After applying those styles, Prophecy can safely inject the DOM node into the DOM tree.

3.3 Caching, Personalization, and Cookies

To enable frame content to be personalized, we extend the approach from Sections 3.1 and 3.2. At a high level, when a server receives an HTTP request for a frame, the server looks inside the request for a cookie that bears a customization id. If the server does not find such a cookie, then the server assumes that the mobile browser has a cold cache; in this case, the server returns the Prophecy frame as described in Section 3.1, placing a cookie in the HTTP response which describes the frame's customization id. If the server does find a customization id in the HTTP request, then the server assumes that the client possesses cached write logs for the frame. The server computes the write logs for the latest customization version of the frame. The server then calculates the diffs between the latest write logs and the ones that are cached on the phone. Finally, the server returns the diffs to the mobile browser. The mobile browser applies the diffs to the cached write logs, and then recreates the frame as described in Section 3.2.

To efficiently track the client-side versions of a frame, the server must store some metadata for each frame:

- The server stores a baseline copy of the three write logs for a frame. Denote those logs $baseline_{JS}$, $baseline_{HTML}$, and $baseline_{images}$. These logs correspond to a default version of the frame that has not been customized.
- For each version v of the frame that has been returned to a client, the server stores three diffs, namely, $diff(baseline_{JS}, customization_{v,JS})$, $diff(baseline_{HTML}, customization_{v,HTML})$, and $diff(baseline_{images}, customization_{v,images})$. The server stores these diffs in a per-frame table, using v as the key.

"Customization" has a site-specific meaning. For example, many sites return the same version of a frame to all clients during some epoch t_{start} to t_{end}. In this scenario, a new version is generated at the start of a new epoch. In contrast, if a site embeds unique, per-user content into a frame, then a version corresponds to a particular set of write logs that were sent to a particular user.

In the cold cache case, the server generates the appropriate write logs for the latest frame version v, and then diffs the latest logs against the baseline logs. The server stores the diffs in $diffTable[v]$, and then returns the latest write logs to the client as in Section 3.1, setting the customization id in the HTTP response to v. The client rebuilds the frame as in Section 3.2, and then stores the three write logs in DOM storage.

In the warm cache scenario, the server extracts v from the HTTP request, finds the associated diffs in $diffTable[v]$, and then applies the diffs to the baseline versions of the write logs. This allows the server to reconstruct the write logs that the client possesses for the old copy of v. The server then generates the write logs for the latest incarnation of v, and diffs the latest write logs against the client-side ones. The server updates $diffTable[v]$ appropriately, and then returns the diffs to the mobile browser. The mobile browser reads the cached write logs from DOM storage, applies the diffs from the server, and then rebuilds the frame using the latest write logs. Finally, the browser caches the latest write logs in DOM storage.

Note that the mobile phone and the server can get out-of-sync with respect to cache state. For example, the server might reboot or crash, and lose its per-frame *diffTables*. The user of the mobile browser could also delete the phone's DOM storage or cookies. Fortunately, desynchronization is only a performance issue, not a correctness one, because desynchronization can always be handled by falling back to the cold cache protocol. For example, suppose that a client clears its DOM storage, but does not delete its cookie. The server will send diffs, but then the client-side resurrection library will discover that no locally-resident write logs exist. The library will delete the cookie, and then refresh the page by calling `window.location.reload()`, initiating a cold cache frame load.

[3] During the instrumented frame load on the server, Prophecy shims timer registration interfaces to track timer state [31].

To minimize the storage overhead for *diffTable*, each frame's baseline should share a non-trivial amount of content with the various customized versions of the frame. Choosing good baselines is easy for sites in which, regardless of whether a user is logged in, the bulk of the site content is the same. For frames with large diffs between customized versions, and a large number of versions, servers can minimize *diffTable* overhead by breaking a single frame into multiple frames, such that highly-customized content lives in frames that are always served using the cold cache protocol (and store no server-side information in a *diffTable*). Less-customized frames can enable support for warm Prophecy caches, and communicate with the highly-dynamic frames using `postMessage()`.

For frames that enable the warm-cache protocol, the server may have to periodically update the associated baselines, to prevent diffs in *diffTable* from growing too large as the latest frame content diverges from that in the baselines. One simple pruning strategy is to generate a new baseline once the associated diffs get too large in terms of raw bytes, or as a percentage of the baseline object's size. After updating a baseline, the server must either discard the associated diffs, or recalculate them with respect to the new baseline.

3.4 Online versus Offline Transformation

Prophecy's server-side code transforms a frame's HTML, CSS, JavaScript, and images into three write logs. The transformation process can happen online or offline. In the online scenario, the server receives an HTTP request for a frame, and then loads and post-processes the frame synchronously, generating the associated write logs on-the-fly. In the offline scenario, the server periodically updates the write logs for each frame, so that, when a client requests a frame, the server already possesses the relevant write logs.

Each approach has trade-offs. Offline processing reduces the client-perceived fetch time for the frame, since the instrumented version of the regular frame does not have to be analyzed in real time. However, offline processing introduces problems of scaling and freshness if each frame has many customized versions, or those versions change frequently. A frame with many customized versions will require the server to generate and store many different sets of write log diffs, some of which may never be used if clients do not issue fetches for the associated frame versions. If versions change frequently, then the server must either frequently regenerate diffs (thereby increasing CPU overheads), or regenerate diffs less often (at the cost of returning stale versions to clients). In contrast, online processing guarantees that clients receive the latest version of a frame. Online processing also avoids wasted storage dedicated to diffs that are never fetched. A single page that contains multiple frames can use the most appropriate transformation policy for each frame.

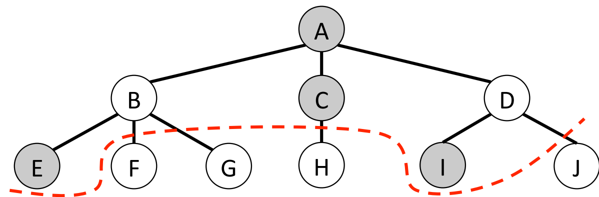


Figure 3: An example of how Prophecy determines the interactive DOM subtree to build before the rest of the DOM nodes are recreated. The shaded circles represent DOM nodes that are 1) above-the-fold, and/or 2) are manipulated by the event handlers of above-the-fold DOM nodes. The interactive subtree resides above the red line.

3.5 Defining Load Time

To fully load a traditional frame, a browser must fetch and evaluate the frame's HTML, and then fetch and evaluate the external objects that are referenced by that HTML. The standard definition for a frame's load time requires all of the external objects to be fetched and evaluated. A newer load metric, called Speed Index [21], measures how quickly a browser renders a frame's above-the-fold[4] visual content. Using write logs, Prophecy improves frame-load time (FLT) by eliding unnecessary intermediate computations and inlining all non-image content. However, as described in Section 3.2, Prophecy completely renders the DOM before constructing the JavaScript heap and then patching cross-references between the two. So, Prophecy gives higher priority to visual content, much like Speed Index (SI).

Both FLT and SI have disadvantages. FLT does not capture the notion that users desire above-the-fold content to appear quickly, even if below-the-fold content is still loading. However, at FLT time, *all* of a frame's content is ready; in contrast, SI ignores the fact that a *visible* DOM element does not become *interactive* until the element's JavaScript event handler state has been loaded. The difference between visibility and interactivity is especially apparent when a web page loads over a high-latency link; in such scenarios (which are common on mobile devices), slow-loading JavaScript can lead to `<button>` tags that do nothing when clicked, or `<input>` tags that do not offer autocompletion suggestions upon receiving user text. The median page in our test corpus had 113 event handlers for GUI interactions, so optimizing for interactivity is useful for many mobile pages.

To optimize for interactivity, Prophecy can explic-

[4]Above-the-fold content refers to the visual portion of a frame that lies within the browser GUI at the beginning of a frame load, before the user has scrolled down.

itly target a newer load metric called Ready Index [37]. Ready Index (RI) declares a frame to be ready when its above-the-fold content is both visible and interactive. To optimize for RI, Prophecy feeds its server-side log of reads and writes (§3.1) to Vesper [37]. Vesper uses load-time read/write logs, as well as read/write logs generated by active, Vesper-driven triggering of event handlers, to identify the frame's interactive state. The interactive state consists of:

- the above-the-fold DOM nodes,
- the JavaScript state which defines the event handlers for above-the-fold DOM nodes,
- the DOM state and the JavaScript state which is manipulated by those event handlers.

Given the DOM nodes in a frame's interactive state, Prophecy finds the minimal HTML subtree, rooted by the top-level `<html>` tag, which contains all of the interactive DOM nodes. Figure 3 shows an example of this interactive DOM subtree. Prophecy then represents a frame using two HTML write logs (one for the interactive subtree, and one for the remaining HTML subtrees), and two JavaScript write logs (one for the state which supports above-the-fold interactive DOM nodes, and another write log for the remaining JavaScript state). Prophecy keeps a single image prefetch log, but places above-the-fold images first in the log. To load a frame on the client browser, Prophecy first renders the above-the-fold DOM nodes, and then builds the JavaScript state which supports interactivity for those DOM nodes. After patching cross-references, the frame is interactive. Prophecy then attaches the below-the-fold DOM nodes, creates the remaining JavaScript state, and patches a final set of cross-references.

By optimizing for RI, Prophecy can minimize the likelihood that attempted user interactions will fail. However, Prophecy cannot eliminate all such problems. For example, if a user issues GUI events before the first set of write logs are applied, the events may race with the browser's creation of above-the-fold DOM elements and interactive JavaScript state. Such race conditions are present during regular, non-Prophecy page loads [40]; by optimizing for RI, Prophecy reduces the size of the race window, but does not completely eliminate it.

3.6 Privacy

A frame from origin X may embed content from a different origin Y. For example, X's frame may embed images or JavaScript from Y. When the mobile browser sends an HTTP request for X's frame, the browser will only include cookies from X, since the URL in the HTTP request has an origin of X. As Prophecy's server-side code loads the frame and generates the associated logs (§3.1), the server from X will fetch content from Y. However, in the HTTP requests that the server sends to Y, the server will not include any of Y's cookies that reside on the mobile browser—the server never received those cookies from the client. This policy amounts to a "no third-party cookie" approach. Variants of this policy are already being adopted by some browsers for privacy reasons, since third party cookies enable users to be tracked across different sites [43]. So, in Prophecy, a server from X only sees cookies that belong to X, and a frame load does not send third party cookies to any external origin Y.

3.7 Discussion

Prophecy is compatible with transport protocols like HTTP/2 [26] and QUIC [7] that pipeline HTTP requests, leverage UDP instead of TCP to transmit data, or otherwise try to optimize a browser's HTTP-level network utilization. Prophecy is also compatible with proxy-based web accelerators like compression proxies [1, 44] or split-browsers [3, 38, 39]. From the perspective of these technologies, the content in a Prophecy frame is no different than the content in a non-Prophecy frame.

Prophecy is also compatible with HTTP/2's server-push feature [7]. Server-push allows a web server to proactively send an HTTP object to a browser, pre-warming the browser's cache so that a subsequent fetch for the object can be satisfied locally. Prophecy-enabled frames use cookies to record the versions of locally-DOM-cached frames (§3.3). So, imagine that a web server would like to push frames. When the server receives an HTTP request for frame f_i, the server can inspect the cookies in the request and determine, for some different frame f_j to push, whether to push a cold-cache or warm-cache version of the frame.

A Prophecy web server does not track any information about a client's DOM storage (besides diffs for the write logs that reside in that DOM storage). Since the server does not track client-side DOM storage, the final result of a frame load should not depend on the client's non-write-log DOM storage—this state will not be available to the server-side frame load that is used to generate write logs. To the best of our knowledge, all web accelerators that use server-side load analysis [3, 38, 39, 51] assume empty client-side DOM storage, since mirroring all of that storage would be expensive, and developer best practice is to use DOM storage as a soft-state cache.

4 Implementation

On the server-side, Prophecy uses a modified version of Scout [36] to rewrite frame content and track reads and writes to the JavaScript heap and the DOM tree. Prophecy extends Scout's JavaScript translator to rewrite closure scopes (so that Prophecy can efficiently resurrect closure functions). Prophecy also extends the translator to log the classes of objects created via the `new` operator

(so that Prophecy can determine the appropriate instance objects to create in the write log for the JavaScript heap).

When rewriting a frame's HTML, Prophecy injects JavaScript source code for a timer that fires in response to the `load` event. This timer serializes the DOM as described in Section 3.1, using `XMLSerializer` to generate the basic HTML string, and using Beautiful Soup [41] to parse and edit the string, e.g., to inject precomputed CSS styles, and to extract the image `src` URLs to place in the image prefetch log.

To support client-side caching, Prophecy servers use the `google-diff-patch-match` library [20] to generate diffs. The scaffolding for Prophecy's server-side logic is implemented as a portable CGI script for Apache.

On the client-side, Prophecy's resurrection code is 1.3 KB in size. The code uses the `google-diff-patch-match` library [20] to perform diffing, and a modified version of the `CSSUtilities` framework [28] to apply styles to dynamically-created DOM nodes (§3.2).

5 Results

We evaluated Prophecy in both mobile and desktop settings. Mobile page loads were performed on a Sony Xperia X (1.8 GHz hexa-core processor) and a Nexus 6 smartphone (2.7 GHz quad core processor); each phone had 3 GB of RAM, and ran Android Nougat v7.1.1 and Chrome v61. Prophecy's performance was similar on both phones, so we only show results for the Nexus 6 device. Desktop page loads were performed on a Lenovo M91p desktop running GNU Linux 14.04. The desktop machine had 8 processors with 8 GB of RAM, and used Google Chrome v60 to load pages.

To create a reproducible test environment, we used Mahimahi [38] to record the content in the Alexa Top 350 pages [2], and later replay that content to test browsers. For pages which defined both a mobile version and a desktop version, we recorded both. Later, at experiment time, Mahimahi always returned the desktop version of a page to the desktop browser; when possible, Mahimahi returned the mobile version of a page to the mobile browser. At replay time, Mahimahi used HTTP/2 for pages that employed HTTP/2 at recording time. Server-push events that were seen at recording time were applied during replay.

The desktop machine hosted Mahimahi's replay environment. For experiments that involved desktop browsing, all web traffic was forwarded over emulated Mahimahi networks with link rates in $\{12, 25, 50\}$ Mbps and RTTs in $\{5, 10, 25\}$ ms. We observed similar trends across all of these desktop network conditions, so we only present results for the 25 Mbps link with RTTs of 10 ms.

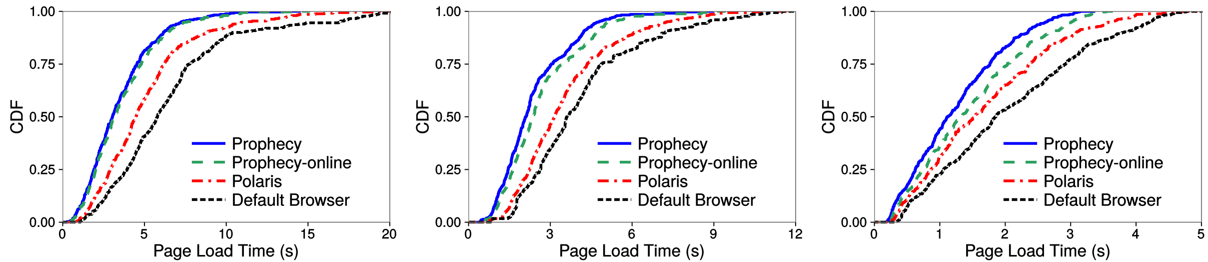
The mobile phone was connected to the desktop via both USB tethering and a live wireless connection (Verizon LTE or WiFi) with excellent signal strength. The desktop ran the test driver, initiating mobile page loads by sending commands through the USB connection. HTTP and DNS traffic between the phone and Mahimahi used the LTE or WiFi link. The live LTE connection had RTTs of roughly 75 ms, and the live WiFi connection had RTTs of roughly 15 ms.

In each of our experiments, we considered two versions of Prophecy: an *offline* version in which Prophecy frames were computed before clients requested them, and an *online* version in which the write logs were computed on-demand, in the critical path of each HTTP request (§3.4). Throughout this section, we refer to the offline version as Prophecy, and the online version as Prophecy-online. We compared the versions to default Chrome page loads, and to page loads that used Polaris [36], a state-of-the-art web accelerator. Polaris uses a client-side JavaScript library to schedule the fetching and evaluation of a page's objects. Polaris improves page load time through parallel use of the CPU and the network, and by prioritizing the fetching of objects along the dynamic critical path in a page's dependency graph. However, Polaris does not inline content or apply precomputation.

We evaluated each system on several metrics. Page load time (PLT) is the page-level equivalent of FLT (§3.5). In other words, PLT measures the time required for a browser to fetch all of the content in all of a page's frames. We evaluated Prophecy using PLT instead of FLT because PLT better captures a human's notion of a page being loaded when all of the page's frames are loaded. To measure PLT, we recorded the time between the JavaScript `navigationStart` and `onload` events. RI was computed using Vesper [37], and SI was measured using Speedline [24]. In each experiment, we loaded every page in our corpus 5 times for each system listed above, recording the median value for each load metric. Unless otherwise specified, all experiments used cold browser caches and DNS caches. In experiments with a mobile phone, energy savings were recorded by directly connecting the phone's battery leads to a Monsoon power monitor [33].

5.1 Reducing PLT

Figure 4 illustrates Prophecy's ability to reduce PLT for both mobile devices and desktop machines. Prophecy's benefits are the largest on mobile devices; for example, when using a phone to load a page over an LTE network, Prophecy reduces median PLT by 53%, and 95th percentile PLT by 67%. Prophecy helps mobile devices more for two reasons.



(a) Mobile: 4G LTE cellular network (b) Mobile: Residential WiFi network (c) Desktop: 25 Mbps link, 10 ms RTT

Figure 4: Distribution of page load times with Prophecy, Prophecy-online, Polaris, and a default browser.

- First, mobile devices suffer from higher CPU overheads for page loads, compared to desktop machines [35, 52]. So, Prophecy's elision of intermediate computation (including reflows and repaints) is more impactful on mobile devices.
- PLT is much more sensitive to network latency than to network bandwidth [1, 6, 46, 47]. Cellular links typically exhibit higher latencies than wired or WiFi links. Prophecy's aggressive use of inlining allows clients to fetch all frame content in a single HTTP-level RTT. Such RTT elision unlocks disproportionate benefits in cellular settings.

That being said, Prophecy enables impressive benefits for desktop browsers too—median PLT decreased by 38%, and 95th percentile PLT reduced by 45%.

Polaris elides no computation; in fact, client-side computational costs are slightly *higher* due to the addition of the JavaScript library which orchestrates object fetches and evaluation. Polaris also inlines no content. So, even though Polaris can keep the client's network pipe full, clients must fetch the same number of objects as in a normal page load. Since browsers limit the number of parallel HTTP requests that a page can make, Polaris generally cannot overlap all requests, leading to serial HTTP-level RTTs to build a frame. In contrast, Prophecy uses a single HTTP-level RTT to build a frame. As a result of these differences, Polaris provides fewer benefits than Prophecy. For example, on a mobile browser with an LTE connection, Polaris reduces median PLT by 23%, whereas Prophecy reduces median PLT by 53%.

As expected, PLT improvements with Prophecy-online are lower than with Prophecy, since Prophecy-online generates a frame's write logs on-demand, upon receiving a request for that frame. However, Prophecy-online still reduces median PLT by 49% on the LTE connection.

5.2 Reducing Bandwidth

Prophecy's server-side frame transformations have different impacts on the size of JavaScript state, HTML state, and image state:

- A Prophecy frame contains a write log which generates the final, precomputed JavaScript heap for the frame. The JavaScript write log is typically smaller than the frame's original JavaScript source code; although the write log must recreate the original function declarations, the log can omit intermediate function *invocations* that would incrementally create frame state.
- The HTML write log for a frame consists of an augmented HTML string that contains precomputed, inline styles for the appropriate tags. The HTML write log tends to be *larger* than a frame's original HTML string, since traditional CSS declarations can often cover multiple tags with a single CSS rule.
- The image prefetch log does not change the size of images. The log is simply a list of image URLs.

Setting	System	Bandwidth Savings (KB)
Mobile	Prophecy	262 (587)
Mobile	Polaris	-37 (-5)
Desktop	Prophecy	336 (695)
Desktop	Polaris	-41 (-12)

Table 1: Median (95th percentile) per-page bandwidth savings with Prophecy and Polaris. The baseline was the bandwidth consumed by a normal page load. The average mobile page in our test corpus was 1519 KB large; the average desktop page was 2388 KB in size.

Table 1 depicts the overall bandwidth savings that Prophecy enables; note that bandwidth savings are identical for Prophecy and Prophecy-online. Table 1 shows that Prophecy's large reductions in JavaScript size outweigh the small increases in HTML size, reducing overall bandwidth requirements by 21% in the mobile setting, and 18% in the desktop setting. In contrast, Polaris *increases* page size by a small amount. This is because a Polaris page consist of a page's original objects, plus the client-side scheduler stub and scheduler metadata.

5.3 Energy savings

Figure 5 demonstrates that Prophecy significantly reduces the energy consumed during a mobile page load. Median reductions in per-page energy usage are 36% on an LTE network, and 30% on a WiFi network. For both networks, Prophecy eliminates the same amount

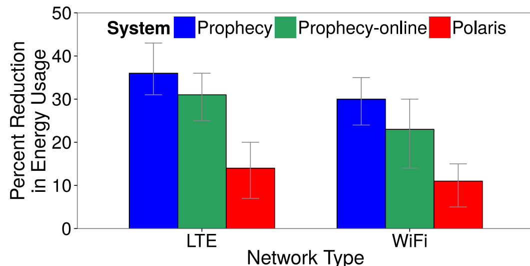


Figure 5: Percent reduction in per-page energy usage with Prophecy, Prophecy-online, and Polaris, relative to a default page load. Bars show median values, and error bars range from the 25th to the 75th percentiles. Results were collected using a Nexus 6 smartphone.

of browser computation, the same number of HTTP-level RTTs, and the same amount of HTTP-level transfer bandwidth. However, LTE hardware consumes more energy in the active state than WiFi hardware [46]; thus, reducing network traffic saves more energy on an LTE network than on a WiFi network.

Prophecy provides more energy reductions than Prophecy-online—36% versus 31% for LTE, and 30% versus 23% for WiFi. The reason is that, in Prophecy-online, server-side request handling takes longer to complete. As a result, the client-side phone must keep its network hardware active for a longer period.

Polaris reduces energy usage by 14% on the LTE network, and 10% on the WiFi network. Polaris keeps the client's network pipe full, decreasing the overall amount of time that a phone must draw down battery power to keep the network hardware on. However, because Polaris elides no computation, Polaris cannot save as much energy as Prophecy.

5.4 Reductions in SI and RI

As described in Section 3.5, SI and RI only consider the loading status of above-the-fold state. SI tracks the visual rendering of above-the-fold content, whereas RI considers both visibility and functionality. Our exploration of SI used Prophecy's default configuration. In contrast, the RI experiments used the version of Prophecy which explicitly optimizes for Ready Index (§3.5).

Speed Index: As shown in Figures 6a and 6b, Prophecy actually reduces SI more than it reduces PLT. For mobile browsing over an LTE network, the median SI reduction is 61%; for desktop browsing over a 25 Mbps link with a 10 ms RTT, the reduction is 52%. Recall that, by default, a Prophecy frame reconstructs the entire DOM tree before resurrecting the JavaScript heap (§3.1). Prioritizing DOM construction results in better SI scores, since the browser totally dedicates the CPU to rendering pre-computed HTML before replaying the JavaScript write log. As with prior experiments, the synchronous computational overheads of Prophecy-online result in slightly worse performance compared to Prophecy—57% SI reduction versus 61% in the mobile scenario, and 45% SI reduction versus 52% in the desktop setting. However, the benefits are still significant, and Prophecy-online has several advantages over Prophecy with respect to server-side overheads (§3.4).

In the mobile setting, Polaris only reduces SI by a median of 10%. In the desktop setting, Polaris actually increases SI by 2%. The reason is that Polaris' client-side scheduler is ignorant of which objects correspond to interactive state—Polaris simply tries to load all objects as quickly as possible. Reducing overall PLT is only weakly correlated with reducing SI.

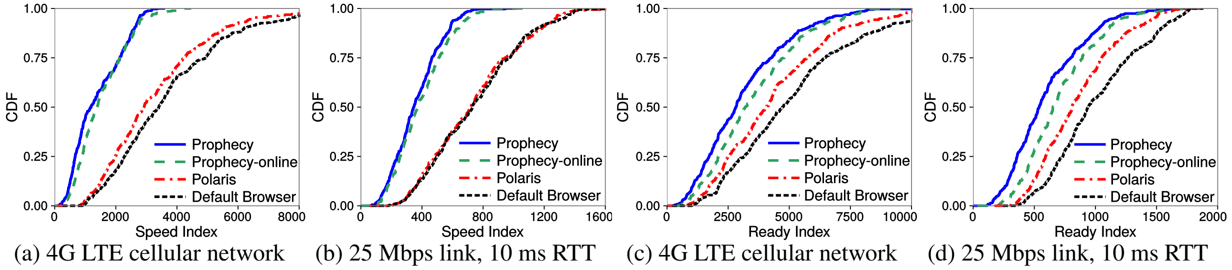
Ready Index: Figures 6c and 6d show that when Prophecy explicitly optimizes for RI (§3.5), Prophecy reduces median RI by 43% in a mobile browsing scenario, and 40% in a desktop setting. User studies indicate that, when users load a page with the expectation of interaction, optimizing for RI leads to happier users [37]. Of course, not all sites have interactive content, or a typical engagement pattern that involves immediate user input. These sites can use the standard Prophecy configuration and enjoy faster PLTs and SIs.

5.5 The Sources of Prophecy's Benefits

Prophecy optimizes a frame load in several ways:

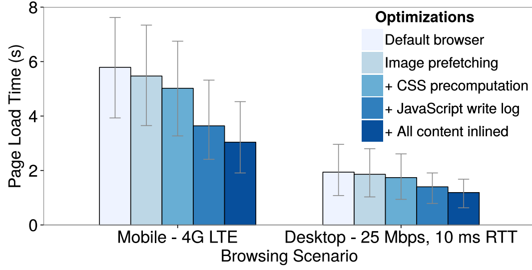
- Image prefetching: The resurrection library issues asynchronous fetches for images before constructing the DOM tree and the JavaScript heap. The asynchronous fetches keep a client's network pipe busy as the CPU works on constructing the rest of the frame.
- CSS precomputation: The DOM write log contains precomputed CSS styles for all DOM nodes, including ones that were dynamically injected by JavaScript code. Precomputation reduces client-side CPU overheads for styling, layout, and rendering.
- JavaScript write log: By only writing to each JavaScript variable once, a browser avoids wasting time and energy on unnecessary JavaScript computations.
- All content inlined: Prophecy consolidates all of the frame content into a single inlined JavaScript file which stores all of the information that is needed to rebuild the frame. Thus, a browser can fetch the entire frame in one HTTP-level round trip, as opposed to needing multiple RTTs to fetch multiple objects.

To better understand how the individual optimizations affect Prophecy's performance, we loaded each page in our corpus with a subset of the optimizations enabled. Our experiments considered mobile browsing using LTE or WiFi networks; we also tested a desktop browser with a 25 Mbps, 10 ms RTT link. In all scenarios, we used a cold cache and measured PLT. In the mobile settings, we

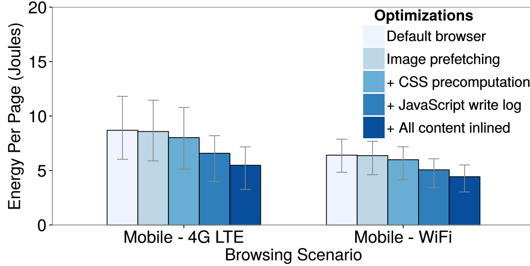


(a) 4G LTE cellular network (b) 25 Mbps link, 10 ms RTT (c) 4G LTE cellular network (d) 25 Mbps link, 10 ms RTT

Figure 6: Evaluating Prophecy using Speed Index and Ready Index.



(a) Breakdown of Prophecy's page load time reductions.



(b) Breakdown of Prophecy's energy savings.

Figure 7: Breakdown of the performance benefits enabled by individual optimizations. Optimization bars begin with image prefetching, and incrementally add new optimizations until "All content inlined," which represents Prophecy's default configuration.

also measured energy usage. Note that Prophecy's bandwidth savings are primarily from the JavaScript heap log; image prefetching and CSS precomputation do not have a significant impact on bandwidth usage (§5.2).

As shown in Figure 7, Prophecy's largest reductions in page load time and energy consumption are enabled by the JavaScript write log optimization. The next most critical optimization is content inlining; saving round trips not only reduces load time, but also reduces the amount of time that a mobile device must actively listen to the network (thereby reducing energy consumption).

5.6 Server-side overhead

When a Prophecy-online server receives a request for a frame, the server must load the requested frame and generate the necessary write logs on-demand. If the client has a warm cache, then the server must also calculate write log diffs on-demand. Figure 8 depicts the impact that these online calculations have on server response throughput. We used the Apache benchmarking tool `ab` [4] to scale client load and measure response times. The server and `ab` ran on the same machine, to isolate the computational overheads of Prophecy-online. We evaluated five server-side configurations: a default server which returned a frame's normal top-level HTML; Prophecy_cold_cache and Prophecy_cold_cache_v100000, in which clients had cold caches, and the server had either an empty *diffTable* or one that had 100,000 54 KB entries; and Prophecy_warm_cache_v100 and Prophecy_warm_cache_v100000, in which clients had warm caches and the server had the indicated number of *diffTable* entries. For warm cache experiments, we orchestrated `ab` so that all frame versions were accessed with an equal random likelihood. In all experiments, the baseline frame was the top-level frame in the `amazon.com` homepage, and the diff was an empirically-observed diff from two snapshots of the frame that were captured a day apart.

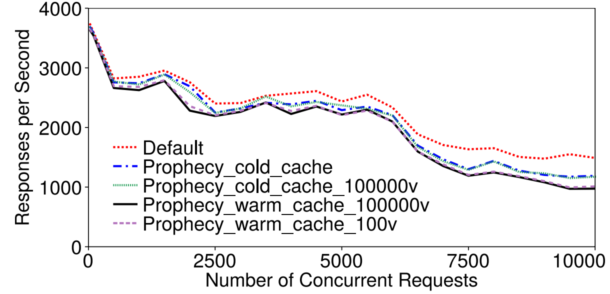
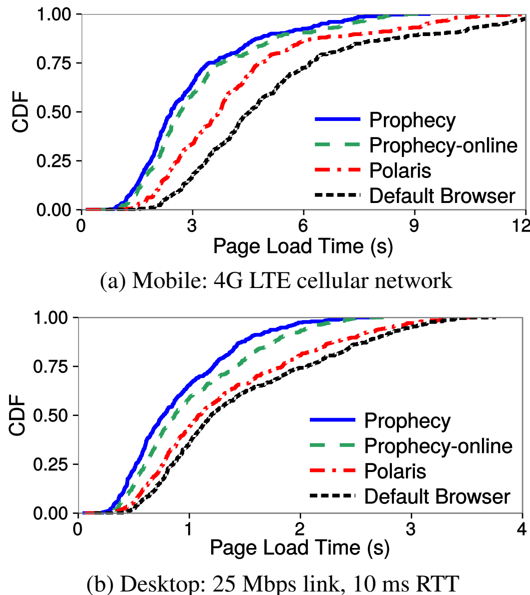


Figure 8: Prophecy-online's impact on server response throughput.

As shown in Figure 8, the performance differences grow as client load grows. Up to 6,500 concurrent requests, all server variants are within 12.1% of each other, but at 10,000 concurrent requests, the difference between the default server and the warm-cache servers is 31.4%. Performance overheads with Prophecy are mostly due to online write log generation. Also note that the CPU overhead of diffing, not the memory overhead of a *diffTable*, leads to the degraded response throughput.



(a) Mobile: 4G LTE cellular network

(b) Desktop: 25 Mbps link, 10 ms RTT

Figure 9: Distribution of warm cache page load times with Prophecy, Prophecy-online, Polaris, and a default browser. Warm cache page loads were performed 1 hour after their cold cache counterparts.

5.7 Additional Results

Due to space restrictions, we defer a full discussion of our remaining experiments to the appendix. Here, we briefly summarize the results of those experiments:

Caching: Roughly 50% of users have an empty browser cache for a particular page load [53, 54]. So, load optimizers should provide benefits if caches are warm *or* cold. The results in this section have assumed a cold cache, but Section A.1 describes the performance of Prophecy in warm cache scenarios. Unsurprisingly, Prophecy's benefits are smaller, but as shown in Figure 9, the gains are still significant, with PLT decreasing by a median of 43% in a mobile setting, and 34% in a desktop setting. Prophecy also maintains its performance advantages over Polaris with respect to SI, RI, bandwidth consumption, and energy expenditure.

Non-landing pages: Our main test corpus consisted of the landing pages for the Alexa Top 350 sites. We also tested Prophecy on 200 interior pages that were linked to by landing pages. As described in Section A.2, Prophecy performs slightly *better* on interior pages, since they tend to have more complicated structures than landing pages.

Diff sizes: We empirically analyzed snapshots of live pages, measuring how large diffs would be for clients with warm caches (Section A.3). For clients with a day-old warm cache, the median diff size was 38 KB, with a 95th percentile size of 81 KB. So, diffs are small enough for a server's *diffTable* to store many of them.

6 Related Work

6.1 Prepack

Prepack [16] is a JavaScript-to-JavaScript compiler. Prepack scans the input JavaScript code for expressions whose results are statically computable; Prepack replaces those expressions with equivalent, shorter sequences that represent the final output of the elided expressions. If JavaScript code contains dynamic interactions with the environment (e.g., via calls to `Date()`), Prepack leaves those interactions in the transformed code, so that they will be performed at runtime.

Prepack does not handle the DOM, or interactions between HTML, CSS, and JavaScript. Prepack is also unaware of important desiderata for web pages, like object cacheability and personalization (§3.3), and incremental interactivity (§3.5). Thus, Prepack is insufficiently powerful to act as a general-purpose web accelerator. Prepack's ability to elide intermediate JavaScript computations is shared by Prophecy, but Prophecy's elision is more aggressive. Prepack uses symbolic execution [10, 14] and abstract interpretation [13] to allow the results of environmental interactions to live in post-processed JavaScript as abstract values; in contrast, Prophecy evaluates all environmental interactions on the server-side, allowing all of the post-processed data to be concrete. This aggressive elision is well-suited for Prophecy's goal of minimizing client-side power usage. For example, if environmental interactions occur in a loop, Prophecy only outputs the final results, whereas Prepack often has to output an abstract, finalized-at-runtime computation for each loop iteration.

6.2 Shandian

Shandian [51] uses a proxy to accelerate page loads. The proxy uses a modified variant of Chrome to load a requested page and generate two snapshots:

- The load-time snapshot is a serialized version of (1) the page's DOM nodes and (2) the subset of the page's CSS rules that are necessary to style the DOM nodes. Importantly, the load-time snapshot does not contain any JavaScript state (although the serialized DOM nodes may contain the *effects* of DOM calls made by JavaScript).
- The post-load snapshot contains JavaScript state and the page's full set of CSS rules.

A user employs a custom Shandian browser to load the page. The browser fetches the load-time snapshot, deserializes it, and displays it. Later, the browser asynchronously fetches and evaluates the post-load snapshot.

At the architectural level, the key difference between Prophecy and Shandian is that Prophecy tracks fine-grained reads and writes during a server-side page load. Shandian does not. This design decision has cascading ramifications for performance, deployability, and robust-

ness, as described in great detail in Section A.4. For example, Shandian cannot optimize for RI; more generally, Shandian cannot interleave the resurrection of JavaScript code and the DOM tree. The reason is that Shandian lacks an understanding of how the JavaScript heap and the DOM tree interact with each other, so Shandian cannot make interleaved reconstruction safe. The specific lack of write logs for the DOM tree and the JavaScript heap also makes it difficult for Shandian to resurrect state and support caching. JavaScript is a baroque, dynamic language, and the lack of write logs forces Shandian's resurrection logic to use complex, overly conservative rules about (for example) which JavaScript statements are idempotent and which ones are not. The complicated logic requires in-browser support to get good performance, and makes caching semantics sufficiently hard to get right that Shandian does not try to support caching for load-time state (and Shandian only supports a limited form of caching for post-load state). In contrast, Prophecy's use of read/write tracking enables straightforward diff-based caching, safe interleaving of DOM construction and JavaScript resurrection, and browser agnosticism (since Prophecy's write logs are just JavaScript variables). Prophecy also enforces traditional privacy policies for cookies, unlike Shandian (§A.4).

Shandian's source code is not publicly available, and there are no public Shandian proxies. So, we could not perform an experimental comparison with Prophecy. Based on the performance numbers in the Shandian paper, we believe that Prophecy's PLT savings are roughly equivalent to those of Shandian, but Prophecy's bandwidth savings are roughly 20% better. The Shandian paper did not evaluate energy consumption, but we believe that Prophecy will consume less energy due to a simpler resurrection algorithm and less network traffic at resurrection time. Prophecy provides these benefits while enabling a constellation of important features (e.g., cacheability, optimization for interactivity) that Shandian does not provide. We refer the interested reader to Section A.4 for a more detailed discussion of Shandian.

6.3 Split browsers

In a split-browser system [38, 45, 46], a client fetches the top-level HTML in a page via a remote proxy. The proxy forwards the request to the appropriate web server. Upon receiving the response, the proxy uses a headless browser to load the page; as the proxy parses HTML, executes JavaScript, and discovers external objects in the page, the proxy fetches those objects and then forwards them to the client. Since the proxy has fast, low-latency network paths to origin servers, the time needed to resolve a page's dependency graph [11, 36] is mostly bound by proxy/origin RTTs (which are small), not the last-mile client/proxy RTTs (which may be large).

Prophecy is compatible with such approaches—a Prophecy frame can be loaded by a split-browser proxy. However, the only external objects that the proxy would discover are images, since a Prophecy frame inlines the (final effects of) external CSS and JavaScript objects. Also note that the goal of a split-browser is to hide the network latency associated with a client's object fetches; split browsers cannot identify *client-side computations* that may be elided. Prophecy does find such computations, while also eliminating fetch RTTs via inlining.

6.4 Mobile web optimizations

Klotski [9] is a mobile web optimizer that uses server-push (§3.7). When a browser fetches HTML for a particular page, the Klotski web server returns the HTML, and also pushes high-priority objects which are referenced by the page (and will later be requested by the browser). Klotski identifies high-priority objects in an offline phase using a utility function (e.g., that prioritizes above-the-fold content). Prophecy is compatible with server-push, but at the granularity of entire frames, not individual objects, since Prophecy inlines content (§3.7). Inlining, combined with final-state patching, allows Prophecy to both lower load time and decrease energy consumption. In contrast, a Klotski page elides no computation. VROOM [42] is similar to Klotski, except that clients prefetch data instead of receiving server pushes; a VROOM server uses link preload headers [22] in returned HTTP responses to hint to clients which objects can be usefully prefetched.

AMP [19] accelerates mobile page loads by requiring pages to be written in a restricted dialect of HTML, CSS, and JavaScript that is faster to load. For example, AMP forces all external `<script>` content to use the `async` attribute so that the browser's HTML parse can continue as the JavaScript code is fetched in the background. AMP forces a page to have at most one CSS file, which must be an inlined `<style>` tag whose contents are less than 50 KB in size. Prophecy is designed to support arbitrary pages that use arbitrary HTML, CSS, and JavaScript. However, Prophecy can be applied to AMP pages since those pages are just HTML, CSS, and JavaScript.

7 Conclusion

Prophecy is a new acceleration system for mobile page loads. Prophecy uses precomputation to reduce (1) the amount of state which must be transmitted to browsers, and (2) the amount of computation that browsers must perform to build the desired pages. Unlike current state-of-the-art systems for precomputation, Prophecy handles all kinds of page state, including DOM trees, and supports critical features like object caching, incremental interactivity, and cookie privacy. Experiments show that Prophecy enables substantial reductions in page load time, bandwidth usage, and energy consumption.

References

[1] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proceedings of NSDI*, 2015.

[2] Alexa. Top Sites in United States. http://www.alexa.com/topsites/countries/US, 2018.

[3] Amazon. What Is Amazon Silk? https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html, 2018.

[4] Apache Software Foundation. ab: Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2018.

[5] A. Barth. HTTP State Management Mechanism. RFC 6265. https://tools.ietf.org/html/rfc6265, April 2011.

[6] M. Belshe. More Bandwidth Doesn't Matter (Much). Google. https://goo.gl/PFDGMi, April 8, 2010.

[7] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2. RFC 7540. https://tools.ietf.org/html/rfc7540, May 2015.

[8] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao. Rethinking Energy-Performance Trade-Off in Mobile Web Page Loading. In *Proceedings of Mobicom*, 2015.

[9] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of NSDI*, 2015.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of OSDI*, 2008.

[11] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi. Deconstructing the Energy Consumption of the Mobile Page Load. In *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, New York, NY, USA, 2017. ACM.

[12] J. Chernofsky. Why emerging markets are dominating mobile browsing. The Next Web. https://thenextweb.com/insider/2016/04/07/first-world-problems-emerging-markets-dominating-mobile-browsing/, April 7, 2016.

[13] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of POPL*, 1977.

[14] P. D. Coward. Symbolic Execution Systems: A Review. *Software Engineering Journal*, 3(6):229–239, November 1988.

[15] T. Everts. New findings: For Top Ecommerce Sites, Mobile Web Performance is Wildly Inconsistent. https://blog.radware.com/applicationdelivery/wpo/2014/10/2014-mobile-ecommerce-page-speed-web-performance/, October 22, 2014.

[16] Facebook. Prepack: Partial evaluator for javascript. https://prepack.io/, 2017.

[17] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? *Journal of the Association for Information Systems*, 5(1), 2004.

[18] S. Gibbs. Mobile web browsing overtakes desktop for the first time. The Guardian. https://www.theguardian.com/technology/2016/nov/02/mobile-web-browsing-desktop-smartphones-tablets, November 2, 2016.

[19] Google. Accelerated Mobile Pages Project - AMP. https://www.ampproject.org/, 2018.

[20] Google. google-diff-match-patch. https://github.com/google/diff-match-patch, February 13, 2018.

[21] Google. Speed Index: WebPagetest Documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2018.

[22] I. Grigorik and Y. Weiss. Preload. https://www.w3.org/TR/preload/, October 26, 2017.

[23] GSMA Intelligence. Global Mobile Trends 2017. https://www.gsmaintelligence.com/research/?file=3df1b7d57b1e63a0cbc3d585feb82dc2&download, September 2017.

[24] P. Irish. Speedline. https://github.com/paulirish/speedline, November 21, 2017.

[25] P. Irish. What forces layout/reflow. https://gist.github.com/paulirish/5d52fb081b3570c81e3a, February 6 2018.

[26] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of SIGCOMM*, 2017.

[27] P. Lewis. Avoid Large, Complex Layouts and Layout Thrashing. Google Developers. https://developers.google.com/web/fundamentals/performance/rendering/avoid-large-complex-layouts-and-layout-thrashing, May 12, 2017.

[28] H. Lindqvist. CSSUtilities. http://www.brothercake.com/site/resources/scripts/cssutilities/, April 4, 2010.

[29] B. McQuade, D. Phan, and M. Vajolahi. Instant Mobile Websites: Techniques and Best Practices. Google I/O Conference presentation. http://goo.gl/DfhPJT, May 16, 2013.

[30] J. Mickens. Rivet: Browser-agnostic Remote Debugging for Web Applications. In *Proceedings of USENIX ATC*, 2012.

[31] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of NSDI*, 2010.

[32] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *Proceedings of NSDI*, 2010.

[33] Monsoon Solutions Inc. Power monitor software. http://msoon.github.io/powermonitor/, 2015.

[34] Mozilla Developer Network. Document Object Model (DOM). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model, August 29, 2017.

[35] J. Nejati and A. Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of WWW*, 2016.

[36] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of NSDI*, 2016.

[37] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring Time-to-Interactivity for Modern Web Pages. In *Proceedings of NSDI*, 2018.

[38] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of USENIX ATC*, 2015.

[39] Opera. Opera Mini. http://www.opera.com/mobile/mini, 2018.

[40] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race Detection for Web Applications. In *Proceedings of PLDI*, 2012.

[41] L. Richardson. Beautiful Soup. http://www.crummy.com/software/BeautifulSoup/, February 17, 2016.

[42] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of SIGCOMM*, 2017.

[43] S. Shankland. Ad industry attacks Safari's effort to protect your privacy. CNET. https://www.cnet.com/news/ad-industry-attacks-safaris-effort-to-protect-your-privacy/, September 15, 2017.

[44] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of Mobicom*, 2015.

[45] A. Sivakumar, C. Jiang, Y. S. Nam, P. Shankaranarayanan, V. Gopalakrishnan, S. Rao, S. Sen, M. Thottethodi, and T. Vijaykumar. NutShell: Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of MobiSys*, 2017.

[46] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. PARCEL: Proxy Assisted BRowsing in Cellular Networks for Energy and Latency Reduction. In *Proceedings of CoNEXT*, 2014.

[47] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei. Measuring and Mitigating Web Performance Bottlenecks in Broadband Access Networks. In *Proceedings of IMC*, 2013.

[48] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption. In *Proceedings of WWW*, 2012.

[49] J. Vesuna, C. Scott, M. Buettner, M. Piatek, A. Krishnamurthy, and S. Shenker. Caching Doesn't Improve Mobile Web Performance (Much). In *Proceedings of USENIX ATC*, 2016.

[50] T. Vrountas. How Slow Mobile Page Speeds Are Ruining Your Conversion Rates. https://instapage.com/blog/optimizing-mobile-page-speed, August 5, 2017.

[51] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of NSDI*, 2016.

[52] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why Are Web Browsers Slow on Smartphones? In *Proceedings of HotMobile*, 2011.

[53] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of WWW*, 2012.

[54] YUI Team. Performance Research, Part 2: Browser Cache Usage - Exposed! https://yuiblog.com/blog/2007/01/04/performance-research-part-2/, January 4, 2007.

[55] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[56] Y. Zhu and V. J. Reddi. WebCore: Architectural Support for Mobileweb Browsing. In *Proceeding of the International Symposium on Computer Architecuture (ISCA)*, 2014.

A Appendix

A.1 Warm Browser Caches

The experiments in Section 5 assumed a cold browser cache. Here, we explore the performance of Prophecy when caches are warm, finding that Prophecy still unlocks significant decreases in page load time, bandwidth consumption, and energy expenditure.

For each page in our test corpus, we used Mahimahi

Setting	System	Bandwidth Savings (KB)
Mobile	Prophecy	176 (441)
Mobile	Polaris	-37 (-5)
Desktop	Prophecy	298 (571)
Desktop	Polaris	-41 (-12)

Table 2: Median (95th percentile) per-page bandwidth savings with Prophecy and Polaris, using warm browser caches. The baseline was the bandwidth consumed by a default browser with a warm cache. The average warm cache mobile page load in our test corpus consumed 664 KB; the average desktop page used 973 KB.

Update frequency	Pages	Frames
<= 1 hour	124	313
1-2 hours	77	114
2-4 hours	41	76
4-8 hours	19	20
8-24 hours	49	109
>= 24 hours	41	229

Table 3: Update frequencies for the pages and Prophecy frames in our corpus.

to take several snapshots of the page. Each snapshot used a different time separation from the initial snapshot: no separation (i.e., a back-to-back load), 1 hour, 8 hours, and 24 hours. During an experiment which tested a particular age for a browser cache, we loaded each page twice. After clearing the browser cache, we loaded the page once using the initial snapshot. We then immediately loaded the later version of the page, recording the time of the second, warm-cache page load. Below, we discuss results for a 1 hour separation, but we observed similar trends for the other time separations.

PLT reductions: Figure 9 corroborates prior caching studies which found that mobile caching is less effective than desktop caching at reducing PLT [49]. However, Figure 9 demonstrates that Prophecy still provides substantial benefits compared to both Polaris and a default page load. For example, Prophecy enables median PLT reductions of 43% in the mobile setting, and 34% in the desktop setting. An important reason for Prophecy's persistent benefit is that, even in a warm-cache Prophecy frame (§3.3), Prophecy elides computation that must be incurred by Polaris and a default page load.

Polaris' gains drop to 15% in the mobile case, and 9% in the desktop setting. All of Polaris' benefits derive from the ability to cleverly schedule network fetches, and overlap those fetches with computation. In a warm cache scenario, a page issues fewer network requests, giving Polaris fewer opportunities for optimization.

Bandwidth savings: Table 2 demonstrates that Prophecy reduces per-page bandwidth consumption by 26% (176 KB) for mobile browsing, and 30% (298 KB) for desktop browsing. The raw savings are less than the cold cache scenarios for obvious reasons. However, since Prophecy can cache at byte granularity, not file granularity (§3.3), Prophecy downloads fewer network bytes than either Polaris or a default load.

Energy savings: Prophecy's energy savings decrease in warm cache page loads. The reason is that caching is more effective at reducing energy costs than page load time [11]; having an object cached will always avoid the battery drain associated with a network fetch, but may not decrease PLT much if the cached object is not on the critical path in the page's dependency graph [11, 36]. Regardless, Prophecy still provides substantial energy savings, reducing median and 95th percentile consumption by 17% and 29% for an LTE network. Prophecy-online's energy savings are lower than Prophecy (12% and 21%), but are higher than those of Polaris (6% and 12%).

A.2 Additional Sites

In addition to the 350 site corpus that we used for our main experiments, we also evaluated Prophecy on two additional sets of sites. First, using a web monkey, we generated a list of 200 additional pages by performing clicks on the pages in our original corpus; we generated 4 clicks per page, and then randomly selected 200 pages from the 1400 page list. These pages represented interior pages for websites, rather than the landing pages which are provided by the Alexa lists. We performed the same PLT experiments as described in Section 5.1, loading pages with a mobile phone over an LTE network. The trends were similar to those in our primary corpus. Median speedups with Prophecy increased to 57%, while Prophecy-online and Polaris accelerated PLT by 53% and 26%, respectively.

We also performed experiments with 100 randomly selected pages from the Alexa top 1000 list. The pages were chosen from the latter part of the list, such that no site was a member of our original corpus. For the new set of pages, the median PLT for a default mobile load was over *2 seconds slower* than the median PLT in our original corpus. Nevertheless, the basic trends from our main experiments persisted. Prophecy reduced the median PLT by 51%, whereas Prophecy-online and Polaris decreased PLT by 45% and 20%, respectively.

A.3 Diff Characteristics

To understand how large diffs would be in practice, we recorded 6 versions of each page in our corpus: a baseline version (at time t=0), and versions recorded at t values of 1 hour, 2 hours, 4 hours, 8 hours, and 24 hours. We then computed Prophecy frames for each version of each page. Finally, we computed diffs for each version of each frame, comparing against the baseline frame from

t=0. The server's diff calculations were fast: across all versions of the page, the median computation time was 4.6 ms, and the 95th percentile time was 8.8 ms. The median size for the largest diff across all frame versions was 38 KB; the 95th percentile largest diff size was 81 KB.

As shown in Table 3, 35% of the pages in our corpus require diff updates at least once an hour. In contrast, 12% of the pages do not require any diff updates within a single day. Similarly, some frames must be updated frequently, and some rarely change.

As a final exploration of diff behavior, we considered personalized versions of a subset of the pages in our corpus. We selected 20 pages from our corpus and created 2 different user profiles on each page. When possible, the preferences for each profile were set to different values. We then recorded three versions of each page: the default page (with no user logged in), the first user's page, and the second user's page. We created Prophecy frames for each version of each page, and compared each user's Prophecy frames to the default frames. The median diff size across all frames was 15 KB, while the maximum diff size was 31 KB. Many diffs were 0 KB, making the average diff size 6 KB.

A.4 Detailed Discussion of Shandian

In Section 6.2, we provided a high-level comparison of Shandian and Prophecy. Here, we provide more technical detail about how Shandian works, and why we believe that Prophecy's write log approach is advantageous.

Robustness: Shandian's load-time snapshot is just serialized HTML and CSS. However, Shandian's post-load snapshot cannot contain the page's unmodified JavaScript code, since client-side execution of the code would encounter a different DOM environment than what would have been seen in a normal page load. Thus, resurrecting the JavaScript state is challenging. Shandian's approach is to create a post-load snapshot which contains (1) a serialized graph of JavaScript objects minus their methods, and (2) a set of JavaScript function definitions that Shandian extracted from the page's original JavaScript code. Splicing this post-load state into the client-side environment requires complex, subtle reasoning about idempotency and ordering. For example, in a JavaScript program, a single function definition can be evaluated multiple times, with each evaluation binding to a different set of closure variables that are chosen using dynamic information. Some of the closure variables may themselves be functions. Thus, Shandian requires careful logic to generate a function evaluation order that results in the desired final state; using the lexical order of function definitions in the original source code is insufficient. Our personal experience writing JavaScript heap serializers [30, 32] has convinced us that serialization-based approaches are fragile and difficult to make correct. Prophecy's ability to track writes dramatically simplifies matters. With knowledge of the final state of each function, object, and primitive property, Prophecy can apply a straightforward three-pass algorithm to recreate an interconnected DOM tree and JavaScript heap (§3.2). Thus, we believe that a write log approach is simpler and more robust than a serialization-based approach.

Liveness: Shandian lacks a fine-grained understanding of interactions between the JavaScript heap and the DOM, so Shandian cannot safely interleave DOM construction and JavaScript evaluation. As a result, Shandian must restore all JavaScript state at once, after the DOM has been constructed. This limitation prevents Shandian from making pages incrementally interactive (§3.5). Deferring JavaScript execution has other disadvantages, like timer-based animations not starting until the associated JavaScript code has been fetched and evaluated. In contrast, Prophecy can identify related clusters of DOM nodes and JavaScript state, enabling safe, interleaved construction of a page's DOM tree and JavaScript heap. Prophecy also returns all of the page state to the client in a single HTTP round trip, unlike Shandian, which requires multiple RTTs.

Deployability: Shandian requires modified client browsers to parse Shandian's special serialization format for JavaScript state, CSS rules, and DOM state. In contrast, Prophecy logs are expressed using regular HTML and JavaScript. Thus, Prophecy works on unmodified browsers, improving deployability.

Caching: Shandian provides no caching support for the content in the initial snapshot. So, if just a single byte in the initial snapshot changes, the client must download an entirely new snapshot, spending precious energy and network bandwidth. Shandian supports caching for the post-load data, but the content in that snapshot is dependent on the content in the load-time snapshot! Thus, if the load-time snapshot changes, then cached post-load content is invalidated. In contrast, Prophecy provides a straightforward caching scheme that supports byte-level diffing (§3.3), maximizing the amount of cached content that can be used to reconstruct new versions of a page. Prophecy's caching approach is naturally suggested by Prophecy's use of write logs—these write logs are easily diffed using standard algorithms. In contrast, given Shandian's complex resurrection approach, it is not immediately clear how Shandian could be extended to support traditional caching semantics.

CSS: On the client browser, Shandian evaluates load-time CSS rules twice: once during the initial load, and again during the evaluation of a page's post-load CSS. As the Shandian paper states, the result is "additional energy consumption and latencies." We cannot quantify the costs due to lack of access to a Shandian system. Thus, we merely observe that Prophecy's inlining of CSS styles avoids *all* client-side CSS parsing for load-time DOM nodes—the associated CSS rules are evaluated *zero* times on the client. Note that, post-load, a Prophecy page can immediately style dynamically-created DOM nodes (§3.2). In contrast, Shandian will either have to wait for post-load CSS styles to be fetched (which may take a long time on a slow mobile link), or style the node immediately, but possibly incorrectly (leading to broken page state).

Privacy: In Shandian, a client-side browser ships *all* cookies, regardless of their origin, to a proxy. This scheme allows a proxy to load arbitrary personalized content on behalf of a user, but risks privacy violations if the proxy is intrinsically malicious, or becomes subverted by an external malicious party. Prophecy only exposes the cookies for origin X to servers from X.

Salsify: Low-Latency Network Video Through Tighter Integration Between a Video Codec and a Transport Protocol

Sadjad Fouladi° John Emmons° Emre Orbay°
Catherine Wu+ Riad S. Wahby° Keith Winstein°

°*Stanford University,* +*Saratoga High School*

Abstract

Salsify is a new architecture for real-time Internet video that tightly integrates a video codec and a network transport protocol, allowing it to respond quickly to changing network conditions and avoid provoking packet drops and queueing delays. To do this, Salsify optimizes the compressed length and transmission time of each frame, based on a current estimate of the network's capacity; in contrast, existing systems generally control longer-term metrics like frame rate or bit rate. Salsify's per-frame optimization strategy relies on a purely functional video codec, which Salsify uses to explore alternative encodings of each frame at different quality levels.

We developed a testbed for evaluating real-time video systems end-to-end with reproducible video content and network conditions. Salsify achieves lower video delay and, over variable network paths, higher visual quality than five existing systems: FaceTime, Hangouts, Skype, and WebRTC's reference implementation with and without scalable video coding.

1 Introduction

Real-time video has long been a popular Internet application—from the seminal schemes of the 1990s [26, 10] to today's widely used videoconferencing systems, such as FaceTime, Hangouts, Skype, and WebRTC. These applications are used for person-to-person videoconferencing, cloud video-gaming, teleoperation of robots and vehicles, and any setting where video must be encoded and sent with low latency over the network.

Today's systems generally combine two components: a transport protocol and a video codec. The transport sends compressed video to the receiver, processes acknowledgments and congestion signals, and estimates the average data rate of the network path. It supplies this estimate to the codec, a distinct module with its own internal control loop. The codec selects encoding parameters (a frame rate and quality setting) and generates a compressed video stream with an average bit rate that approximates the estimated network capacity.

In this paper, we explore and evaluate a different design for real-time Internet video, based on a video codec that is integrated tightly into the rest of the application. This system, known as Salsify, combines the transport protocol's packet-by-packet *congestion control* with the video codec's frame-by-frame *rate control* into one algorithm. This allows Salsify to avoid provoking in-network buffer overflows or queueing delays, by matching its video transmissions to the network's varying capacity.

System	Video delay 95th %ile vs. Salsify-1c (lower is better)	Video quality SSIM vs. Salsify-1c (higher is better)
Salsify-1c	[449 ms]	[15.4 dB]
FaceTime	2.3×	−2.1 dB
Hangouts	4.2×	−4.2 dB
Skype	1.2×	−6.9 dB
WebRTC	10.5×	−2.0 dB
WebRTC (VP9-SVC)	7.9×	−1.3 dB

Figure 1: Performance of Salsify (single-core version) and other real-time Internet video systems over an emulated AT&T LTE network path. Full results are in Section 5.

Salsify's video codec is implemented in a purely functional style, which lets the application explore alternative encodings of each video frame, at different quality levels, to find one whose compressed length fits within the network's instantaneous capacity. Salsify eschews an explicit bit rate or frame rate; it sends video frames when it thinks the network can accommodate them.

No individual component of Salsify is exactly new or was co-designed expressly to be part of the larger system. The compressed video format (VP8 [36]) was finalized in 2008 and has been superseded by more efficient formats in commercial videoconferencing programs (e.g., VP9 [14] and H.265 [32]). Salsify's purely functional implementation of a VP8 codec has been previously described [9], its loss-recovery strategy is related to Mosh [38, 37], and its rate-control scheme is based on Sprout [39].

Nonetheless, as a concrete system that integrates these components in a way that responds quickly to network variation without provoking packet loss or queueing delay, Salsify outperforms the commercial state of the art—Skype, FaceTime, Hangouts, and the WebRTC implementation in Google Chrome, with or without scalable video coding—in terms of end-to-end video quality and delay (Figure 1 gives a preview of results).

These results suggest that improvements to video *codecs* may have reached the point of diminishing returns in this setting, but changes to the architecture of video *systems* can still yield significant benefit. In separating the codec and transport protocol at arm's length—each with its own rate-control algorithm working separately—today's applications reflect current engineering practice in which codecs are treated largely as black boxes, at the cost of a significant performance penalty. Salsify demonstrates a way to integrate these components more tightly, while preserving an abstraction boundary between them.

This paper proceeds as follows. Section 2 discusses background information on real-time video systems and related work. We describe the design and implementation of Salsify in Section 3 and of our measurement testbed for black-box video systems in Section 4. Section 5 presents the results of the evaluation. We discuss the limitations of the system and its evaluation in Section 6.

We also performed two user studies to estimate the relative importance of quality and delay on the subjective quality of experience (QoE) of real-time video systems. These results are described more fully in Appendix A.

Salsify is open-source software, and the experiments reported in this paper are intended to be reproducible. The source code and raw data from the evaluation are available at https://snr.stanford.edu/salsify.

2 Related work

Adaptive videoconferencing. Skype and similar programs perform adaptive real-time videoconferencing over an Internet path, by sending user datagrams (UDP) that contain compressed video. In addition to Skype, such systems include FaceTime, Hangouts, and the WebRTC system, currently in development to become an Internet standard [1]. WebRTC's reference implementation [35] has been incorporated into major Web browsers.

These systems generally include a video codec and transport protocol as independent subsystems, each with its own rate-control logic and control loop. The transport provides the codec with estimates of the network's data rate, and the video encoder selects parameters (including a frame rate and bit rate) to match its average bit rate to the network's data rate. Salsify, by contrast, merges the rate-control algorithm of each component into one, leveraging the functional nature of the video codec to keep the length of each compressed frame within the transport's instantaneous estimate of the network capacity.

Joint source-channel video coding. The IEEE multimedia communities have extensively studied low-latency real-time video transmission over digital packet networks (a survey is available in Zhai & Katsaggelos [45]). The bulk of this work targets heavily-multiplexed networks, where data rates are treated as fixed or slowly varying, and packet loss and queueing delay can be modeled as random processes independent of the application's own behavior (e.g., [5]). In this context, prior work has focused on combining *source coding* (video compression) with *channel coding* (forward error correction) in order for the application to gracefully survive packet drops and delays caused by independent random processes [45].

Salsify is aimed at a different regime, more typical of today's Internet access networks, where packet drops and queueing delays are influenced by how much data the application chooses to send [39, 13], the bottleneck data rate can decay quickly, and forward-error-correction schemes are less effective in the face of bursty packet losses [33]. Salsify's main contribution is not in combining video coding and error-correction coding to weather packet drops that occur independently; it is in merging the rate-control algorithms in the video codec and transport protocol to avoid provoking packet drops (e.g., by overflowing router buffers) and queueing delay with its own traffic.

Cross-layer schemes. Schemes like SoftCast [19] and Apex [30] reach into the physical layer, by sending analog wireless signals structured so that video quality degrades gracefully when there is more noise or interference on the wireless link. Much of the IEEE literature [45] also concerns regimes where modulation modes and power levels are under the application's control. Salsify is also designed to degrade gracefully when the network deteriorates, but Salsify isn't a cross-layer scheme in the same way—it does not reach into the physical layer. Like Skype, FaceTime, etc., Salsify sends conventional UDP datagrams over the Internet.

Low-latency transport protocols. Prior work has designed several transport protocols and capacity-estimation schemes for real-time applications [21, 18, 39, 17, 6]. These schemes are often evaluated with the assumption that the application always has data available to the transport, allowing it to run "full throttle"; e.g., Sprout's evaluation made this assumption [39]. In the case of video encoders that produce frames intermittently at a particular frame rate and bit rate, this assumption has been criticized as unrealistic [16]. Salsify's transport protocol is based on Sprout-EWMA [39], but enhanced to be video-aware: the capacity-estimation scheme accounts for the intermittent (frame-by-frame) data generated by the video codec.

Scalable or layered video coding. Several video formats support scalable encoding, where the encoder produces multiple streams of compressed video: a base layer, followed by one or more enhancement layers that improve the quality of the lower layers in terms of frame rate, resolution, or visual quality. Scalable coding is part of the H.262 (MPEG-2), MPEG-4 part 2, H.264 (MPEG-4 part 10 AVC) and VP9 systems. A real-time video application may use scalable video coding to improve performance over a variable network, because the application can dis-

card enhancement layers immediately in the event of congestion, without waiting for the video codec to adapt. (Improvements in quality, however, must wait for a coded enhancement opportunity.) Scalability is particularly useful in multiparty videoconferences, because it allows a relay node to adapt a sender's video stream to different receiver network capacities by discarding enhancement layers, without re-encoding. Salsify is aimed at unicast situations; in this setting, we evaluated a contemporary SVC system, VP9-SVC as part of WebRTC in Google Chrome, and found that it did not improve markedly over conventional WebRTC.

Measurement of real-time video systems. Prior work has evaluated the performance of integrated videoconferencing applications. Zhang and colleagues [46] varied the characteristics of an emulated network path and measured how Skype varied its network throughput and video frame rate. Xu and colleagues [41] used Skype and Hangouts to film a stopwatch application on the receiver computer's display, producing two clocks side-by-side on the receiver's screen to measure the one-way video delay.

Salsify complements this literature with an end-to-end measurement of videoconferencing systems' video quality as well as delay. From the perspective of the sending computer, the testbed appears to be a USB webcam that captures a repeatable video clip. On the receiving computer, the HDMI display output is routed back to the testbed. The system measures the end-to-end video quality and delay of every frame.

QoE-driven video transport. Recent work has focused on optimization approaches to delivery of adaptive video. Techniques include control-theoretic selection of pre-encoded video chunks for a Netflix-like application [44] and inferential approaches to selecting relays and routings [20, 12]. Generally speaking, these systems attempt to evaluate or maximize performance according to a function that maps various metrics into a single quality of experience (QoE) figure. Our evaluation includes two user studies to calibrate a QoE metric and find the relative impact of video delay and visual quality on quality of experience in real-time video applications (a videochat and a driving-simulation videogame).

Loss recovery. Existing systems use several techniques to recover from packet loss. RTP and WebRTC applications sometimes retransmit the lost packet, and sometimes re-encode missing slices of a video frame *de novo* [28, 25]. By contrast, Salsify's functional video decoder retains old states in memory until the sender gives permission to evict them. If a network has exhibited recent packet loss, the encoder can start encoding new frames in a way that depends only on an older state that the receiver has acknowledged, allowing the frame to be decoded even if intervening packets turn out to have been lost. This approach has been described as "prophylactic retransmission" [37].

3 Design and Implementation

Real-time Internet video systems are built by combining two components: a transport protocol and a video codec. In existing systems, these components operate independently, occasionally communicating through a standardized interface. For example, in WebRTC's open-source reference implementation, the video encoder reads frames off the camera at a particular frame rate and compresses them, aiming for a particular average bit rate. The transport protocol [17] updates the encoder's frame rate and target bit rate on a roughly one-second timescale. WebRTC's congestion response is generally reactive: if the video codec produces a compressed frame that overshoots the network's capacity, the transport will send it (even though it will cause packet loss or bloated buffers), but the WebRTC transport subsequently tells the codec to pause encoding new frames until congestion clears. Skype, FaceTime, and Hangouts work similarly.

Salsify's architecture is more closely coupled. Instead of allowing the video codec to free-run at a particular frame rate and target bit rate, Salsify fuses the video codec's and transport protocol's control loops into one. This architecture allows the transport protocol to communicate network conditions to the video codec before each frame is compressed, so that Salsify's transmissions match the network's evolving capacity, and frames are encoded when the network can accommodate them.

Salsify achieves this by exploiting its codec's ability to save and restore its internal state. Salsify's transport estimates the number of bytes that the network can safely accept without dropping or excessively queueing frames. Even if this number is known before encoding begins for each frame, it is challenging to predict the encoder parameters (quality settings) that cause a video encoder to match a pre-specified frame length.

Instead, each time a frame is needed, Salsify tries encoding with two different sets of encoder parameters in order to bracket the available capacity. The system examines the encoded sizes of the resulting compressed frames and selects one to send, based on which more closely matches the network capacity estimate. The state induced by this frame is then restored and used as the basis for both versions of the next coded frame. We implemented two versions of Salsify: one that does the two encodings serially on one core (Salsify-1c), and one in parallel on two cores (Salsify-2c).

3.1 Salsify's functional video codec

Salsify's video codec is written in about 11,000 lines of C++ and encodes/decodes video in Google's VP8 format [36]. It differs from previous implementations of VP8 and other codecs in one key aspect: it exposes the internal

"state" of its encoder/decoder to the application in explicit state-passing style (We previously described an earlier version of this codec in ExCamera [9].)

The state includes copies of previous decoded frames, known as reference images, and probability tables used for entropy coding. At a resolution of 1280×720, the internal state of a VP8 decoder is about 4 MiB. To compress a new image, the video encoder takes advantage of its similarities with the reference images in the state. The video decoder can be modeled as an automaton, with coded frames as the inputs that cause state transitions between a source and target state. The automaton starts in the source state, consumes the compressed frame, outputs an image for display, and transitions to the target state.

In typical implementations, whether hardware or software, this state is maintained internally by the encoder/decoder and is inaccessible to the application. The encoder ingests a stream of raw images as the input, and produces a compressed bitstream. When a frame is encoded, the internal state of the encoder changes and there is no way to undo the operation and return to the previous state. Salsify's VP8 encoder and decoder, by contrast, are pure functions with no side effects and all state maintained externally. The interface is:

$$\textbf{decode}(\texttt{state}, \texttt{frame}) \rightarrow (\texttt{state}', \texttt{image})$$
$$\textbf{encode}(\texttt{state}, \texttt{image}, \texttt{quality}) \rightarrow \texttt{frame}$$

Using this interface, the application can explore different quality options for encoding each frame and start decoding from a desired state. This allows Salsify to (1) encode frames at a size and quality that matches the network capacity and (2) efficiently recover from packet loss.

Encoding to match network capacity. Current video encoders, including Salsify's, are unable to compress a single frame to accurately match a specified coded length. Only after compressing a frame does the encoder discover the resulting length with any precision. As a result, current videoconferencing systems generally track the network's capacity in an average sense, by matching the encoder's average bit rate over time to the network's data rate.

Salsify, by contrast, exploits the functional nature of its video encoder to optimize the compressed length of each individual frame, based on a current estimate of the network's capacity. Two compression levels are explored for each frame by running two encoders (serially or in parallel), initialized with the same internal state but with different quality settings. Salsify selects the resulting frame that best matches the network conditions, delaying the decision of which version to send until as late as possible, after knowing the exact size of the compressed frames. Since the encoder is implemented in explicit state-passing style, it can be resynchronized to the state induced by whichever version of the frame is chosen to be transmitted. Salsify chooses the two quality settings for the next frame based on surrounding (one higher, one lower) whichever settings were successful in the previous frame.

There is also a third choice: not to send *either* version, if both exceed the estimated network capacity. In this case, the next frame will be encoded based on the same internal state. Salsify is therefore able to vary the frame cadence to accommodate the network, by skipping frames in a way that other video applications cannot (conventional applications can only pause frames on *input* to the encoder—they cannot skip a frame after it has been encoded without causing corruption).

Loss recovery. Salsify's loss recovery strategy condenses into picking the *right* source state for encoding frames. In the absence of loss, the encoder produces a sequence of compressed frames, each one depending on the target state resulting from the previous frame, even if that frame has not yet been acknowledged as received—the sender assumes that all the packets in flight will be delivered. Packet loss, however, causes incomplete or missing frames at the receiver, putting its decoder in a different state than the one assumed by the sender and corrupting the decoded video stream. To remedy this, when the sender detects packet loss (via ACKs from the receiver, § 3.2), it resynchronizes the receiver's state by creating frames that depend on a state that the receiver has explicitly acknowledged (Algorithm 1.3 and Algorithm 2.3); these frames will be usable by the receiver, even if intermediate packets are lost. This approach requires the sender and receiver to save the sequence of target states in memory, only deleting them when safe. Specifically, upon receiving a frame based on some state, the receiver discards all older ones; and when the sender receives an ACK for some state, it discards all older ones.

In case of packet reordering, if a fragment for a new frame is received before the current frame is complete, the receiver still decodes the incomplete frame—which puts its decoder in an invalid state—and moves on the next frames. The sender recognizes this situation as packet loss and handles it the same way. Packet reordering within the fragments of a frame is not disruptive, as the receiver first waits for all the fragments before reassembling and decoding a frame.

3.2 Salsify's transport protocol

We implemented Salsify's transport protocol in about 2,000 lines of C++. The sender transmits compressed video frames over UDP to the receiver, which replies with acknowledgments. Each video frame is divided into one or more MTU-sized fragments.

Other than the frame serial number and fragment index, each frame's header contains the hash of its source state, and the hash of its target state. With these headers, a compressed video frame becomes an idempotent operator that

Algorithm 1 Salsify transport protocol

```
1: procedure ON-RECEIVING-ACK(ack)
2:     set to values indicated by ack:
         mean_interarrival_time,
         known_receiver_codec_state,
         num_packets_outstanding
3:     if ack indicates loss then
4:         /* enter loss recovery mode for next 5 seconds */
5:     end if
6:     max_frame_size ← MTU × (100 ms /
         mean_interarrival_time −
         num_packets_outstanding)
7: end procedure
```

acts on the identified source state at the receiver, transforming it into the identified target state, and producing a picture for display in the process. The receiver stores the target state in memory, in the case that the sender wants to use that state for loss recovery. In reply to each fragment, the receiver sends an acknowledgment message that contains the frame number and the fragment index, along with its current decoder hash.

The receiver treats the incoming packets as a packet train [21, 18] to probe the network and maintains a moving average of packet inter-arrival times, similar to WebRTC [17, 6] and Sprout-EWMA [39]. This estimate is communicated to the sender in the acknowledgment packets. However, the sender does not transmit continuously—it pauses between frames. As a result, the inter-arrival time between the last fragment of one frame and the first fragment of the next frame is not as helpful an indicator of the network capacity (Figure 2). This pause could give the receiver an artificially pessimistic estimate of the network because the application is not transmitting "full throttle." To account for this, the sender includes a *grace period* in each fragment, which tells the receiver about the duration between when the current and previous fragments were sent. As fragment i is received, the receiver calculates the smoothed inter-arrival time, τ_i, as

$$\tau_i \leftarrow \alpha(T_i - T_{i-1} - \text{grace-period}_i) + (1-\alpha)\tau_{i-1},$$

where T_i is the time fragment i is received. The value of α is 0.1 in our implementation, approximating a moving average over the last ten arrivals.

At the sender side, Salsify's transport protocol estimates the desired size of a frame based on the latest average inter-arrival time reported by the receiver. To calculate the target size at time step i, the sender first estimates an upper bound for the number of packets already in-flight, N_i, by subtracting the indices of the last-sent packet and the last-acknowledged packet. Let τ_i be the latest average inter-arrival time reported by the receiver at i. If the sender aims to keep the end-to-end delay less than d (set to 100 ms in our implementation) to preserve interactivity, there can be no more than d/τ_i packets in flight. Therefore, the target size is $(d/\tau_i - N_i)$ MTU-size fragments (Algorithm 1.6). At the time of sending, the sender will pick the largest frame that doesn't exceed this length. If both encoded versions are over this size, the sender discards the frame and moves on to sending the next frame. To be able to receive new feedback from the receiver, if more than four frames are skipped in a row, the sender sends the low quality version (Algorithm 2).

Algorithm 2 Salsify sender program

```
 1: procedure SEND-NEXT-FRAME
 2:     image ← NEXT-IMAGE-FROM-CAMERA
 3:     source_state ← loss_recovery_mode
          ? known_receiver_codec_state
          : last_sent_frame.target_codec_state

 4:     frame_lower_quality ← ENCODE(
          source_state, image,
          last_sent_frame.quality − DECR)
 5:     frame_higher_quality ← ENCODE(
          source_state, image,
          last_sent_frame.quality + INCR)

 6:     frame_to_send ← SELECT-FRAME(
          frame_lower_quality,
          frame_higher_quality)
 7:     if frame_to_send ≠ null then
 8:         SEND(frame_to_send)
 9:         consecutive_skip_count ← 0
10:         last_sent_frame ← frame_to_send
11:     end if
12: end procedure
13:
14: function SELECT-FRAME(lower, higher)
15:     if higher.length < max_frame_size then
16:         return higher
17:     else if lower.length < max_frame_size or
          consecutive_skip_count ≥ 4 then
18:         return lower
19:     else
20:         consecutive_skip_count++
21:         return null
22:     end if
23: end function
```

4 Measurement testbed

To evaluate Salsify, we built an end-to-end measurement testbed for real-time video systems that treats the sender and receiver as black boxes, emulating a time-varying network while measuring application-level video metrics that affect quality of experience. This section describes the testbed's metrics and requirements (§4.1), its design (§4.2), and its implementation (§4.3).

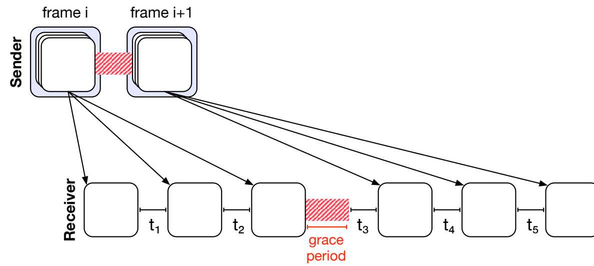


Figure 2: The receiver maintains a moving average of packet inter-arrival times, t_is. The sender includes the delay between sent packets as a "grace period," so the receiver can account for the sender's pauses between frames.

4.1 Requirements and metrics

Requirements. The testbed needs to present itself as a webcam and supply a high-definition, 60 fps video clip in a repeatable fashion to unmodified real-time video systems. At the same time, the testbed needs to emulate a varying network link between the sender and receiver in the system, with the time-varying behavior of the emulated network synchronized to the test video. Finally, the testbed needs to capture frames coming out of the display of an unmodified receiver, and quantify their quality (relative to the source video) and delay.

Metrics. The measurement testbed uses two principal metrics for evaluating the video quality and video delay of a real-time video system. For quality, we use mean *structural similarity* (SSIM) [34], a standard measure that compares the received frame to the source video.

To measure interactive video delay, the testbed calculates the difference between the time that it supplies a frame (acting as a webcam) and when the receiver displays the same frame (less the testbed's inherent delay, which we measure in §5.1).

For frames on the 60 fps webcam input that weren't sent or weren't displayed by the receiver, we assign an arrival time equal to the *next* frame shown. As a result, the delay metric rewards systems that transmit with a higher frame rate. The goal of this metric is to account for both the frame rate chosen by a system, and the delay of the frames it chooses to transmit. A system that transmits one frame per hour, but those frames always arrive immediately, will still be measured as having delay of up to an hour, even though the rare frame that *does* get transmitted arrives quickly. A system that transmits at 60 frames per second, but on a one-hour tape delay, will also be represented as having a large delay.

4.2 Design

Figure 3 outlines the testbed's hardware arrangement. At a high level, the testbed works by injecting video into a sending client, simulating network conditions between sender and receiver, and capturing the displayed video at the receiving client. It then matches up frames injected into the sender with frames captured from the receiver, and computes the delay and quality.

Hardware. The sender and receiver are two computers running an unaltered version of the real-time video application under test. Each endpoint's video interface to the testbed is a standard interface: For the sender, the testbed emulates a UVC webcam device. For the receiver, the testbed captures HDMI video output.

The measurement testbed also controls the network connection between the sender and receiver. Each endpoint has an Ethernet connection to the testbed, which bridges the endpoints to each other and to the Internet.

Video analysis. To compute video-related metrics, the testbed logs the times when the sending machine is presented with each frame, captures the display output from the receiver, and timestamps each arriving frame in hardware to the same clock.

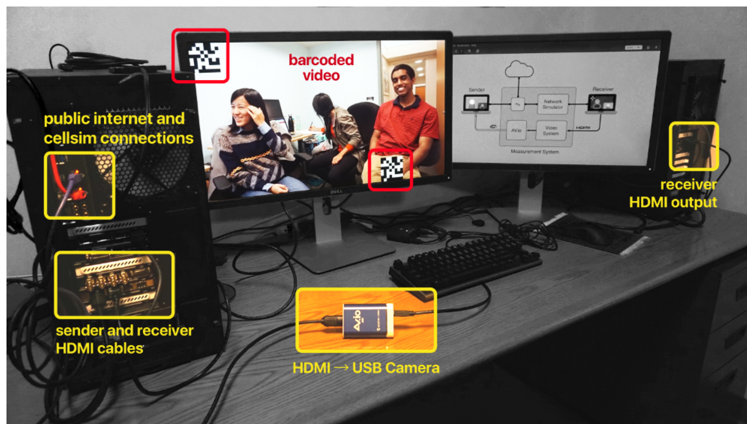
The testbed matches each frame captured from the receiver to a frame injected at the sender. To do so, the testbed preprocesses the video to add two small barcodes, in the upper-left and lower-right of each frame.[1] Together, the barcodes consume 3.6% of the frame area. Each barcode encodes a 64-bit random number that is unique over the course of the video. An example frame is shown in figures 3 and 4. The quality and delay metrics are computed in postprocessing by matching the barcodes on sent and received frames, then comparing corresponding frames.

4.3 Implementation

The measurement testbed is a PC workstation with specialized hardware. To capture and play raw video, the system uses a Blackmagic Design DeckLink 4K Extreme 12G card, which emits and captures HDMI video. The DeckLink timestamps incoming and outgoing frames with its own hardware clock. To convert outgoing video to the UVC webcam interface, the testbed uses an Epiphan AV.io HDMI-to-UVC converter. At a resolution of 1280×720 and 60 frames per second, raw video consumes 1.8 gigabits per second. The testbed uses two SSDs to simultaneously play back and capture raw video.

The measurement testbed computes SSIM using the Xiph Daala tools package. For network emulation, we use Cellsim [39], always starting the program synchronized to the beginning of an experiment. To explore the sensitivity to queueing behavior in the network, we configured Cellsim with a DropTail queue with a dropping threshold of 64, 256, or 1024 packets; ultimately we found applications were not sensitive to this parameter and conducted

[1] The two barcodes were designed to detect tearing within a frame, when the receiver displays pieces of two different source frames at the same time. In our evaluations, we did not see this occur.



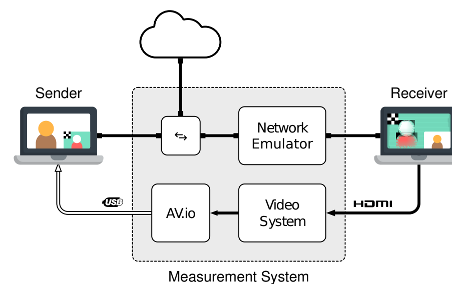


Figure 3: Testbed architecture. The system measures the performance of an unmodified real-time video system. It emulates a webcam to supply a barcoded video clip to the sender. The sender transmits frames to the receiver via an Ethernet connection. The measurement testbed interposes on the receiver's network connection and controls network conditions using a network emulator synchronized to the video. The receiver displays its output in a fullscreen window via HDMI, which the testbed captures. By matching barcodes on sent and received frames, the testbed measures the video's delay and quality, relative to the source. The measurement testbed timestamps outgoing and incoming frames with a dedicated hardware clock, eliminating the effect of scheduling jitter in measuring the timing of 60 fps video frames.

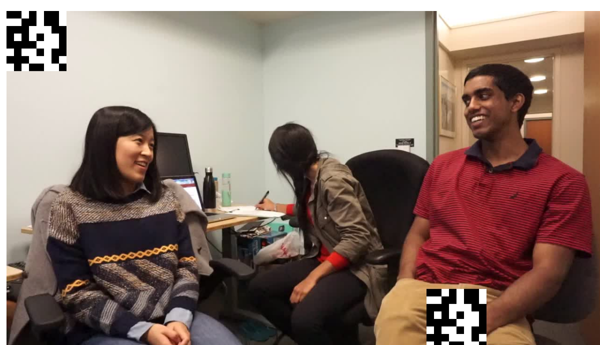


Figure 4: An example barcoded video frame sent by the measurement testbed (§4.2). The barcodes each represent a 64-bit random number that is unique over the course of the video.

remaining tests with a 256-packet buffer. The round-trip delay was set to 40 ms for cellular traces. We developed new software for barcoding, playing, capturing, and analyzing video. It comprises about 2,500 lines of C++.

5 Evaluation of Salsify

This evaluation answers the question: how does Salsify compare with five popular real-time video systems in terms of video delay and video quality when running over a variety of real-world and synthetic network traces? In sum, we find that, among the systems tested, Salsify gave the best delay and quality by substantial margins over a range of cellular traces; Salsify also performed competitively on a synthetic intermittent link and an emulated Wi-Fi link.

5.1 Setup, calibration, and method

Setup. We ran all experiments using the measurement testbed described in Section 4. Figure 5 lists applications and versions. Tests on macOS used late-model MacBook Pro laptops running macOS Sierra. WebRTC (VP9-SVC) was run on Chrome with command line arguments to enable VP9 scalable video coding; the arguments were suggested by video-compression engineers on the Chrome team at Google.[2] Tests on Linux used Ubuntu 16.10 on desktop computers with recent Intel Xeon E3-1240v5 processors and 32 GiB of RAM. We tested Salsify using the same Linux machine.

All machines were physically located in the same room during experiments and were connected to each other and the public Internet through gigabit Ethernet connections. Care was taken to ensure that no other compute- or network-intensive processes were running on any of the client machines while experiments were being performed.

Calibration. To calibrate the measurement testbed, we ran a loopback experiment with no network: we connected the testbed's UVC output to the desktop computer described above, configured that computer to display incoming frames fullscreen on its own HDMI output using `ffplay`, and connected that output back to the testbed.

We found that the delay through the loopback connection was 4 frames, or about 67 ms; in all further experiments we subtracted this intrinsic delay from the raw results. The difference between the output and input images was negligible, with SSIM in excess of 25 dB, which

[2] The arguments were: out/Release/chrome --enable-webrtc-vp9-svc-2sl-3tl --fake-variations-channel=canary --variations-server-url=https://clients4.google.com/chrome-variations/seed .

Application	Platform	Version	Configuration change
Skype	macOS	7.42	Turned off Skype logo on the receiver.
FaceTime	macOS	3.0	Blacked out self view in post-processing.
Hangouts	Chrome (Linux)	Chrome 55.0 Chrome 62.0 (Figure 6e)	Edited CSS to hide self view.
WebRTC	Chrome (Linux)	Chrome 62.0, https://appr.tc Chrome 55.0 (Figure 7) Chrome 65.0 (Figure 8)	Edited CSS to hide self view.
WebRTC (VP9-SVC)	Chrome (Linux)	Chrome 62.0, https://appr.tc	Edited CSS to hide self view.

Figure 5: Application versions tested. For each application, we slightly modified the receiver to eliminate extraneous display elements that would have interfered with SSIM calculations. For WebRTC (VP9-SVC), we passed command-line arguments to Chrome to enable the scalable video codec.

corresponds to 99.7% absolute similarity.

Method. For each experiment below, we evaluate each system on the testbed using a specified network trace, computing metrics as described in Section 4.1. The stimulus is a ten minute, 60 fps, 1280×720 video of three people having a typical videoconference call. We preprocessed this video as described in Section 4.2, labeling each frame with a barcode. The network traces are long enough to cover the whole length of the video.

5.2 Results

Experiment 1: variable cellular paths. In this experiment, we measured Salsify and the other systems using the AT&T LTE, T-Mobile UMTS ("3G"), and Verizon LTE cellular network traces distributed with the Mahimahi network-emulation tool [27]. The experiment's duration is 10 minutes. The cellular traces vary through a large range of network capacities over time: from more than 20 Mbps to less than 100 kbps. The AT&T LTE and T-Mobile traces were held out and not used in Salsify's development, although an earlier (8-core) version of Salsify was previously evaluated on these traces before we developed the current 1-core and 2-core versions.

Figures 6a, 6b and 6c show the results for each scheme on each trace. Both the single-core (Salsify-1c) and dual-core (Salsify-2c) versions of Salsify outperform all of the competing schemes on both quality and delay (and therefore, on either QoE model from the user study). We saw little difference in performance between the serial and parallel versions of Salsify; this suggests that having the two video encoders run *in parallel* is not very important on the PC workstation tested.

Salsify's loss-recovery strategy requires the sender and receiver keep a set of decoder states in memory, in order to recover from a known state if loss occurs. After the receiver acknowledges a state, or the sender sends a frame based on a state, the other program discards the older states. In our AT&T LTE trace experiment, Salsify-2c sender kept 6 states on average, each 4 MiB in size, during the course of its execution, while the receiver kept 3 states at a time on average. Additionally, in the same experiment, Salsify-2c picked the better-quality option in 50%, the lower-quality option in 32%, and not sending the frame at all in 18% of the opportunities to send a frame.

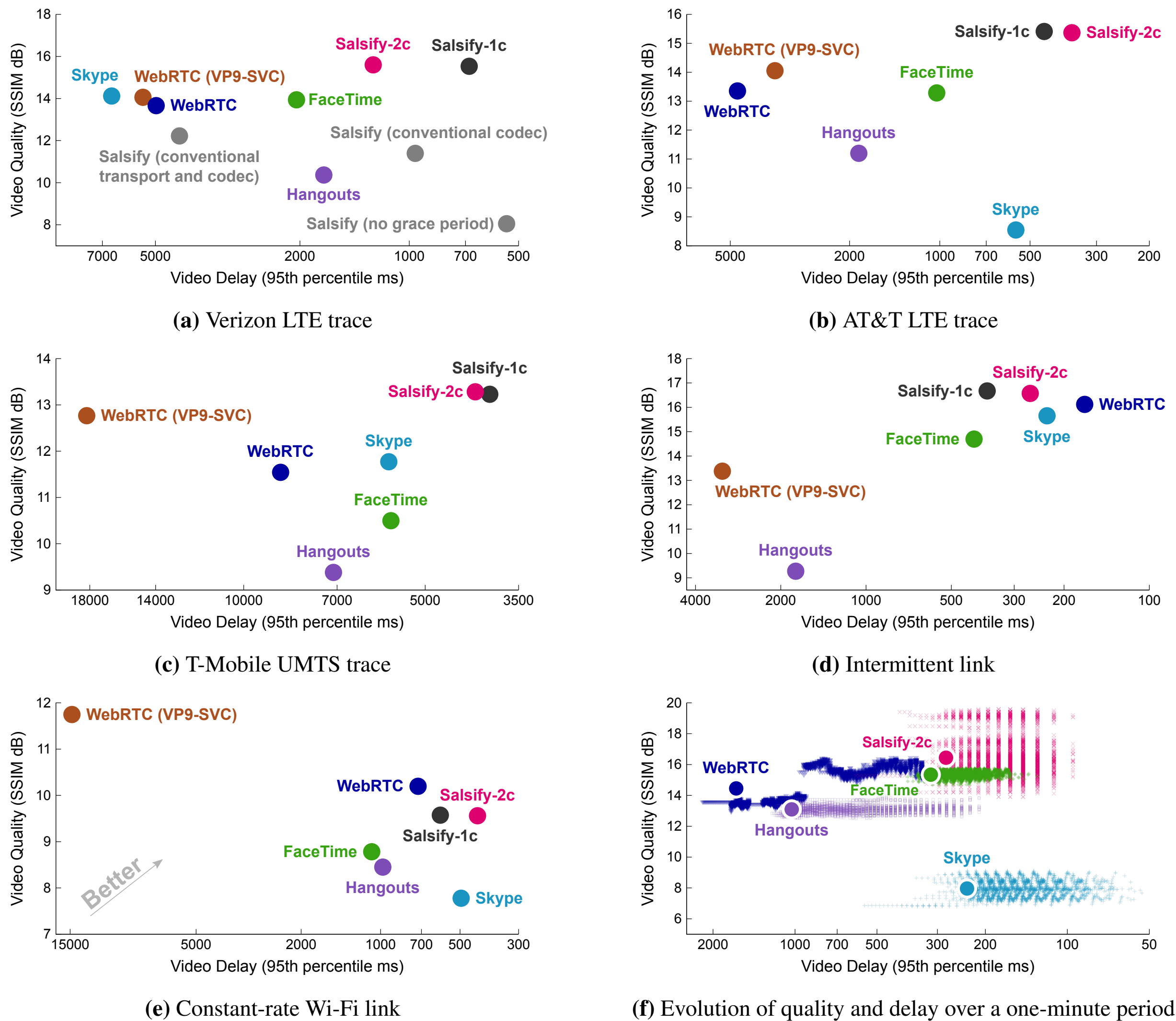
Figure 6f shows how the quality and delay of different schemes vary during a one-minute period of AT&T LTE trace where the network path was mostly steady in capacity.

Experiment 2: intermittent link. In this experiment, we evaluated Salsify's method of loss resilience. We ran each system on a two-state intermittent link. The link's capacity is 12 Mbps with no drops until a "failure" arrives, which happens on average after five seconds (exponentially distributed). During failure, all packets are dropped until a "restore" event arrives, on average after 0.2 seconds of failure. The experiment's duration is 10 minutes.

Figure 6d shows the results for each scheme. The Salsify schemes had the best quality, and their delay was better than all schemes except Skype and WebRTC. Salsify and WebRTC are both on the Pareto frontier of this scenario; further tuning will be required to see if Salsify can improve its delay without compromising quality.

Experiment 3: emulated Wi-Fi. In this experiment, we evaluated Salsify and the other systems on a challenged network path that, unlike the cellular traces, does not vary its capacity with time. This emulated Wi-Fi network path matches the behavior of a long-distance free-space Wi-Fi hop, with the emulation parameters taken from [42], including an average data rate of about 570 kbps and Poisson packet arrivals. Figure 6e shows the results. Salsify is on the Pareto frontier, and WebRTC also performs well when the network data rate does not vary with time.

Experiment 4: component analysis study. In this experiment, we removed the new components implemented



(a) Verizon LTE trace

(b) AT&T LTE trace

(c) T-Mobile UMTS trace

(d) Intermittent link

(e) Constant-rate Wi-Fi link

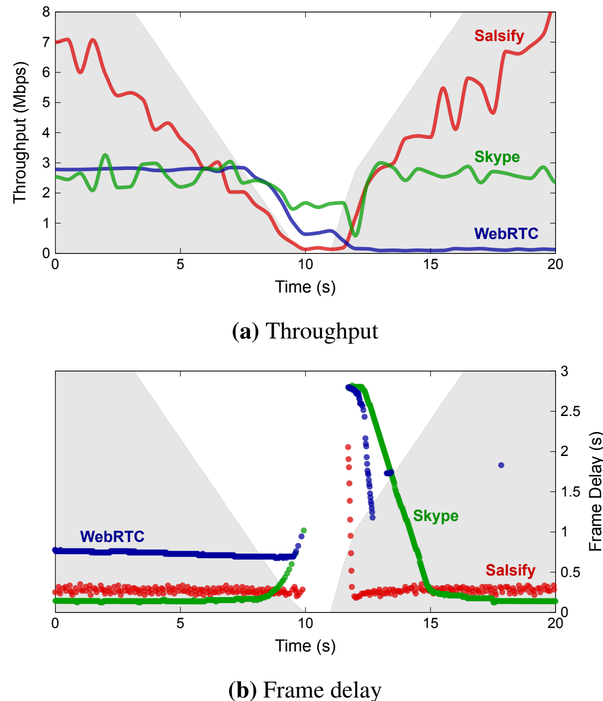
(f) Evolution of quality and delay over a one-minute period

Figure 6: Figures (a)–(e) show the end-to-end video quality and video delay over four emulated networks. Salsify-1c and Salsify-2c achieve better video quality (both on average and the "worse" tail at 25th percentile) and better video delay (both on average and the worse tail at 95th percentile) than other systems for the three real-world network traces (AT&T, T-Mobile, Verizon). Salsify-1c and Salsify-2c perform competitively on the artificially generated "intermittent link" network, which occasionally drops packets but is otherwise a constant 12Mbps (§5), and an emulated constant-rate Wi-Fi link. The AT&T, T-Mobile, Wi-Fi, and and intermittent-link scenarios were held out during development. Figure (f) shows the evolution of the quality and delay during a one-minute period of the AT&T LTE trace when network capacity remained roughly constant.

in Salsify one-by-one to better understand their contribution to the total performance of the system. First, we removed the feature of Salsify's transport protocol that makes it video-aware: the "grace period" to account for intermittent transmissions from the video codec. The performance degradation of this configuration is shown in Figure 6a as the "Salsify (no grace period) dot"; without this component, Salsify underestimates the network capacity and sends low-quality, low-bitrate video.

We then removed Salsify's explicit state-passing-style video codec, replacing it with a conventional codec where the state is opaque to the application, and the appropriate encoding parameters must be predicted upfront (instead of choosing the best compressed version of each frame after the fact). The codec predicted these parameters by performing a binary search for the quality setting on a decimated version of the original frame, attempting to hit a target frame size and extrapolating the resulting size to the full frame. The target size was selected using the same transport protocol in normal Salsify. The result is also in Figure 6a as "Salsify (conventional codec)."

As shown in the plot, Salsify's performance is again substantially reduced. This is a result of two factors: (1) The transport no longer has access to a choice of frames at



(a) Throughput

(b) Frame delay

Figure 7: Salsify's reacts more quickly to changes in network conditions than other video systems. This is illustrated by comparing the performance of Skype, WebRTC, and Salsify over a network path whose capacity decreases gradually to zero, then back up again (instantaneous network capacity shown in gray).

transmission time; if the capacity estimate changed during the time it took to compress the video frame, Salsify will either incur delay or have missed an opportunity to improve the quality of the video, and (2) it is challenging for any codec to choose the appropriate quality settings upfront to meet a target size; the encoder will be liable to under- or overshoot its target.

Finally, we created an end-to-end system that emulates the behavior of the conventional videoconferencing systems, by removing the distinctive features of both Salsify's video codec and transport protocol. Rather than operating frame by frame, the transport protocol in this implementation estimates the network's average data rate and updates the quality settings of the video codec, once every second. As the "Salsify (conventional transport and codec) dot" in Figure 6a shows, this implementation has a similar performance to Skype and WebRTC.

We conclude that each of Salsify's distinctive features—the video-aware transport protocol, purely functional codec, and the frame-by-frame coupling between them that merges the rate-control algorithms of each module—contributes positively to the system's overall performance.

Experiment 5: capacity ramp. In this experiment, we evaluated how Salsify, Skype, and WebRTC handle a network with a gradual decrease in data rate (to zero), then a gradual resumption of network capacity. We created the synthetic network trace depicted in light gray in Figures 7a and 7b. The experiment's duration is 20 seconds.

Figure 7a shows the data transmission rate each scheme tries to send through the link, versus time. Salsify's throughput smoothly decreases alongside link capacity, then gracefully recovers. The result is that Salsify's video display recovers quickly after link capacity is restored, as shown in Figure 7b.

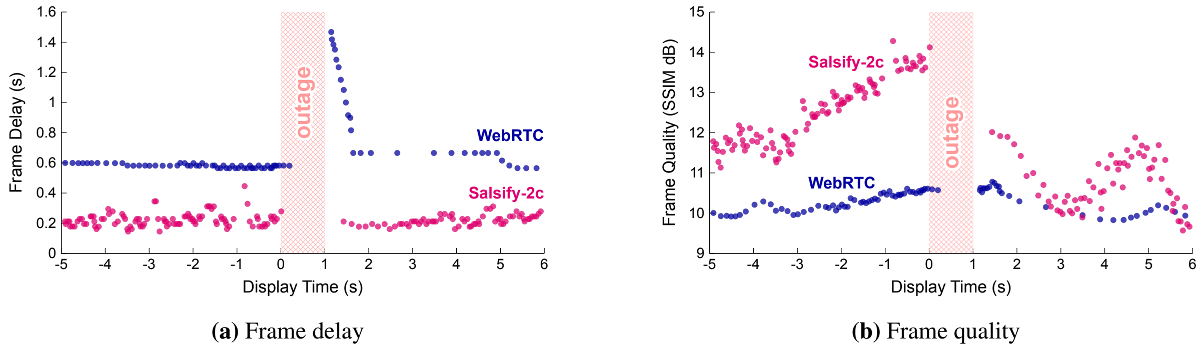
In contrast, Skype reacts slowly to degraded network capacity, and as a result induces loss on the link and a standing queue, both of which delay the resulting video for several seconds. WebRTC reacts to the loss of link capacity, but ends up stuck in a bad mode after the network is restored; the receiver displays only a handful of frames (marked with blue dots) in the eight seconds after the link begins to recover.

Experiment 6: one-second dropout experiment. In this experiment, we compared the effect of packet loss on Salsify-2c and WebRTC by introducing a single dropout for 1 s while running each application on an emulated link with a constant data rate of 500 kbps. All of the packets that were scheduled to be delivered during the outage were dropped. Figure 8 shows the results. After the network is restored, WebRTC's transport protocol retransmits the *packets* that were lost during the outage, causing a spike in delay before its video codec starts encoding new frames of video (WebRTC's baseline delay is also considerably larger). Salsify does not have the same independence between its video codec and transport protocol; upon recovery of the network, Salsify's functional video codec can immediately encode new frames in terms of reference images that are already present at the receiver. This results in faster recovery from the dropout.

Experiment 7: sensitivity to queueing policy. In this experiment, we quantified the performance impact of network buffer size on Salsify, Hangouts, WebRTC, and WebRTC (VP9-SVC) for the Verizon-LTE network trace. The plots in Figure 9 show the performance of each system on the trace across various DropTail thresholds (at 64, 256, and 1024 MTU-sized packets). The performance of the tested systems was not significantly influenced by the choice of buffer size, perhaps because all schemes are striving for low delay and therefore are unlikely to build up a large-enough standing queue to see DropTail-induced packet drops. We ran the remaining tests using the middle setting (256 packets).

5.3 Modifications to systems under test

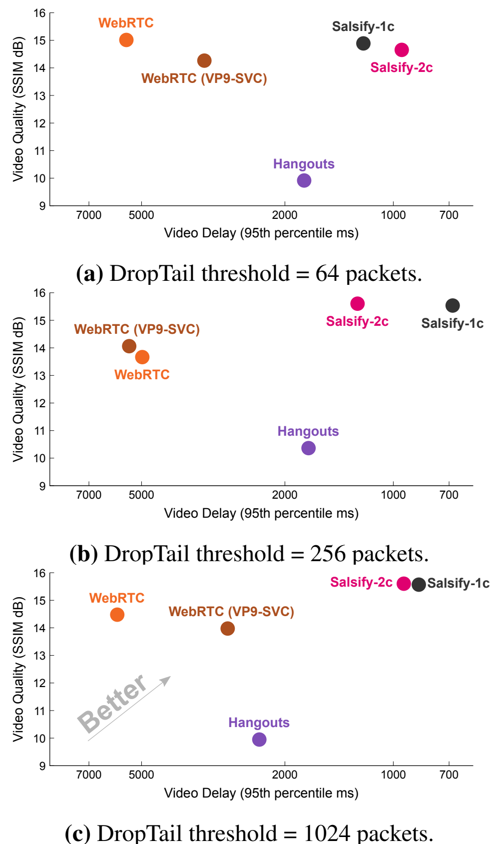
Although the testbed was designed to work with unmodified real-time video systems as long as the sender accepts input on a USB webcam and the receiver displays output



(a) Frame delay

(b) Frame quality

Figure 8: Comparison of the response of Salsify and WebRTC (implementation in Chrome 65) to a single loss event for one second, while communicating over a network path with a constant data rate of 500 kbps. During the loss episode, all packets were dropped. WebRTC displays frames out of a receiver-side buffer during the outage. In contrast to Salsify's strategy of always encoding the most recent video in terms of references available at the receiver, WebRTC's transport protocol retransmits packets lost during the outage before its video encoder starts encoding new frames. This causes a spike in the video delay and wide variations in the frame rate.



(a) DropTail threshold = 64 packets.

(b) DropTail threshold = 256 packets.

(c) DropTail threshold = 1024 packets.

Figure 9: Sensitivity to queueing policy. We measured performance over the Verizon LTE network trace with different in-network buffer sizes. The tested systems were not particular sensitive to this parameter. We ran the other experiments with a buffer size of 256 packets.

fullscreen via HDMI, in practice we found that to evaluate the commercial systems fairly, small modifications were needed. We describe these here.

FaceTime two-way video and self view. Unlike the other video conferencing programs we tested, FaceTime could not be configured to disable bidirectional video transmission. We physically covered the webcam of the receiver when evaluating FaceTime in order to minimize the amount of data the receiver needed to send.

Also, like most video conferencing programs, FaceTime provides a self-view window so users can see the video they are sending. This cannot be disabled in FaceTime and is present in the frames captured by our measurement testbed. To prevent this from unfairly lowering FaceTime's SSIM score, we blacked out the self view window in post-processing (in both the sent and received raw videos) before computing SSIM. These regions accounted for approximately 4% of the total frame area.

Hangouts & WebRTC watermarks and self view. By default, Google Hangouts and our reference WebRTC client (appr.tc) had several watermarks and self view windows. Since these program run in the browser, we modified their CSS files to remove these artifacts so we could accurately measure SSIM.

Hangouts did not make P2P connections. Unlike all the other systems we evaluated, Google Hangouts did not make a direct UDP connection between the two client machines. Rather, the clients communicated through a Google relay server, still via UDP. We measured this delay by pinging the Google server used by the client machines. The round trip delay was < 20 ms in all cases and $\sim$5 ms on average.

6 Limitations and Future Work

Salsify and its evaluation feature a number of important limitations and opportunities for future work.

6.1 Limitations of Salsify

No audio. Salsify does not encode or transmit audio. When testing other applications, we disabled audio to

avoid giving Salsify an unfair advantage. Adding audio to a videoconferencing system creates a number of opportunities for future work in QoE optimization—e.g., it is generally beneficial for the receiver to delay audio in a buffer to allow uninterrupted playback in the face of network jitter, even at the expense of some added delay. To what extent should video be similarly delayed to keep it in sync with the audio, and what are the right metrics to evaluate any compromise along these axes?

Most codecs do not support save/restore of state. Salsify includes a VP8 codec—an existing format that we did not modify—with the addition that the codec is in a functional style and allows the application to save and restore its internal state. Conventional codecs, whether hardware or software, do not support this interface, although we are hopeful these results will encourage implementers to expose such an interface. On power-constrained devices, only hardware codecs are sufficiently power-efficient, so we are hopeful that Salsify's results will motivate hardware codec implementers to expose state as Salsify does.

Improved conventional codecs could render Salsify's functional codec unnecessary. The benefit of Salsify's purely functional codec is principally in its ability to produce (by trial and error) compressed frames whose length matches the transport protocol's estimate of the network's instantaneous capacity. To the extent that conventional codecs also grow this capability, the benefits of a functional codec in this setting will shrink.

Benefits are strongest when the network path is most variable. Salsify's main contribution is in combining the rate-control algorithms in the transport protocol and video codec, and exploiting the functional codec to coax individual compressed frames that match the network's instantaneous capacity, even when it is highly variable. On network paths that exhibit such variability (e.g. cellular networks while moving), Salsify demonstrated a significant performance advantage over current applications. On less-variable networks, Salsify's performance was closer to existing applications.

6.2 Limitations of the evaluation

Unidirectional video. Our experiments used a dedicated sender and receiver, whereas a typical video call has bidirectional video. This is because the testbed only has one Blackmagic card (and pair of high-speed SSDs) and cannot send and capture two video streams simultaneously.

The traces do not reflect multiple flows sharing the same queue. To achieve a fair evaluation of each application, we used the same test video and ran over a series of reproducible network emulators. We did not evaluate the schemes over "wild" real-world paths. The trace-based network emulation replays the actual packet timings (in both the forward and reverse direction) captured from cellular networks. These traces capture several phenomena, including the effect of multiple hops, ACK compression in the reverse path, and cross traffic from other flows and users sharing the network while the traces were recorded, reducing the available data rate of the network path. However, the emulation does not capture cross traffic that shares the same bottleneck queue as the application under test. Generally speaking, no end-to-end application can achieve low-latency video when the bottleneck queue is shared with "bufferbloating" cross traffic [13].

7 Conclusion

In this paper, we presented Salsify, a new architecture for real-time Internet video that tightly integrates a video codec and a network transport protocol. Salsify improves upon existing systems in three principal ways: (1) a video-aware transport protocol achieves accurate estimates of network capacity without a "full throttle" source, (2) a functional video codec allows the application to experiment with multiple settings for each frame to find the best match to the network's capacity, and (3) Salsify merges the rate-control algorithms in the video codec and transport protocol to avoid provoking packet drops and queueing delay with its own traffic.

In an end-to-end evaluation, Salsify achieved lower end-to-end video delay and higher quality when compared with five existing systems: Skype, FaceTime, Hangouts, and WebRTC's reference implementation with and without scalable video coding (VP9-SVC).

It is notable that Salsify achieves superior visual quality than other systems, as Salsify uses our own implementation of a VP8 codec—a largely superseded compression scheme, and an unsophisticated encoder for that scheme. The results suggest that further improvements to video *codecs* may have reached the point of diminishing returns in this setting, but changes to the architecture of video *systems* can still yield significant benefit.

Acknowledgments

We thank the NSDI reviewers and our shepherd, Kyle Jamieson, for their helpful comments and suggestions. We are grateful to James Bankoski, Josh Bailey, Danner Stodolsky, Timothy Terriberry, and Thomas Daede for feedback throughout this project, and to the participants in the user study. This work was supported by NSF grant CNS-1528197, DARPA grant HR0011-15-2-0047, the NSF Graduate Research Fellowship Program (JE), and by Google, Huawei, VMware, Dropbox, Facebook, and the Stanford Platform Lab.

References

[1] ALVESTRAND, H. T. Overview: Real Time Protocols for Browser-based Applications. Internet-Draft draft-ietf-rtcweb-overview-16, Internet Engineering Task Force, Nov. 2016. Work in Progress.

[2] CHEN, M., PONEC, M., SENGUPTA, S., LI, J., AND CHOU, P. A. Utility maximization in peer-to-peer systems. In *ACM SIGMETRICS* (June 2008).

[3] CHEN, X., CHEN, M., LI, B., ZHAO, Y., WU, Y., AND LI, J. Celerity: A low-delay multi-party conferencing solution. *IEEE Journal on Selected Areas in Communications 31*, 9 (Sept. 2013), 155–164.

[4] CHENG, R., WU, W., CHEN, Y., AND LOU, Y. A cloud-based transcoding framework for real-time mobile video conferencing system. In *IEEE MobileCloud* (Apr. 2014).

[5] CHOU, P. A., AND MIAO, Z. Rate-distortion optimized streaming of packetized media. *IEEE Transactions on Multimedia 8*, 2 (April 2006), 390–404.

[6] CICCO, L. D., CARLUCCI, G., AND MASCOLO, S. Experimental investigation of the Google congestion control for real-time flows. In *ACM FhMN* (Aug. 2013).

[7] ELMOKASHFI, A., MYAKOTNYKH, E., EVANG, J. M., KVALBEIN, A., AND CICIC, T. Geography matters: Building an efficient transport network for a better video conferencing experience. In *CoNEXT* (Dec. 2013).

[8] FENG, Y., LI, B., AND LI, B. Airlift: Video conferencing as a cloud service. In *IEEE ICNP* (Feb. 2012).

[9] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)* (2017), USENIX Association, pp. 363–376.

[10] FREDERICK, R. Experiences with real-time software video compression. In *Proceedings of the Sixth International Workshop on Packet Video* (1994).

[11] FUND, F., WANG, C., LIU, Y., KORAKIS, T., ZINK, M., AND PANWAR, S. S. Performance of DASH and WebRTC video services for mobile users. In *IEEE PV* (Dec. 2013).

[12] GANJAM, A., JIANG, J., LIU, X., SEKAR, V., SIDDIQUI, F., STOICA, I., ZHAN, J., AND ZHANG, H. C3: Internet-scale control plane for video quality optimization. In *NSDI* (May 2015).

[13] GETTYS, J., AND NICHOLS, K. Bufferbloat: Dark buffers in the Internet. *Queue 9*, 11 (Nov. 2011), 40:40–40:54.

[14] GRANGE, A., DE RIVAZ, P., AND HUNT, J. VP9 Bitstream & Decoding Process Specification version 0.6, March 2016. http://www.webmproject.org/vp9/.

[15] HAJIESMAILI, M. H., MAK, L., WANG, Z., WU, C., CHEN, M., AND KHONSARI, A. Cost-effective low-delay cloud video conferencing. In *IEEE ICDCS* (June 2015).

[16] HERMANNS, N., AND SARKER, Z. Congestion control issues in real-time communication—"Sprout" an example. Internet Congestion Control Research Group. https://datatracker.ietf.org/meeting/88/materials/slides-88-iccrg-3.

[17] HOLMER, S., LUNDIN, H., CARLUCCI, G., CICCO, L. D., AND MASCOLO, S. A Google congestion control algorithm for real-time communication, 2015. draft-alvestrand-rmcat-congestion-03.

[18] JAIN, M., AND DOVROLIS, C. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2002), SIGCOMM '02, ACM, pp. 295–308.

[19] JAKUBCZAK, S., AND KATABI, D. A cross-layer design for scalable mobile video. In *MobiComm* (Sept. 2011).

[20] JIANG, J., DAS, R., ANANTHANARAYANAN, G., CHOU, P. A., PADMANABHAN, V. N., SEKAR, V., DOMINIQUE, E., GOLISZEWSKI, M., KUKOLECA, D., VAFIN, R., AND ZHANG, H. VIA: Improving internet telephony call quality using predictive relay selection. In *SIGCOMM* (Aug. 2016).

[21] KESHAV, S. A control-theoretic approach to flow control. In *Proceedings of the Conference on Communications Architecture & Protocols* (1991), SIGCOMM '91, ACM, pp. 3–15.

[22] LI, J., CHOU, P. A., AND ZHANG, C. Mutualcast: An efficient mechanism for content distribution in a peer-to-peer (P2P) network. Tech. Rep. MSR-TR-2004-98, Microsoft Research, 2004.

[23] LIANG, C., ZHAO, M., AND LIU, Y. Optimal bandwidth sharing in multiswarm multiparty P2P video-conferencing systems. *IEEE/ACM Trans. Networking 19*, 6 (Dec. 2011), 1704–1716.

[24] LIU, X., DOBRIAN, F., MILNER, H., JIANG, J., SEKAR, V., STOICA, I., AND ZHANG, H. A case for a coordinated Internet video control plane. In *SIGCOMM* (Aug. 2012).

[25] LUMIAHO, L., AND NAGY, M., Oct. 2015. Error Resilience Mechanisms for WebRTC Video Communications http://www.callstats.io/2015/10/30/error-resilience-mechanisms-webrtc-video/.

[26] MCCANNE, S., AND JACOBSON, V. Vic: A flexible framework for packet video. In *Proceedings of the Third ACM International Conference on Multimedia* (1995), MULTIMEDIA '95, ACM, pp. 511–522.

[27] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., AND BALAKRISHNAN, H. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX Annual Technical Conference* (2015), pp. 417–429.

[28] OTT, J., AND WENGER, D. S. Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF). RFC 4585, July 2006.

[29] PONEC, M., SENGUPTA, S., CHIN, M., LI, J., AND CHOU, P. A. Multi-rate peer-to-peer video conferencing: A distributed approach using scalable coding. In *IEEE ICME* (June 2009).

[30] SEN, S., GILANI, S., SRINATH, S., SCHMITT, S., AND BANERJEE, S. Design and implementation of an "approximate" communication system for wireless media applications. In *Proceedings of the ACM SIGCOMM 2010 Conference* (2010), SIGCOMM '10, ACM, pp. 15–26.

[31] SEUNG, Y., LENG, Q., DONG, W., QIU, L., AND ZHANG, Y. Randomized routing in multi-party internet video conferencing. In *IEEE IPCCC* (Dec. 2014).

[32] SULLIVAN, G. J., OHM, J.-R., HAN, W.-J., AND WIEGAND, T. Overview of the high efficiency video coding (HEVC) standard. *IEEE Trans. Cir. and Sys. for Video Technol. 22*, 12 (Dec. 2012), 1649–1668.

[33] SWETT, I. QUIC FEC v1. https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPTcLNJewj7Nk.

[34] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing 13*, 4 (2004), 600–612.

[35] WEBRTC.ORG. WebRTC Native Code. https://webrtc.org/native-code.

[36] WILKINS, P., XU, Y., QUILLIO, L., BANKOSKI, J., SALONEN, J., AND KOLESZAR, J. VP8 Data Format and Decoding Guide. RFC 6386, Oct. 2015.

[37] WINSTEIN, K., AND BALAKRISHNAN, H. Mosh: A State-of-the-Art Good Old-Fashioned Mobile Shell. In *;login:* (37, 4, August 2012).

[38] WINSTEIN, K., AND BALAKRISHNAN, H. Mosh: An interactive remote shell for mobile clients. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012), USENIX. Available at https://mosh.org., pp. 177–182.

[39] WINSTEIN, K., SIVARAMAN, A., AND BALAKRISHNAN, H. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)* (2013), USENIX, pp. 459–471.

[40] WU, Y., WU, C., LI, B., AND LAU, F. C. M. vSkyConf: Cloud-assisted multi-party mobile video conferencing. In *ACM MCC* (Aug. 2013).

[41] XU, Y., YU, C., LI, J., AND LIU, Y. Video telephony for end-consumers: Measurement study of Google+, iChat, and Skype. In *IMC* (Nov. 2012).

[42] YAN, F. Y., MA, J., HILL, G., RAGHAVAN, D., WAHBY, R. S., LEVIS, P., AND WINSTEIN, K. Pantheon: the training ground for Internet congestion-control research. Measurement at http://pantheon.stanford.edu/result/1622/.

[43] YAP, K.-K., HUANG, T.-Y., YIAKOUMIS, Y., MCKEOWN, N., AND KATTI, S. Late-binding: how to lose fewer packets during handoff. In *Proceedings of the 2013 Workshop on Cellular Networks: Operations, Challenges, and Future Design* (2013), ACM, pp. 1–6.

[44] YIN, X., JINDAL, A., SEKAR, V., AND SINOPOLI, B. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *SIGCOMM* (Aug. 2015).

[45] ZHAI, F., AND KATSAGGELOS, A. *Joint Source-Channel Video Transmission*. Morgan & Claypool, 2007. https://doi.org/10.2200/S00061ED1V01Y200707IVM010.

[46] ZHANG, X., XU, Y., HU, H., LIU, Y., GUO, Z., AND WANG, Y. Modeling and analysis of Skype video calls: Rate control and video quality. *IEEE Trans. Multimedia 15*, 6 (Oct. 2013), 1446–1457.

A User studies to calibrate QoE metrics

As part of the development of Salsify, we conducted two user studies to quantify the relative impact of video delay and video quality on quality of experience (QoE) in real-time video applications. These studies were approved by the Institutional Review Board at Stanford University. The participants were all Stanford graduate students and were unpaid. In both studies, we varied video delay and quality over the ranges observed in the comparative evaluation (Section 5). Our results show that small variations in video delay greatly affect mean opinion score; video quality also affects mean opinion score but less so.

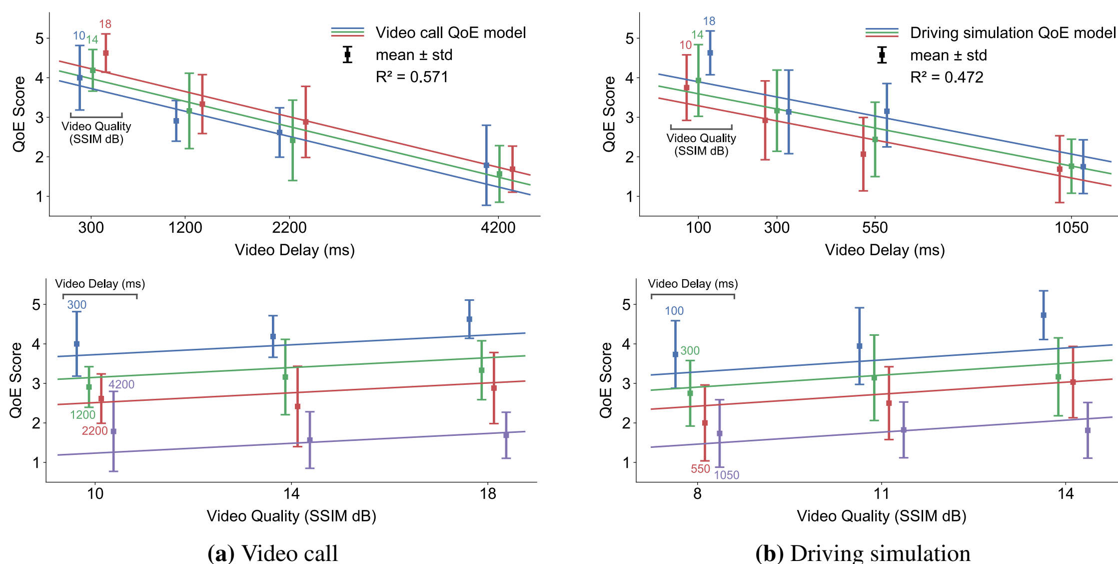
In the first study, participants engaged in a simulated long-distance video conference call with a partner. As part of this study, we built a test jig that captured the audio and video of both participants and played it to their partner with a controlled amount of added delay and visual quality degradation, achieved by encoding and then decoding the video with the x264 H.264 encoder at various quality settings. Participants conversed for one minute on each setting of delay and quality; after each one-minute interval, participants scored their subjective quality of experience on a scale from 1 (worst) to 5 (best). Twenty participants performed this user study, and every participant experienced the same 12 video delay and video quality settings (SSIM dB $\times$ delay: $\{10\,\text{dB}, 14\,\text{dB}, 18\,\text{dB}\} \times \{300\text{ms}, 1200\text{ms}, 2200\text{ms}, 4200\text{ms}\}$).

The second user study put participants behind the wheel of a race car in a simulated environment: a PlayStation 4 playing the "Driveclub" videogame. Using a second test jig, the visual quality and the delay between the PlayStation's HDMI output and the participant's display were controlled. Participants drove their simulated vehicle for 45 seconds on each quality and delay setting, then rated their quality of experience from 1 (worst) to 5 (best). Seventeen participants performed this user study, and all participants experienced the same 12 video delay and video quality settings (SSIM dB $\times$ delay: $\{8\,\text{dB}, 11\,\text{dB}, 14\,\text{dB}\} \times \{100\text{ms}, 300\text{ms}, 550\text{ms}, 1050\text{ms}\}$).

Results and interpretation. We used a two-dimensional linear equation as our QoE model; the model for each user study was fit using ordinary least squares. The resultant best-fit lines (one for the videochat, and one for the driving simulation) are shown in Figure 10. Using the learned coefficients from the videoconferencing study, we predict that a 100 ms decrease in video delay produces the same quality of experience improvement as a 1.0 dB increase in visual quality (SSIM dB). Likewise, in the driving simulation we predict that a 100 ms decrease in video delay is equivalent to a 1.9 dB increase in visual quality. This suggests that in settings such as teleoperation of vehicles, achieving low video delay is more critical than increasing video quality, even more than in person-to-person videoconferencing.

The equations for the best fit lines are given below.

$$\begin{aligned} \text{QoE}_{\text{video call}} &= -6.39 \cdot 10^{-6} \times \text{DELAY}_{\text{ms}} + \\ &\quad 6.22 \cdot 10^{-2} \times \text{SSIM}_{\text{dB}} + 3.30 \\ \text{QoE}_{\text{driving}} &= -1.92 \cdot 10^{-3} \times \text{DELAY}_{\text{ms}} + \\ &\quad 1.01 \cdot 10^{-1} \times \text{SSIM}_{\text{dB}} + 2.67 \end{aligned}$$



(a) Video call **(b)** Driving simulation

Figure 10: The results of the two user studies. The data from each study were fit to a two-dimensional linear model—one for videoconferencing, one for driving—using ordinary least squares. The upper plots project the learned bilinear models onto the delay-QoE axes; similarly, the lower plots show the quality-QoE projection. We found that for a given delay, the quality has only a small impact on the QoE (upper plots); conversely, for a given quality, the delay has a large impact on the QoE (lower plots).

B Numerical evaluation results

System	Trace	Video Quality (SSIM dB) p25	Video Quality (SSIM dB) mean	Video Delay (ms) mean	Video Delay (ms) p95	Data
Salsify-1c	Verizon LTE	15.1	15.5	517.4	684.0	↗
Salsify-2c	Verizon LTE	15.1	15.6	496.7	1256.8	↗
FaceTime	Verizon LTE	15.0*	13.9	658.6	2044.2	↗
Hangouts	Verizon LTE	9.8	10.4	560.9	1719.0	↗
Skype	Verizon LTE	15.1*	14.1	1182.6	6600.2	↗
WebRTC	Verizon LTE	13.2	13.7	973.0	4977.4	↗
WebRTC (VP9-SVC)	Verizon LTE	13.6	14.1	1196.1	5411.9	↗
Salsify-1c	AT&T LTE	15.0	15.4	349.1	448.5	↗
Salsify-2c	AT&T LTE	15.0	15.4	282.1	362.4	↗
FaceTime	AT&T LTE	12.6	13.3	469.4	1023.6	↗
Hangouts	AT&T LTE	10.7	11.2	846.4	1862.4	↗
Skype	AT&T LTE	8.2	8.5	322.1	557.4	↗
WebRTC	AT&T LTE	12.4	13.4	934.7	4729.9	↗
WebRTC (VP9-SVC)	AT&T LTE	13.5	14.1	775.2	3547.2	↗
Salsify-1c	T-Mobile UMTS	13.0	13.2	840.1	3906.8	↗
Salsify-2c	T-Mobile UMTS	12.9	13.3	803.3	4129.4	↗
FaceTime	T-Mobile UMTS	8.8	10.5	1206.8	5699.6	↗
Hangouts	T-Mobile UMTS	8.5	9.4	1012.0	7096.9	↗
Skype	T-Mobile UMTS	11.1	11.8	1451.8	5745.9	↗
WebRTC	T-Mobile UMTS	10.4	11.5	1795.7	8685.5	↗
WebRTC (VP9-SVC)	T-Mobile UMTS	12.1	12.8	2585.2	18215.3	↗
Salsify-1c	Intermittent Link	15.9	16.7	265.2	373.6	↗
Salsify-2c	Intermittent Link	15.8	16.6	181.9	263.3	↗
FaceTime	Intermittent Link	14.6	14.7	280.2	415.8	↗
Hangouts	Intermittent Link	9.1	9.3	437.0	1771.4	↗
Skype	Intermittent Link	15.5	15.7	128.4	229.7	↗
WebRTC	Intermittent Link	16.0	16.1	155.8	169.1	↗
WebRTC (VP9-SVC)	Intermittent Link	12.3	13.4	1735.2	3216.9	↗
Salsify-1c	Emulated Wi-Fi Link	9.0	9.6	317.7	593.9	↗
Salsify-2c	Emulated Wi-Fi Link	9.0	9.6	234.8	429.2	↗
FaceTime	Emulated Wi-Fi Link	8.5	8.8	609.5	1080.2	↗
Hangouts	Emulated Wi-Fi Link	8.2	8.4	514.4	980.5	↗
Skype	Emulated Wi-Fi Link	7.5	7.8	250.9	495.1	↗
WebRTC	Emulated Wi-Fi Link	10.0	10.2	315.2	721.0	↗
WebRTC (VP9-SVC)	Emulated Wi-Fi Link	11.4	11.7	2512.4	14767.3	↗

Figure 11: Summary of results of the evaluation (Section 5). The best results on each metric are highlighted. Two entries marked with a * have a 25th-percentile SSIM that is higher than their mean SSIM; this indicates a skewed distribution of video quality. In the PDF version of this paper, the icons in the data column link to the raw data for each item, within the repository at https://github.com/excamera/salsify-results.

ResQ: Enabling SLOs in Network Function Virtualization

Amin Tootoonchian[⋆] Aurojit Panda[¶‡] Chang Lan[†] Melvin Walls[§]
Katerina Argyraki[•] Sylvia Ratnasamy[†] Scott Shenker[†‡]

[⋆]Intel Labs [†]UC Berkeley [‡]ICSI [¶]NYU [§]Nefeli [•]EPFL

Abstract

Network Function Virtualization is allowing carriers to replace dedicated middleboxes with Network Functions (NFs) consolidated on shared servers, but the question of how (and even whether) one can achieve performance SLOs with software packet processing remains open. A key challenge is the high variability and unpredictability in throughput and latency introduced when NFs are consolidated. We show that, using processor cache isolation and with careful sizing of I/O buffers, we can directly enforce a high degree of performance isolation among consolidated NFs – for a wide range of NFs, our technique caps the maximum throughput degradation to 2.9% (compared to 44.3%), and the 95th percentile latency degradation to 2.5% (compared to 24.5%). Building on this, we present ResQ, a resource manager for NFV that enforces performance SLOs for multi-tenant NFV clusters in a resource efficient manner. ResQ achieves 60%-236% better resource efficiency for enforcing SLOs that contain contention-sensitive NFs compared to previous work.

1 Introduction

Modern networks are replete with dedicated "middlebox" appliances that perform a wide variety of functions. In recent years, operators have responded to the growing cost of procuring and managing these appliances by adopting Network Function Virtualization (NFV). In NFV, middlebox functionality is implemented using software Network Functions (henceforth NFs), which are deployed on racks of commodity servers [18, 36, 38]. This approach offers several advantages including lower costs, easier deployment, and the ability to share infrastructure (*e.g.,* servers) between NFs.

However, there is one oft-overlooked disadvantage to the move to software. Because physical instantiations of these functions relied on dedicated hardware; they had well-understood performance properties which allowed operators to offer performance SLOs [3, 7, 44]. Providing performance guarantees is harder with software, particularly when multiple NFs are *consolidated* on the same server.

While current NFV solutions [28, 41, 47, 49] typically place NFs on dedicated cores, this is insufficient to ensure *performance* isolation. Even when run on separate cores, NFs share other processor resources such as the last-level cache (LLC), memory, and I/O controllers (Figure 1),

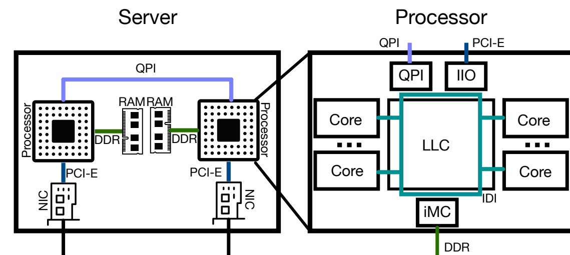


Figure 1: *High-level view of shared resources inside a server and CPU.* A typical NFV deployment consists of racks of servers interconnected with a commodity fabric. Each server consists of a set of resources (CPU, RAM, NIC) interconnected with standard interfaces (QPI, DDR, PCIe). A modern general-purpose Intel CPU consists of a number of processor cores all sharing the *uncore* that includes I/O controller (IIO), integrated memory controller (iMC), last-level cache (LLC), and in-die interconnect (IDI).

collectively referred to as *uncore* resources [29]. NFs contend for these uncore resources and, as we show, such contention can degrade an NF's throughput by as much as 40% compared to its performance when run in isolation (§2).

Providing performance guarantees in NFV essentially boils down to solving the *noisy neighbor* problem, common in multi-tenant environments [62]. Traditionally, this problem has been addressed through resource partitioning. However, in the NFV context, performance variability primarily stems from contention for the LLC [10] and, until recently, no mechanism existed to partition the LLC.[1] This changed with the introduction of processor features – *e.g.,* Intel® Cache Allocation Technology (CAT) [27] – that provides hardware mechanisms for partitioning the LLC across cores.

CAT is a mechanism that opens the door to a new approach for performance isolation in NFV. However, this mechanism has neither been widely tested in nor applied to the NFV context. Hence, in this paper, we study whether and how CAT can be applied to support performance SLOs for NFV workloads. More specifically, we explore the following two questions.

First, we evaluate whether CAT is *sufficient* to ensure performance isolation across NFs? We show that CAT "out of the box" does *not* provide predictable performance: instead, some NFs' performance continues to vary (by as much as 14.7%) depending on their neighboring NFs. This contradicts

[1] Instead, prior work on providing SLOs aimed to *predict* the impact of contention on performance [10]. However, such prediction is difficult and, as we show in §6.2, is no longer accurate with newer hardware and software.

Application	Description	Mpps	Instructions/Cycle	L3 refs/Packet	L3 hit rate	Kilocycles/Packet
Efficuts [61]	Efficuts classifier (32k rules)	1.224	0.63	10.23	99.92	1.72
EndRE [1]	Click-based WAN optimizer	3.770	1.95	1.54	99.95	0.56
Firewall	Click-based classifier (250 rules, sequential search)	0.366	0.59	1.59	99.44	5.74
IPsec	Click-based IPsec tunnel using IPsec elements	0.442	3.31	5.13	99.83	4.75
LPM	Click-based IP router pipeline with RadixIPLookup	5.475	1.92	3.87	99.80	0.38
MazuNAT	Click-based NAT pipeline by Mazu Networks	2.698	1.53	12.14	99.92	0.78
Snort [53]	Inline IDS (20k rules [16, 57]) with netmap for I/O	0.683	1.94	25.61	97.03	3.06
Stats	Click-based flow stat collection with AggregateIPFlows	3.685	1.28	10.22	99.92	0.57
Suricata [46]	Inline IDS (20k rules [16, 57]) with netmap for I/O	0.205	1.61	26.89	98.36	10.
vEPC	Standalone software implementation of LTE core network	-	-	-	-	-

Table 1: *Characteristics of NFs used in this work.* The performance is measured when the NF is run alone with exclusive access to 45 MB LLC with a test traffic of min-sized packets sampled from a pool of 100k flows uniformly at random. Snort and Suricata use netmap while the rest use DPDK for I/O. We do not report these statistics for vEPC due to constraints discussed in §4.1.

prior work [10, 63] which identified cache contention as the main source of performance variability for NFs. Careful investigation reveals the cause of this problem: poor buffer management with *Intel Data Direct I/O* [31], a processor feature that enables direct NIC-to-LLC transfers (*i.e.,* bypassing memory), can lead to a *leaky DMA* problem in which packets are unnecessarily evicted from LLC leading to variable performance. We describe a simple buffer sizing policy that avoids the leaky-DMA problem and show that, with this policy, CAT *is* sufficient to ensure performance isolation (with performance variability under 3% across all scenarios).

Next, having ensured the robustness of CAT as a performance isolation knob given proper buffer sizing, we turn to the question of how to *apply* CAT in a practical system. The challenge here lies in designing a scheduler that assigns resources to NFs in a manner that is accurate (no SLO violations), efficient (minimizing resource use), and scalable (so that decisions can be easily adapted to changing workloads and infrastructure).

We develop ResQ, a cluster resource scheduler that provides performance guarantees for NFV workloads. ResQ computes the number of NF instances required to satisfy SLO terms, and allocates LLC and cores to them. ResQ balances accuracy and efficiency by first *profiling* NFs to understand how their performance varies as a result of LLC allocation. For scalability, ResQ uses a fastpath-slowpath approach. We formulate the scheduling problem as a mixed-integer linear program (MILP) that minimizes the number of machines to guarantee SLOs. Solving this MILP optimally is NP-hard and hence ResQ uses a greedy approximation to schedule NFs upon admission. In the background, it periodically computes a near-optimal solution, and only moves from the greedy to this solution when doing so would lead to a sufficiently large improvement.

We show that ResQ is accurate (with zero SLO violations in our test scenarios), efficient (achieving between 60–236% better resource efficiency compared to prior work based on prediction [10]) and scalable (can profile and admit new SLOs in under a minute).

To our knowledge, our work is the first to analyze the efficacy of using CAT to solve the noisy neighbor problem for a wide range of NFs and traffic types, and ResQ is the first NFV scheduler to support performance SLOs, showing that the benefits of NFV need not come with the loss of what has traditionally been a vital part of carriers' service offerings. ResQ is open source and the code can be found at `https://github.com/netsys/resq`.

The remainder of this paper is organized as follows: we start by quantifying the impact of contention on NFV workloads (§2) and then provide relevant background information and elaborate on the problem we address (§3). We study whether CAT is sufficient for performance isolation in §4, then present the design and evaluation of ResQ in §5 and §6 respectively. We discuss related work in §7, and finally conclude.

2 Motivation

A reasonable first question to ask is whether the current NFV approach of running multiple NFs on shared hardware results in performance variability, *i.e.,* does the *noisy neighbor* problem matter in practice for NFV workloads. We address this question by evaluating the effects of sharing resources for a range of NFs (listed in Table 1), and by comparing their throughput and latency when they are run in *isolation* – *i.e.,* on a dedicated server with no other NFs – to their performance in a *shared* environment comprising of a mix of 11 other NFs (see §4.1). In both cases, we run the NF being evaluated on its own core and allocate the same set of resource to it, thus avoiding any contention due to core sharing. We repeat our measurements using both small (64 B) packets and large (1518 B) packets, and send sufficient traffic to saturate NF cores. We delay a more in depth discussion of our experimental setup to §3.

We show the results of our comparison in Figure 2, which shows the percent degradation in throughput and 95$^{\text{th}}$ percentile latency. Each bar shows the maximum performance loss for an NF running on shared infrastructure when compared to the isolated run. We observe that 7 of the NFs we test demonstrate a performance degradation of more than 10%,

while another 5 show a degradation of more than 20%. Some suffer significant throughput (up to 44.3%) and latency degradation (up to 24.5%) and we find that this holds for both small and large-packet workloads. We also tested the effect of contention on a virtual Evolved Packet Core (vEPC) system using a domain specific packet generator, and observed an 80% degradation in throughput. The vEPC packet generator does not measure latency, and as a result we do not include these results in Figure 2. Finally, we expect that NF degradation will worsen as we increase the number of NFs that share a server.

In conclusion, we find that most NFs suffer significant degradation due to resource contention – this holds for both small and large packet traffic.

3 Background and Problem Definition

Next we present some background for our work, focusing in particular on describing NFV workloads, identifying sources of contention that affect network functions, evaluating prior work in this area, and introducing the processor cache isolation mechanism used by ResQ. Finally, we define the NFV SLO enforcement problem that we address in the rest of this paper.

3.1 NFV Workloads

NFV workloads consist of packet-processing applications, canonically referred to as Network Functions (NFs); they range from relatively simple with lightweight processing (*e.g.,* NAT, firewall) to more heavyweight ones (*e.g.,* vEPC [17]). NFs may be *chained* together such that packets output from one NF is steered to another. For example packets might first be processed by a firewall and then a NAT.

NF performance can vary – even in the absence of the noisy neighbor problems, and an individual NF running in isolation will often display variance in performance across runs. Work over the last decade has led to practices that have been shown to improve performance stability for software packet processors and are now widely understood and adopted [11, 13, 42]. The most significant ones include running NFs on dedicated and isolated cores that use local memory and NICs (NUMA affinity), maintaining interrupt-core affinity, disabling power saving features (*i.e.,* idle states, core and uncore frequency scaling), and disabling transparent huge pages. We adopt the same and, from here on, all our discussion of performance predictability assumes that the above techniques are already in use. As we shall show, these are necessary but not sufficient – we still need to address contention for shared resources, which is our focus in this paper.

3.2 Sources of Contention

Naïvely, one might believe that placing NFs on independent cores ensures that they do not share resources.[2] However, in modern processors, cores share several resources. Resources shared across cores include: PCI-e lanes and CPU's integrated I/O controller, and memory channels and CPU's integrated memory controller, and last-level cache (LLC) as shown in Figure 1. Currently, most servers do not oversubscribe PCIe lanes, and NICs do not contend for these resources. While independent NFs might share PCIe lanes when sharing NICs using SR-IOV [33] or through a software switch [25], one can control contention for these resources by rate limiting ingress traffic received by an interface. As a result the main resource that NFs in shared infrastructure can compete on are memory and LLC, and we study the effect of both in this paper.

[2] There may also be contention within the fabric connecting different servers; that is outside the scope of this paper, but we envisage that standard fabric QoS and provisioning mechanisms [6, 54] can be applied there.

3.3 Prior Work (or Lack Thereof) in NFV

To our knowledge, the only work that analyzes the impact of resource contention on NFV workloads is a work by Dobrescu *et al.* [10]. That work proposes using a simple model for predicting performance degradation due to contention for the last level cache. However, as shown in §6.2, the model is inaccurate when tested under newer hardware and different workloads – *e.g.,* we find that their model overestimates the impact of contention by as much as 13% (a relative error of 75%) for newer hardware and workloads. In addition, that work focuses on predicting degradation rather than meeting performance guarantees; consequently, it does not discuss how one can *enforce* a desired limit on the level of contention. In contrast, our work focuses on enforcing SLOs using hardware mechanisms such as CAT. As we show in §4, ResQ provides robust performance guarantees for a variety of workloads.

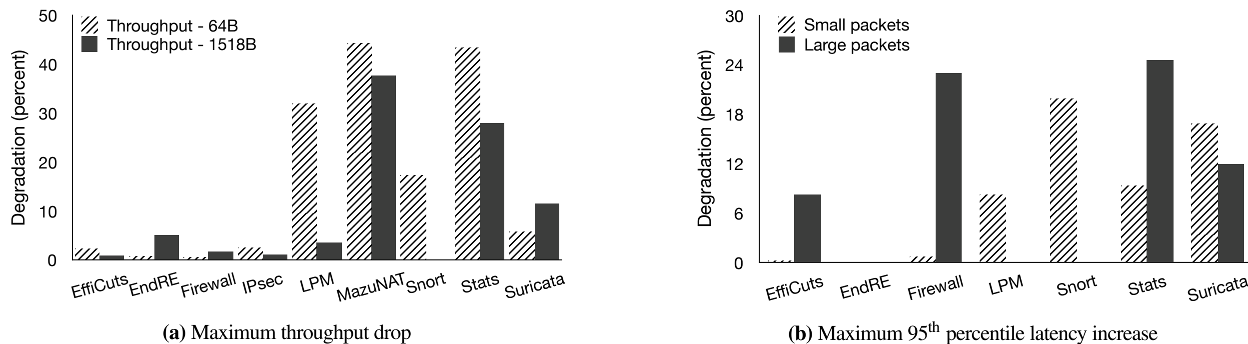
Other work has looked at managing NFV jobs. This includes works such as E2 [47], Stratos [20], OPNFV [36]. While these systems perform some basic allocation of resources to NFs, none consider contention nor do they aim to provide performance guarantees. ResQ can be incorporated into these systems allowing them to provide performance guarantees; ResQ is currently under evaluation for adoption in one commercial orchestrator.

3.4 Hardware Cache Isolation

ResQ's enforcement relies on recent processor QoS features implemented in processors that enable monitoring and control of shared processor resources. For Intel processors, these features are collectively known as the Intel Resource Director Technology (RDT) which include Intel Cache Allocation Technology (CAT) [30][3] and Cache Monitoring Technology (CMT). They allow users to allocate or monitor the amount of cache accessible to or used by threads, cores, or processes.

To monitor a set of processes or cores using CMT, the kernel allocates a resource monitoring ID (RMID) which the processor uses to collect usage statistics for them. The kernel

[3] Cache partitioning is also available in other server processors, for example Qualcomm's Amberwing processor [43] which is based on ARM64.



(a) Maximum throughput drop

(b) Maximum 95th percentile latency increase

Figure 2: *Maximum performance degradation for minimum and MTU-sized packets without isolation.* Due to interference, throughput and latency degrade up to 44.3% and 24.5% respectively. Small and large packet trends are similar for all NFs. We do not measure latency for NFs that mangle packets in a way incompatible with our traffic generator's timestamp embedding (MazuNAT, IPSec), and vEPC whose domain-specific traffic generator does not report latency. LPM and Snort do not exhibit sensitivity with large packets due to a testbed limitation: their cores were not saturated at line rate.

updates a core register with this RMID upon context switch to bind the monitored entities to the RMID. Similarly, when limiting the cache available to processes or cores, the kernel first allocates a class of service identifier (CLOS). It then updates a register to specify the amount of cache accessible to a CLOS. Finally, the kernel can associate a CLOS with a process by updating the appropriate register when switching to the process. Linux allows users to specify the set of processes to be monitored using a newly introduced `RDT` interface. For the evaluation reported in this paper, we used these features as implemented on the Xeon® processor E5 v4 family which allows users to specify up to 16 cache classes. The processor we use for our evaluation allows us to enable access to between 5%–100% of the cache, in 5% increments, for each CLOS. To our knowledge ResQ is the first research work that uses CAT to provide performance isolation in NFV.

3.5 Problem Definition

Our goal is to support performance SLOs for NFV workloads. The conjecture driving this paper is that CAT gives us a powerful and practical knob to achieve this. To validate this conjecture we must answer the following questions:

1. The crux of providing SLOs is knowing how to isolate different NFs from a performance standpoint. Is CAT sufficient to ensure performance isolation between NFs or do we also have to consider contention for other resources? We study this question in §4.
2. CAT is ultimately just a configuration knob and using it in a practical system raises a number of questions: what is a good scheduling algorithm that balances scalability (scheduling decisions per second), accuracy (minimizing SLO violations), and efficiency (minimizing use of server resources)? What is the API for SLOs or *contract* between NFs and the NFV scheduler? What information do we need from NFs to make good scheduling decisions? We address these through the design, implementation, and evaluation of ResQ in §5 and §6.

4 Enforcing Performance Isolation

Dobrescu *et al.* [10] argued that the level of LLC contention entirely determines NF performance degradation. This observation would lead one to believe that merely enabling CAT – which controls the level of cache contention – is sufficient to ensure performance isolation, *i.e.,* ensure that one NF's performance is unchanged due to the actions of any other colocated NF. In this section we evaluate this hypothesis, and find that it does not hold; we then explain why this is the case and present our strategy for mitigating this issue.

4.1 Experimental Setup

NF workloads. We ran our evaluation on a range of NFs (see Table 1) including: NFs from the research community (*e.g.,* Efficuts [61], EndRE [1]) and industry (*e.g.,* Snort [53], Suricata [46], vEPC [17]); NFs with simple (*e.g.,* Firewall, LPM) and complex (*e.g.,* Snort, Suricata) packet processing; NFs with small (*e.g.,* IPSec) and large (*e.g.,* Snort, Stats) working set sizes; NFs using netmap [52] (*e.g.,* Snort and Suricata) and DPDK [12] (*e.g.,* Efficuts and Click) for I/O; NFs that are standalone (*e.g.,* Snort) and those that are built on frameworks like Click (*e.g.,* MazuNAT). We also evaluated the impact of contention on an industrial virtual Evolved Packet Core [17] system[4] that implements LTE core network functionality in software. Due to licensing issues these tests were run on a different testbed, and made use of a domain-specific commercial traffic generator.

Test setup and CAT configuration. We ran all our evaluation on a server with an Intel Xeon E5-2695 v4 processor and dedicated 10 Gb/s and 40 Gb/s network ports.

We repeat the same experiments as in §2 after enabling CAT. We evaluate two scenarios for each NF:

- *Solo* run, where we run the NF under test on a single core and CAT is configured to allocate 5% of LLC to the NF (the smallest allocation with CAT). We run *no* other NFs run on the machine. This provides us with a baseline for

[4] Vendor name anonymized due to licensing requirements.

how the NF behaves with a specific LLC allocation but no contention on the other resources.

- *Shared* runs, where the NF under test is run on a single core and we use CAT to allocate 5% of LLC to the NF. We run 11 instances of a different competing NF on the remaining cores, these instances share the remaining 95% of LLC. We repeat this experiment to analyze the performance impact of each type of competing NF, *i.e.,* in each iteration we pick a different NF from Table 1 to use as the competing NF.

Observe that in both cases, the NF under test is allocated the same number of cores and the same amount of LLC. In our experiments we measure the target NF's performance in terms of throughput and 95th percentile latency, and compute performance degradation for *shared runs* compared to a *solo run*.

NF	Size (bytes)	Degradation (%)	LLC Miss Rate (%) NF	TX	RX	Mem BW Util. (%)
MazuNAT	64	2.4	65.5	0.02	8.49	31
	1518	12.2	72.9	55.3	88.6	94
Stats	64	0.3	64.5	0.07	8.84	31
	1518	14.7	73.1	63.5	89.1	99

Table 2: *CAT does not sufficiently isolate NFs in shared runs with large packets.* The culprit is the high memory bandwidth utilization with large packets which in turn is because the "leaky DMA" issue renders DDIO ineffective. All numbers are the worst-case numbers for shared runs. TX/RX LLC miss rate and memory bandwidth utilization are processor-scoped whereas NF degradation and its LLC miss-rate are NF-scoped.

4.2 Is CAT Sufficient?

Surprisingly, our results were mixed and showed that CAT is *not always* sufficient for providing performance isolation: while CAT was successful at isolating NFs when processing small (64 B) packets, where throughput and latency degradation remained below 3%, it could not isolate all NFs when large (1518 B) packets were used: we observed degradation of up to 14.7% for some NFs (*e.g.,* MazuNAT and Stats). This is particularly surprising in light of the fact that NFs process packets at a higher rate when small packets are used, as a result, memory accesses should be more frequent for smaller packets compared to larger packets, and one would expect greater degradation for smaller packets.

We began our investigation into this anomalous result by checking whether there was a difference in cache miss rates between shared and solo runs. Unsurprisingly, we found no noticeable difference and concluded that CAT was functioning as expected. Next, we analyzed measurements from other hardware counters in the platform and found that memory bandwidth utilization increased substantially in going from small to large packets (Table 2).

Can memory contention affect NF performance? To answer this question, we first used the Intel Memory Latency Checker (MLC) [32] to measure memory access latency as a function of increasing memory bandwidth utilization. We plot the memory access speed (*i.e.,* inverse of the latency) in Figure 3 and find that with up to approximately 90% load on memory channels, the memory access speed degrades linearly with increase in load, and subsequently experiences super-linear degradation dropping to 40% of the baseline value.

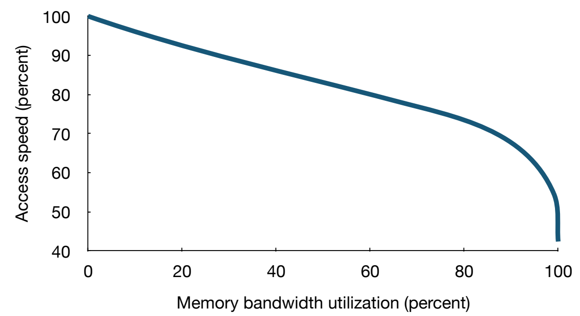


Figure 3: *Memory access speed as a function of load.* The memory access latency increases linearly with load on memory controller up to around 90% utilization.

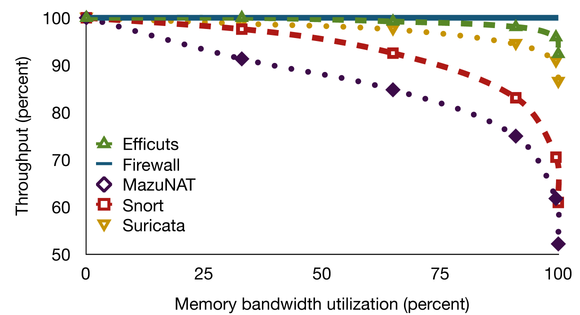


Figure 4: *Normalized NF throughput for a selection of NFs as a function of memory load.* The curves track the memory access speed curve (Figure 3) very closely.

Next, we checked whether this observation meant that NF performance would also degrade with increased memory contention. We analyzed this by running NFs under the same environment as was used for the solo runs and running MLC on the other cores of the same server to generate memory bandwidth load. We show the results for this experiment in Figure 4 and find that the added memory contention does lead to performance degradation for NFs; NFs like MazuNAT are up to 50% slower with aggressive memory contention. Note that this is a near worst-case degradation in response to memory contention – MLC exhibits a more aggressive memory access pattern when compared to network functions (and most other applications).

What causes memory contention? We certainly did not expect to see much memory traffic in our shared workload. While a single core is capable of inducing around 12 GB/s traffic on the memory controller, we expect cores running NFs to generate a fraction of this load. That is because, cycles during which the NF may access state are spaced out by cycles spent

on compute intensive portions of packet processing including I/O, framework processing, and stateless portions of the NF processing. To empirically validate this hypothesis, we wrote a synthetic NF that accesses DRAM 1000 times per packet and observe that it can only generate 2.5 GB/s of memory traffic – we expect a realistic NF to generate far less traffic.

Furthermore, our processor is equipped with Intel Data Direct I/O (DDIO) technology [31] which lets DMAs for packet I/O interact with the last level cache rather than going to DRAM. As a result, we did not expect packet I/O to contribute to memory contention. However, given our expectation that NF state accesses should be more frequent with small packets (due to higher packet rates) we suspected that some interaction with DDIO might be the root cause of the substantial increase in DRAM traffic.

4.3 The Leaky DMA Problem

By default, DDIO is limited to using 10% of the LLC. When a buffer that needs to be DMAed is not present in LLC, it is first brought into the LLC resulting in memory traffic. We hypothesized that this might be the cause of memory contention in our system. Furthermore, DMA transfers are mediated by the processor DMA engine (as opposed to a core), therefore, cache misses during DMA are not included in the core-based LLC counters we used when evaluating the efficacy of CAT above. To test our hypothesis, we looked at CPU performance counters that measure PCIe-sourced LLC references and misses, and found that the I/O-related LLC miss-rates increased from nearly 0% to around 60% on the TX path and 90% on the RX path when going from small packets to large packets in the shared runs. This showed that DDIO is ineffective at preventing memory contention in our system, but why?

The LLC space used by DDIO cannot be partitioned using CAT, and is shared across NFs. As a result, if the aggregate number of packet buffers exceed DDIO's LLC space then packet I/O can contend for cache space and evict buffers holding packets being processed. The maximum number of in-flight packets is bounded by the number of descriptors available to NIC queues. For the experiments above, NFs had their own queue each with 2048 descriptors – that is a total of 24576 buffers for 12 queues. In the shared runs, this translates to requiring 3 MB of cache space for small packets (each spanning 2 cache lines), but a whopping 37.5 MB for large packets (each spanning 25 cache lines).

As noted earlier, we had not observed significant changes in NF cores' cache miss rates when comparing solo and shared runs. This suggested that LLC contention due to DDIO does not affect parts of the packet that are processed by the NF. We thus found that DDIO frequently evicts cache lines belonging to packets that are being processed, and these are needed soon after eviction for packet TX. Similarly the RX path frequently needs to fetch buffers that were previously evicted due to DDIO space contention. Together, they result in much of the network traffic and stale buffers to bounce back and forth between LLC and DRAM multiple times. We refer to this problem as the *leaky DMA* problem and identify it as the root cause of performance variability for NFs when CAT is used.

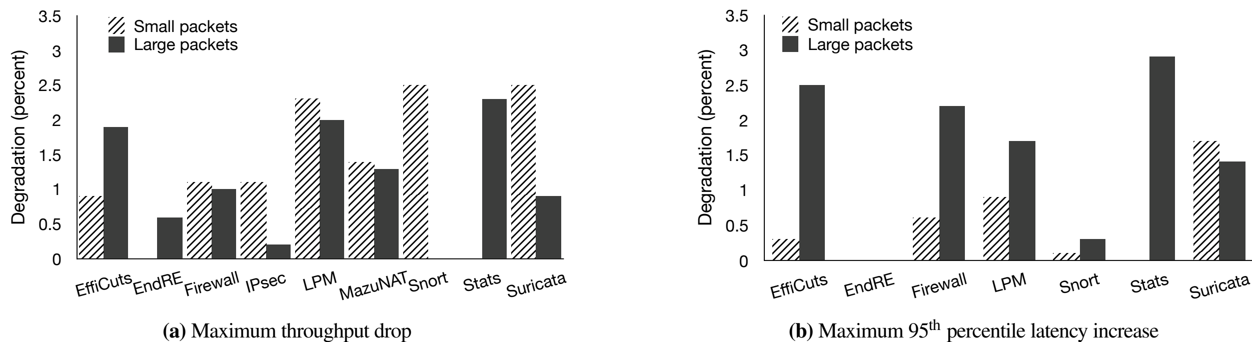
4.4 Solution: CAT + Buffer Sizing

Fortunately, both DPDK and netmap provide mechanisms to control the number of DMA buffers used by the system. In case all DMA buffers are in use, no packets are received from the NIC. DMA buffers become available once the packet data contained within them is freed, at which point new data can be received. Therefore controlling the number of in-flight buffers allows us to control the efficacy of DDIO. In ResQ, we restrict the size of the pool from which packet buffers are allocated based on the aggregate number of MTU sized packets that can fit in the LLC space reserved for DDIO, thus avoiding the leaky DMA problem. Note that NICs also contain a sizeable buffer (4096 packets in Intel NICs) and as a result this restriction does not result in packet loss unless the incoming link is congested.

We evaluated the efficacy of using buffer sizing to solve the *leaky DMA* problem by rerunning both solo and shared runs after fixing the number of allocated packet buffers to the number calculated above. As shown in Figure 5, this resulted in a situation where for both packet sizes throughput and latency degradation were less that 3% (including the vEPC which is not reported in the Figure 5), thus confirming our fix.

Other Resources. Given our experience with memory contention, one might be concerned about contention on other resources which we briefly discuss here. We do not observe notable *IOMMU* contention [50] since we use statically-mapped DMA buffers backed by huge pages. The maximum degradation we observe in a microbenchmark that maximizes core to *IDI* traffic is below 4% – in practice, the IDI utilization is much lower and the degradation is negligible. *IIO* throughput in the Haswell/Broadwell processors is around 160 Gbit/s which may introduce a bottleneck if all PCIe lanes (40) are more than half utilized. However, the aggregate traffic per CPU remains below 150 Gbit/s in our experiments; classical QoS mechanisms would sufficiently address the fair-sharing of this resource. Consequently, we conclude that contention for these other resources is not a concern given the current architecture.

Recap. To summarize we found that while memory contention can be a source of performance variability, this is *not* a result of NF behavior, but rather because of poor DMA buffer sizing which can result in the *leaky DMA* problem. The leaky DMA problem causes DMA buffers to be repeatedly evicted from cache which in turn results in high memory bandwidth utilization. We address the leaky DMA problem by appropriately controlling the aggregate number of active DMA buffers, and find that this, in conjunction with CAT, is sufficient to ensure performance isolation for NFV workloads. Fi-



(a) Maximum throughput drop

(b) Maximum 95th percentile latency increase

Figure 5: *Maximum degradation in throughput and 95th percentile latency for minimum and MTU-sized packets when target NF is isolated per §4.4.* They are both consistently below 3% across all the experiments. When comparing, note the difference in the y-axis range with Figure 2 and that the results do not include latency measurements for IPSec and MazuNAT due to traffic generator's constraints.

nally, our results here show that as opposed to what was found by Dobrescu *et al.* [10], issues beyond LLC sizing can affect NF performance. We believe this is because of changes to software (*e.g.,* use of DPDK), and hardware architecture (*e.g.,* DDIO). Additionally that work considered only small packet sizes and did not analyze accuracy with MTU-sized packets.

5 ResQ

In this section we present the design and implementation of ResQ, a cluster resource manager that is designed to efficiently schedule NFs while guaranteeing that SLOs hold. We begin by describing ResQ's design and we focus our discussion on three aspects:

Service interfaces: Traditionally, network operators have relied on resource overprovisioning to meet performance objectives with hardware network appliances. NFV can allow us to guarantee SLOs while more efficiently utilizing resources. However, achieving this greater efficiency requires that tenants provide ResQ with workload and other information in addition to the NF. We describe ResQ's inputs and the types of guarantees it can enforce in §5.1.

NF profiling: Given an input NF, ResQ needs to determine its resource requirements. These depend on the NF configuration, input traffic, and platform and thus varies across tenants and operators. In §5.2, we describe ResQ's efficient and automatic profiler that measures how NF performance (both throughput and latency) varies as a function of LLC allocation. The ResQ profiler minimizes the number of executions required to collect this information, and can thus rapidly profile a large set of NFs. The profiler's output is a key input to the ResQ scheduler.

NF scheduling: Finally, we present our scheduler in §5.3. ResQ implements a two-level scheduler that takes as input NFs, SLO specifications and requirements, and profiling results and determines *(a)* the number of NF instances to start, *(b)* the server(s) on which these instances must be placed; and *(c)* the amount of the LLC to assign to each instance.

5.1 ResQ SLOs

How do we improve efficiency of resource utilization while continuing to meet performance objectives? Our insight is that how an NF performs – given a fixed set of resources – depends on two factors. First, NF configuration such as rule set of a firewall or an IDS – the size and complexity of this configuration directly affects performance [4, 15]). Second, traffic profile which captures characteristics such as distributions for flow arrival, flow sizes, packet sizes, and packet interarrivals. NF data sheets often highlight that performance depends not just on the input traffic rate but also on factors such as the number of new sessions per second and traffic mix [48]. ResQ improves scheduling efficiency by accounting for these factors when allocating resources.

Tenants can specify two types of performance SLOs: *reserved* and *on-demand*, which we explain next.

Reserved SLOs specify the NF or chain, its expected configuration and traffic profile, and its performance target (*i.e.,* expected latency and throughput). Given this information, ResQ profiles (see §5.2) the NF to determine its performance as a function of resource use, and uses this information to allocate resources. ResQ does not distinguish between NFs or chains of NFs and profiles a chain similarly to a single NF. Since we assume that the traffic profile, configuration, and maximum input rate (specified as part of the performance target) do not vary, implementing the computed allocation is sufficient to satisfy the SLO term. Run-time deviations from the specified traffic profile or NF configuration may only violate the corresponding SLO term – it does not affect other SLOs because ResQ provides sufficient isolation among SLOs. Tenants are required to submit a new admission request to ResQ in the event any of these parameters change; in response, ResQ may either reallocate resources or deny admission if objectives cannot be met.

ResQ ensures stable resource usage for NFs making use of reserved SLOs. This simplifies resource provisioning for the network operator without significantly affecting efficiency for NFs with stable configuration and input

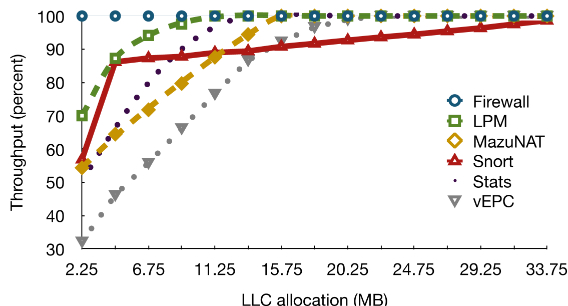


Figure 6: *Normalized throughput as a function of LLC allocation for a selection of NFs.*

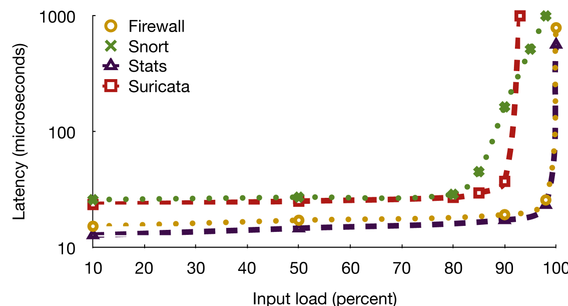


Figure 7: *Latency as a function of normalized input traffic load for a selection of NFs.*

traffic. We envision that, similar to cloud providers, network operator will encourage the use of reserved SLOs by providing volume discounts to tenants. Reserved SLOs are, however, inefficient for NFs with highly variable workloads – *e.g.,* NFs with high traffic variance – since these must be overprovisioned to meet worse-case traffic demands. Such NFs are better suited to use on-demand SLOs.

On-demand SLOs specify the NF and target latency. ResQ continuously monitors NF latencies and resource utilization, and dynamically adjusts resource allocations to meet the target latency independent of the input traffic or configuration. If the target latency could not be met under a best-case allocation, ResQ raises an error. Furthermore, ResQ relies on traffic policing to appropriately reduce input load if it is unable to meet the total traffic demand – *e.g.,* due to lack of resources or reaching a user-specified cap on resource usage. We provide further details about in §5.3.

5.2 ResQ Profiler

ResQ relies on performance profiles to determine resource allocation for reserved SLOs. A ResQ profile consists of throughput-LLC allocation (*e.g.,* Figure 6) and latency-load (*e.g.,* Figure 7) curves. To construct these curves, the profiler runs a set of experiments and collects measurements. The time taken to run one experiment varies depending on the traffic pattern – it takes around 5 seconds with our sample traffic profiles. Building a general NF profile that is valid across all configurations and traffic patterns would likely require exploring a potentially unbounded space and is infeasible. Profiles generated by ResQ are, therefore, specific to not just the NF, but also the configuration and traffic pattern specified by a reserved SLO. Since our profiles are quite specific, we might require a large number of profiles for an NFV cluster; consequently, we must ensure that profile generation is *fast*. Furthermore, errors in an NF profile affect ResQ's accuracy and efficiency, and therefore we need to ensure that generated profiles are *accurate*. We rely on interpolation, with dynamically varying interpolation intervals, to quickly produce accurate profiles as described below.

The throughput-LLC allocation curve for an NF can be generated by running it alone on a profiling server and measuring its throughput as the profiler varies the amount of allocated LLC using CAT §3.4. To generate the latency-throughput curve (*e.g.,* Figure 6), the profiler launches the NF with a given LLC allocation and measures the 95$^{\text{th}}$ percentile latency as a function of input traffic rate. This measurement is repeated for different LLC allocations to produce a latency-throughput curve. In Figure 7, we show an example of such curves for a fixed LLC allocation (we chose to allow NFs to access all of the LLC in this case).

Since the profiler is in the critical path of the admission control process, naïvely running all the required experiments (400 datapoints for around 20 utilization levels and 20 LLC allocations) delays the process significantly (*e.g.,* 34 minutes with 5 second runs). To alleviate this bottleneck in the admission control process, we observe that these curves could be accurately constructed with far fewer datapoints.

We observed that, across a wide range of NFs, the latency-throughput curves vary only slightly for different LLC allocations. As a result, we can safely approximate this curve by measuring an NF's worst-case latency, which corresponds to the LLC allocation that maximizes NF throughput. Furthermore, we observed that both sets of curves are monotonically increasing, and that in all cases the throughput-LLC allocation is concave, while the latency-throughput curve is convex. This allows us to approximate the curve by measuring throughput and latency at a few points, and using linear interpolation to compute values for intermediate points. We implement our interpolation as follows: the profiler begins by measuring the minimum, maximum and midpoint of each curve. It then computes the linear interpolation error by comparing the interpolated value with the measured mid-point. If the interpolation error is above 1%, the profiler recursively splits both intervals and repeats the same procedure. The profiler stops collecting additional measurements once the interpolation error falls below 1%. In our experience, each profile required between 8–12 measurements and could be constructed in under a minute.

5.3 ResQ Scheduler

The ResQ scheduler is comprised of two parts:

- A *centralized scheduler* is responsible for admission control, placement for all SLOs, resource allocation for

reserved SLOs, and setting aside resources on individual servers for on-demand SLOs.

- A *server agent* that runs on each server and is responsible for configuring the server, monitoring resource utilization, and detecting SLO violations. The server agent implements a *local scheduler* that is responsible for allocating resources to NFs with on-demand SLOs that are placed on the server by the centralized scheduler.

In clusters running ResQ, tenants submit SLO requests to the centralized scheduler which performs admission control. For on-demand SLOs, the scheduler checks if the cluster has sufficient resources available to launch one instance of the NF (the supplied NF description includes information about minimum resources required by an instance), and rejects the SLO should sufficient resources not be available. For reserved SLOs, the scheduler consults the NF profile (see §5.2) to determine whether the SLO is feasible; if so, the scheduler uses a greedy algorithm (see §5.3.1) to compute NF placement and resource requirements for meeting the performance objectives. Admission is denied if the greedy algorithm cannot find a fit, otherwise it notifies the appropriate server agents to launch NF instances and allocate the requested resources to them. The scheduler also programs the datacenter fabric so as to steer traffic to these NF instances – similarly to existing NFV schedulers [20, 21, 47, 51], we assume that the fabric will split traffic across these instances.

While the greedy allocation computed by the central scheduler is sufficient for meeting the performance objectives, it might not be optimal in terms of resource use. Therefore, in the background, ResQ also periodically solves a mixed-integer linear program (MILP) to find a (near-)optimal schedule. If the gap in resource usage between this and the greedy schedules exceeds a configurable threshold, ResQ migrates running NFs[5] to implement the optimal schedule. Migrating to the optimal schedule frees up more resources that can be used to accommodate other SLOs.

On-demand SLOs are scheduled locally by server agents. Upon submission of an on-demand SLO, the centralized scheduler finds a server that has sufficient resource to run one instance of the NF and assigns the on-demand SLO to that server. The server agent uses max-min fair allocation to partition the on-demand LLC space among such NFs. If the server agent is unable to meet the NFs latency targets, it notifies the central scheduler which in turn adds NF instances to the cluster.

Next we provide more details about the algorithm used for scheduling both types of NFs.

5.3.1 Reserved SLOs

Computing the optimal schedule for reserved SLOs is an NP-hard problem. Hence, we develop an online greedy algorithm for fast admission. After the profile is generated, ResQ attempts to greedily bin-pack the NF instances using a first-fit heuristic, which works as follows.

1. It divides the target throughput by the expected throughput of a single instance to estimate the number of NF instances required to meet the objective. The expected throughput of one instance is what a single instance can sustain when allocated a fair share of LLC (*i.e.,* the available LLC divided by the number of cores) such that its latency does not exceed the target latency.
2. It calculates the minimal LLC allocation for each instance by iteratively adding a unit of LLC allocation to each instance in a round-robin fashion until the aggregate throughput is above the target.
3. It places instances on servers using the first-fit decreasing heuristic, *i.e.,* places the largest instance first. If this algorithm succeeds, ResQ launches the instances each with the computed schedule.

The greedily generated schedule may be suboptimal because *(a)* it is online and incremental (does not move running instances), and *(b)* uses a heuristic to determine how many NF instances to run. To improve the placement efficiency, in the background, ResQ computes an optimal schedule. We formulate the placement problem as a mixed-integer linear program whose objective is to minimize the number of servers used (see Appendix A). We use a MILP solver [24] to compute the (near-)optimal schedule. In our experiments, the greedily and incrementally computed schedule's resource use is within 20% of the optimal one (see §6).

The solver typically finds near-optimal solution(s) for inputs which require a cluster size of around 40 servers in seconds to minutes. To scale to larger-sized clusters, we partition the SLOs into sets and pass each to a different solver. The computed schedules are instantiated on different slices of the cluster. This allows us to trade off computation time for schedule optimality.

The computed MILP-based schedule might be different than the running schedule that was greedily updated during admission. To converge to the new schedule various NFs must be migrated; this problem has been studied in the literature in the form of migration of stateful middleboxes or scaling out NFs [21, 51, 55]. This is likely an expensive and disruptive process, therefore we migrate only when the optimality gap is large enough.

Alternatively, a *migration avoidance* [47] strategy could be deployed to avoid the disruption or complexity of state migration. This involves booting up new instances but leaving old instances (that were to be terminated) running – the old instances will continue serving their traffic but no new traffic is directed to them. When their traffic eventually dies down they will be terminated. This strategy is only effective when sufficient spare capacity is available to bring up new instances without terminating the old ones.

[5]We rely on standard VM migration techniques.

5.3.2 On-Demand SLOs

Resource allocation for on-demand SLOs is jointly performed by the local resource scheduler and the centralized scheduler. The central scheduler is responsible for leasing dedicated cores and LLC space to local schedulers for scheduling on-demand SLOs, and for assigning new NF instances to servers with spare resources. Leases are dynamically adjusted when reserved SLOs arrive or leave the system – *i.e.,* beyond configured resources reserved for on-demand SLOs, the central scheduler may make spare resources available to local schedulers.

When computing LLC allocations for on-demand SLOs, NFs are placed in a shared LLC space or dedicated partitions based on whether or not their latency objectives can be met with sharing. Sharing LLC space (when possible) helps minimize the overhead of LLC partitioning since CAT allocates LLC space in fixed and relatively large increments. The NFs that require isolation are put in separate classes. If, despite isolation, the local scheduler fails to meet an NF's latency objective, it notifies the central scheduler which in turn adds more instances or resources for the failed SLO if possible.

The above-mentioned LLC allocation is computed as follows. First, all on-demand NFs are placed in a cache class that includes all the on-demand LLC space leased to the local scheduler. The server agent waits for a period of time for NFs to serve traffic before monitoring SLO violations and LLC occupancies (using Intel CMT – see §3.4) – occupancy measurements are used to estimate NFs' LLC demand. If one or more SLO violations are observed, the local scheduler continues with a max-min fair allocation of the LLC space. SLO-compliant NFs and SLO-violating NFs with low cache miss rates that use less than their fair share of LLC are put in a shared cache class with an allocation closest to the sum of their LLC occupancy. SLO-violating NFs with high cache miss rates that use less than their fair share of LLC are put in an isolated cache class with an allocation close to their fair share. The rest remain in the shared cache class whose size is reduced to the remaining on-demand LLC space. This procedure is repeated for the shared cache class to completion.

The server agent is responsible for monitoring on-demand NF instances for SLO violations. Such violations may occur when traffic pattern or NF configuration changes. Upon detection of a violation or change in the leased LLC space size, the local scheduler repeats the LLC allocation procedure to find whether it could meet the new demand with local resources and if not would notify the central scheduler of the new failed SLO.

6 Evaluation

In this section, we address the following questions:

Accuracy: To what extent do contention-agnostic schedulers violate SLOs? We compare against a simple bin-packing strategy adopted by current contention-agnostic schedulers [20, 45, 47]. We compare accuracy both without CAT and when we use CAT to evenly partition LLC across NFs.

Efficiency: We compare the efficiency of ResQ's online (greedy) and offline (mixed integer program based) schedulers against a prediction-based online scheduler [10] and an E2-like scheduler [47] which dynamically scales the number of NF instances in response to input traffic load.

To answer these questions, we generated three sets of reservation-based SLOs each with around 200 terms: one involving only the cache-sensitive NFs; one involving only the cache-insensitive NFs; and one involving a mixture of all the NFs. We set the target throughput and latency for each SLO to 90% and 100% of what a single instance of the corresponding NF can sustain when run in isolation without LLC contention. To avoid interfering with DDIO's reserved LLC space (10%), ResQ uses only 90% of the available LLC. To enable comparison with the fair allocation scheme, we use 9 cores per server so that each core can be allocated an equal cache partition (10%).

6.1 Accuracy

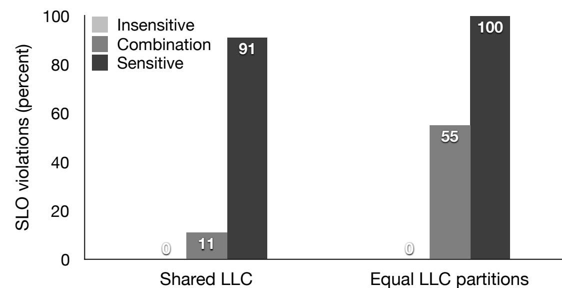


Figure 8: *SLO violations with > 5% error for contention-agnostic methods.* Contention-agnostic placement results in throughput and latency SLO violations. As expected, violations increase with the sensitivity of NFs. Naïvely partitioning LLC has an adverse effect.

If SLO violations were rare, it would be appealing to opt for a simpler contention-agnostic scheduler. To assess this choice, we evaluate the ability of current contention-agnostic schedulers to meet SLOs. To do so, we first run each NF on a dedicated server without restricting cache access to determine its throughput and latency. We then use this information to pack NF instances on the first available server. We show the results in Figure 8. Unsurprisingly, no SLO violation are observed for the cache-insensitive workload. However, SLO violations are common for combination (11%) and cache sensitive workloads (91%) workloads.

Next, to check whether a naïve cache isolation strategy is sufficient to reduce violations, we reran the same workload after using CAT to partition the LLC evenly between all NFs on a server. The number of violations worsened in this case: 55% of SLOs are violated in the combination workload, while all SLOs are violated for the cache-sensitive case. In the combination case, this difference is due to the

unavailability of the underutilized dedicated LLC space of the cache-insensitive NFs to the cache-sensitive ones.

Note that schedules computed by ResQ have no violations in all cases.

6.2 Efficiency

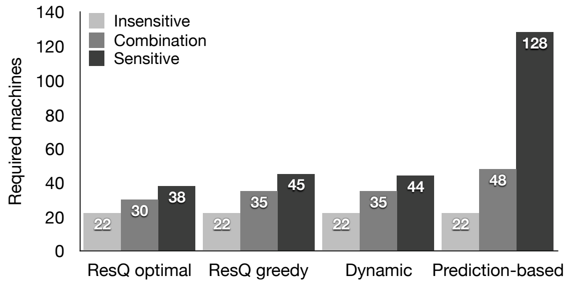


Figure 9: *Resource efficiency of different schemes and SLO mixes.* Not surprisingly, all methods are similar when NFs are cache-insensitive. The ResQ's greedy admission is within 19% of the optimal solution. The prediction-based scheme uses significantly more servers because it overestimates degradation.

ResQ builds on availability of a hardware mechanism (cache isolation) to provide predictable performance regardless of contention. Two alternative strategies for getting predictable performance involve: *(a)* online scheduling where one measures NF performance and dynamically allocates NF instances in response to SLO violations, and *(b)* using a performance predictor (*e.g.,* Dobrescu *et al.*'s predictor [10]) to predict throughput degradation due to resource sharing and using its result for scheduling. We analyze ResQ's efficiency in contrast to these options next.

ResQ's efficiency. For reserved SLOs, ResQ implements both an offline MILP-based scheduler that computes near optimal schedules and an online greedy scheduler. In Figure 9, we first evaluate the accuracy gap between these options. The optimal scheduler performs up to 19% better than the greedy scheduler. However, as previously noted in §5.3.1, the optimal scheduler may take much longer than the greedy scheduler, and ResQ can opportunistically move to using the optimal schedule if warranted.

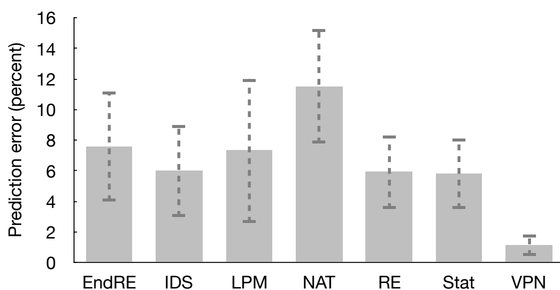


Figure 10: *Error of the throughput degradation prediction method [10].* We observe that errors are significantly higher using the current generation of hardware than what was previously observed. To follow the original setup, we use the following chains: EndRE is LPM → Stats → EndRE, VPN is LPM → Stats → IPsec, IDS is Snort, RE [58] is LPM → Stats → RE, STAT is LPM → Stats, NAT is MazuNAT, and IP is LPM.

Comparison with elastic scaling. Systems like E2 [47] continuously monitor NFs and dynamically add new instances if demand could not be met. A major drawback of this approach is that it cannot be used to enforce any *latency SLOs*. Despite being dynamic, this approach is not significantly more efficient than ResQ as seen in Figure 9. The dynamic approach uses the same number of instances for both cache-insensitive and combined traffic. It does provide a small savings of 1 machine (*i.e.,* a 2.2% improvement) for cache-sensitive workloads. However, this saving comes at the cost of no SLO isolation (variations in an NF's behavior may affect all its neighboring SLOs) and no latency guarantees.

Comparison with the predication-based approach. Finally, one could use a performance predictor to predict degradation due to resource sharing; this produces a safe schedule assuming the predictor never underestimates degradation. Dobrescu *et al.* [10] previously proposed such a predictor. Their predictor works as follows. Each NF is profiled using a series of synthetic benchmarks with tunable pressure on the LLC. The result is a curve which one can use to determine throughput as a function of competing LLC references. The competing LLC references are approximated by counting the LLC references of NFs' solo runs. This method was reported as being very accurate in 2012.

To study its robustness against significant hardware and software changes, we reran the experiments on our testbed using similar NFs and setup (6 competing NFs and 19.5 MB of LLC). Figure 10 shows the average prediction error in percentage points. Each bar shows the difference between predicted and observed performance drop suffered by a target NF when sharing a processor with 5 identical competing application instances (9 different sets of NFs for each NF) similarly to their choice of competitors. We find that this predictor is conservative and consistently overestimates degradation by a large margin. Consequently, it can be used to enforce (throughput) SLOs albeit not efficiently.

We use this predictor to build an online first-fit bin-packing scheduler. The scheduler packs an instance on the first server whose existing SLOs do not get affected by the new instance; it proceeds to pack a second instance if the predicted throughput is below the target throughput. We ran all the computed placements and recorded the *real* throughput and latency to assess SLO compliance. All schedules remain SLO compliant regardless of cache sensitivity except the prediction-based scheduler that violates 0.5% latency SLOs in the combination case. This is not surprising because this method does not predict latency degradation.

In Figure 9, we compare the efficiency of ResQ with the prediction-based method – by efficiency, we mean the number of CPUs (equivalent to servers for single-CPU servers) each scheduler needs to satisfy its SLOs. With

cache-insensitive NFs, all schedulers need the minimum number of CPUs because consolidation does not affect the performance. As expected, the efficiency gap widens as the cache sensitivity of the mix of SLO NFs increases. The gap between the prediction-based scheduler and ResQ's greedy scheduler increases from 37.1% to 184.4%.

The are two reasons for such a sharp increase in resource usage for the prediction-based scheduler: *(a)* overly conservative performance estimate results in more false positives (mispredicting violations), and *(b)* lack of a mechanism to predict how much traffic an NF can handle without SLO violations. The gap in the sensitive case is due to the latter reason: individual servers have spare capacity but the scheduler cannot use any because an NF serving maximum traffic will violate the existing NFs' SLOs, but what if it only serves 20% of its capacity? These issues aside, scheduling is much simpler in ResQ because isolation is enforced by hardware regardless of contention.

Based on this result, we conclude that ResQ's simple first-fit bin-packing heuristic using CAT (online admission) is effective in maintaining a resource efficient and SLO-compliant schedule, while there is opportunity to further optimize this schedule by periodically running a slow offline scheduler.

7 Related Work

Performance modeling. Prior work [5, 9, 23, 40, 60] has investigated modeling and predicting the effect of resource contention in the context of HPC and datacenter applications. These models are often useful in contexts where the relative performance of two settings needs to be compared, *e.g.,* when scheduling or placing jobs. However, they are not accurate enough for our purposes. Dobrescu *et al.* [10] have proposed using cache references as a predictor of throughput with contending processes. While this was highly accurate given the hardware and software stacks available at the time, we find that it consistently underestimate throughput (§6.2) in today's systems. We showed that a scheduler using this predictor may consume up to 3× more resources compared to ResQ (§6.2). Moreover, this work on prediction models still leaves open the question of *enforcement* wherein an NF that deviates from its predicted behavior (whether due to malicious behavior, configuration changes, or varying traffic) can impact the performance of its neighboring NFs.

Performance isolation. Packet processing and NFV platforms [28, 41, 47, 49] do not isolate NFs from contending on uncore resources. Such systems can be extended to use CAT to provide performance isolation. Our contribution lies in showing how cache isolation can be used to both provide performance isolation and guarantee SLOs. Other systems that provide end-to-end performance guarantees for multi-tenant networks [2, 39, 56] treat CPUs as independent resource units and do not account for interference across cores. DRFQ [22] models a packet-processing platform as a pipeline of resources where each packet is sequentially processed by each resource. DRFQ's primary goal is to provide per-flow fairness while we focus on SLO guarantees. Ginseng [19] presents an auction-based LLC allocation mechanism, but does not offer SLOs. Heracles [37] uses CAT and other mechanisms to co-locate batch and latency-sensitive jobs while maintaining millisecond time-scale latency SLOs; we target more aggressive latency SLOs (high throughput, microsecond scale).

Mechanisms. The mechanisms and use cases of cache partitioning have been studied in the past [8, 34, 35]. A rich body of literature looks at software-only methods for cache isolation [14, 26, 59, 64]. Their performance implications have not been studied in the NFV context but they may be used as an alternative to CAT when hardware support is not available or more granular allocations are desired. A recent work [63] has also briefly looked at the benefit of using CAT to alleviate a specific instance of the noisy neighbor problem. It focuses on a single workload and demonstrates that, in one specific case, CAT notably improves performance in presence of a noisy neighbor problem. By contrast our work is general (covering a wide range of NFs and workloads), identifies cases where CAT alone does not sufficiently isolate NFs, and develops a contention-aware scheduler that uses our isolation mechanism to provide SLO guarantees for NFs.

8 Concluding Remarks

Despite no algorithmic innovation, ResQ's simple greedy scheduler achieves a significantly higher resource efficiency than prior prediction-based methods and its efficiency is on-par with elastic schedulers that do not guarantee SLOs. Moreover, despite its hardness, ResQ's MILP formulation yields (near-)optimal schedules in a matter of seconds to minutes. These advances were all made possible because we identified a technique – building on hardware cache isolation and proper buffer management – that ensures strong performance isolation regardless of noisy neighbors. ResQ is open source and available at `https://github.com/netsys/resq`.

Acknowledgement

We would like to thank Andrew Herdrich, Edwin Verplanke, Priya Autee, Christian Maciocco, Charlie Tai, Rich Uhlig, Michael Alan Chang, Yashar Ganjali, David Lie, Hans-Arno Jacobsen, our shepherd Tim Wood, and the NSDI reviewers for their comments and suggestions. This work was funded in part by NSF-1553747, NSF-1704941, and Intel corporation.

References

[1] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. EndRE: An End-system Redundancy Elimination Service for Enterprises. In *NSDI*, 2010.

[2] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*, 2014.

[3] H. Basilier, M. Darula, and J. Wilke. Virtualizing Network Services- The Telecom Cloud. Ericsson Review, 2014. URL: `http://tinyurl.com/j5adfts`.

[4] Y. Beyene, M. Faloutsos, and H. V. Madhyastha. SyFi: A Systematic Approach for Estimating Stateful Firewall Performance. In *PAM*, 2012.

[5] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *USENIX ATC*, 2011.

[6] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633.

[7] Broadband Forum. TR-178: Multi-service Broadband Network Architecture and Nodal Requirements, 2014. URL: `http://tinyurl.com/z7vkk6h`.

[8] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness. In *ISCA*, 2013.

[9] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *ASPLOS*, 2013.

[10] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-processing Platforms. In *NSDI*, 2012.

[11] T. L. K. Documentation. Reducing OS Jitter Due to Per-CPU kthreads. URL: `http://tinyurl.com/mpnf4m3`.

[12] Data Plane Development Kit (DPDK), 2015. URL: `http://dpdk.org/`.

[13] DPDK Performance Tuning Guide, 2016. URL: `http://tinyurl.com/jkngtok`.

[14] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems. In *ASPLOS*, 2010.

[15] S. Ehlert, G. Zhang, and T. Magedanz. Increasing SIP firewall performance by ruleset size limitation. In *PIMRC*, 2008.

[16] Emerging Threats. Emerging Threats Open Rulesets, 2016. URL: `http://tinyurl.com/nppr7ut`.

[17] The Evolved Packet Core. URL: `http://tinyurl.com/hvkukyw`.

[18] ETSI. Network Functions Virtualisation. URL: `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`.

[19] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. Ginseng: Market-Driven LLC Allocation. In *USENIX ATC*, 2016.

[20] A. Gember-Jacobson, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *CoRR*, abs/1305.0209, 2013.

[21] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *SIGCOMM*, 2014.

[22] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource Fair Queueing for Packet Processing. In *SIGCOMM*, 2012.

[23] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *SOCC*, 2011.

[24] Gurobi Optimization, Inc. Gurobi Optimizer Reference Manual, 2015. URL: `http://www.gurobi.com`.

[25] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, 2015.

[26] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based QoS techniques for cache/memory in CMP platforms. In *ICS*, 2009.

[27] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5-2600 v3 Product Family. In *HPCA*, 2016.

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

[29] Intel® Xeon® Processor E5 and E7 v4 Families Uncore Performance Monitoring, 2016. URL: `http://tinyurl.com/zpsj63k`.

[30] Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family, 2016. URL: `http://tinyurl.com/hasjlm2`.

[31] Intel® Data Direct I/O (DDIO), 2014. URL: `http://tinyurl.com/jlkzvll`.

[32] Intel® Memory Latency Checker, 2015. URL: `http://tinyurl.com/kgroxnw`.

[33] I. L. A. Division. PCI-SIG SR-IOV Primer, 2011. URL: http://tinyurl.com/kt7bwqb.

[34] R. Iyer. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *ICS*, 2004.

[35] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *SIGMETRICS*, 2007.

[36] Linux Foundation. OPNFV, 2016. URL: https://www.opnfv.org/.

[37] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, 2015.

[38] D. Lopez. OpenMANO: The Dataplane Ready Open Source NFV MANO Stack. In *IETF Meeting Proceedings, Dallas, Texas, USA*, 2015.

[39] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *NSDI*, 2015.

[40] 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 *MICRO*, 2011.

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

[42] Performance Tuning for Mellanox Adapters. URL: http://tinyurl.com/y8slm66k.

[43] T. P. Morgan. ARM Servers: Qualcomm is Now a Contender. https://www.nextplatform.com/2017/08/23/arm-servers-qualcomm-now-contender/, 2017.

[44] Nokia. Solutions: Residential Services Delivery, 2016. URL: http://tinyurl.com/h3cwqsy.

[45] T. L. Foundation. ONAP: Open Network Automation Platform. https://www.onap.org/ retrieved 09/21/2017.

[46] Open Information Security Foundation. Suricata: Open Source IDS/IPS/NSM engine, 2015. URL: http://suricata-ids.org/.

[47] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *SOSP*, 2015.

[48] P. A. Networks. PA-3000 Series Datasheet. https://www.paloaltonetworks.com/products/secure-the-network/next-generation-firewall/pa-3000-series retrieved 09/21/2017.

[49] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *OSDI*, 2016.

[50] O. Peleg, A. Morrison, B. Serebrin, and D. Tsafrir. Utilizing the IOMMU Scalably. In *USENIX ATC*, 2015.

[51] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *NSDI*, 2013.

[52] L. Rizzo. Revisiting Network I/O APIs: The Netmap Framework. *ACM Queue*, 10(1), 2012.

[53] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *LISA*, 1999.

[54] S. Blake and D. Black and M. Carlson and E. Davies and Z. Wang and W. Weiss. An Architecture for Differentiated Services. RFC 2475, 1998.

[55] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-Recovery for Middleboxes. In *SIGCOMM*, 2015.

[56] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *OSDI*, 2012.

[57] Sourcefire's Vulnerability Research Team. VRT Rule Set, 2015. URL: https://www.snort.org/talos.

[58] N. T. Spring and D. Wetherall. A Protocol-independent Technique for Eliminating Redundant Network Traffic. In *SIGCOMM*, 2000.

[59] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *ASPLOS*, 2009.

[60] L. Tang, J. Mars, and M. L. Soffa. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers. In *CGO*, 2012.

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

[62] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing attacks: improve your cloud performance (at your neighbor's expense). In *CCS*, 2012.

[63] P. Veitch, E. Curley, and T. Kantecki. Performance evaluation of cache allocation technology for NFV noisy neighbor mitigation. *NetSoft*, 2017.

[64] X. Zhang, S. Dwarkadas, and K. Shen. Towards Practical Page Coloring-based Multicore Cache Management. In *EuroSys*, 2009.

A MILP Formulation

When a new reserved SLO is submitted, ResQ profiles the given NF (or chain) and, if admissible, greedily schedules one or more instances of it. Periodically, ResQ looks for a more optimal schedule to switch to if this results in significant resource savings. We formulate this optimal scheduling as a mixed-integer linear program.

Symbol	Type	Description
θ_i	Constant	Target throughput of SLO i
π_{il}	Constant	Pivot point of piece l of SLO i
α_{il}	Constant	Slope of piece l of SLO i
β_{il}	Constant	Y-intercept of piece l of SLO i
τ	Constant	Number of cores per machine
φ	Constant	LLC size per machine
U_i	Constant	Maximum input load of SLO i
I_{ijk}	Binary Var	Instance j of SLO i is assigned to machine k
N_k	Binary Var	Machine k is active
C_{ij}	Integer Var	LLC allocated to instance j of SLO i
λ_{ijl}	Binary Var	Piece l of instance j of SLO i is used

Table 3: *List of symbols used in MILP.* We use indices i, j, k, l for SLO terms, instances, machines, and profiles' linear fit pieces respectively. The number of variables and constants depend on the size of the cluster, number of SLO terms, maximum number of instances per SLO term, and number of pieces of individual profiles.

The objective of the MILP in Listing 1 is to minimize the number of machines used to satisfy all the SLO terms. As input, it expects system configuration and profiles, and produces a schedule as output. For each SLO term, this schedule provides the number of instances to start, where each instance should be placed, and the amount of LLC allocated to each instance. We encode the SLO profiles in the form of piecewise linear approximations of their throughput-LLC curves.

$$\min \sum_k N_k$$

$$\text{s.t.} \quad \sum_k I_{ijk} \leq 1 \qquad \forall i,j \quad (1)$$

$$\sum_{i,j} C_{ij} . I_{ijk} \leq \varphi \qquad \forall k \quad (2)$$

$$N_k \leq \sum_{i,j} I_{ijk} \leq N_k . \tau \qquad \forall k \quad (3)$$

$$\theta_i \leq \sum_j [\mathrm{U}_i . \sum_l \lambda_{ijl} . [\alpha_{il} . C_{ij} + \beta_{il}]] \qquad \forall i \quad (4)$$

$$\sum_l \lambda_{ijl} . \pi_{il} \leq C_{ij} \leq \sum_l \lambda_{ij(l+1)} . \pi_{i(l+1)} \qquad \forall i,j \quad (5)$$

$$\sum_l \lambda_{ijl} = \sum_k I_{ijk} \qquad \forall i,j \quad (6)$$

Listing 1: *Mixed-integer linear program that minimizes the number of machines used to meet reserved SLOs in ResQ.* A brief description of the symbols appear in Table 3.

We use a set of variables to capture the scheduling results and constants to encode the system configuration and profiles:

θ_i specifies the target throughput for SLO term i.

π_{il} specifies the pivot point for piece l of the throughput-LLC linear approximation of SLO term i.

α_{il}, β_{il} specify the slope and y-intercept for piece l of the throughput-LLC linear approximation of SLO term i.

τ, φ specify the number of cores and LLC size available on each machine.

U_i is the maximum input load level that below which the latency objective of SLO term i is satisfied across all LLC allocations.

I_{ijk} indicates whether instance j of SLO term i is active on machine k.

N_k is set if and only if machine k is active – *i.e.,* at least one instance is assigned to it.

C_{ij} indicates the amount of LLC allocated to instance j of SLO term i. For an active instance, each such variable takes a value between the minimum and maximum permissible LLC allocation.

λ_{ijl} indicates whether linear fit l is chosen for instance j of SLO term i.

Below we briefly describe the goal of each constraint in the order they appear in Listing 1:

1. An instance runs on at most one machine.
2. The total LLC allocated to instances assigned to a machine is less than or equal to the machine's total LLC size (φ).
3. A machine is active when there is at least one instance running on that machine, and an active machine may host no more instances than its available cores (τ).
4. The aggregate throughput of instances of each SLO is greater than or equal to its target throughput (θ_i).
5. Linear piece l of a profile is chosen if and only if the LLC allocated to instance j of SLO term i lies in the range corresponding to piece l of the throughput-LLC linear approximation.
6. Exactly one linear piece is chosen when instance j of SLO term i is active, otherwise, none is chosen.

For simplicity, we assume a homogeneous infrastructure and that each SLO term instance requires a single CPU core; the MILP could be adjusted to account for differences if necessary. To account for small performance degradation despite ResQ's isolation (see §4.4), we include a 3% discount in U_i.

Elastic Scaling of Stateful Network Functions

Shinae Woo[⋆†], *Justine Sherry*[‡], *Sangjin Han*[⋆], *Sue Moon*[†], *Sylvia Ratnasamy*[⋆], *and Scott Shenker*[⋆§]

[⋆]*University of California, Berkeley* [†]*KAIST* [‡]*CMU* [§]*ICSI*

Abstract

Elastic scaling is a central promise of NFV but has been hard to realize in practice. The difficulty arises because most Network Functions (NFs) are *stateful* and this state need to be *shared* across NF instances. Implementing state sharing while meeting the throughput and latency requirements placed on NFs is challenging and, to date, no solution exists that meets NFV's performance goals for the full spectrum of NFs.

S6 is a new framework that supports elastic scaling of NFs without compromising performance. Its design builds on the insight that a distributed shared state abstraction is well-suited to the NFV context. We organize state as a distributed shared object (DSO) space and extend the DSO concept with techniques designed to meet the need for elasticity and high-performance in NFV workloads. S6 simplifies development: NF writers program with no awareness of how state is distributed and shared. Instead, S6 transparently migrates state and handles accesses to shared state. In our evaluation, compared to recent solutions for dynamic scaling of NFs, S6 improves performance by 100x during scaling events [25], and by 2-5x under normal operation [27].

1 Introduction

The Network Function Virtualization (NFV) [13] vision advocates moving middlebox functionality – called Network Functions (NFs) – from dedicated hardware devices to software applications that run in VMs or containers on shared server hardware. An important benefit of the NFV vision is elastic scaling — the ability to increase or decrease the number of VMs/containers currently devoted to a particular NF, in response to changes in offered load. However, realizing such elastic scaling has proven challenging and solutions to date come with a significant cost to performance, functionality, and/or ease of development (§3).

The difficulty arises in that most NFs are *stateful*, with state that may be read or updated very frequently (e.g., per-packet or per-flow). Hence, elastic scaling requires more than simply spinning up another VM/container and updating a load-balancer to send some portion of the traffic to it.

Instead, scaling can involve *migrating* state across NF instances. Migration is important for high performance (as it avoids remote state accesses) but its implementation must be fast (to avoid long "pause times" during scaling events) and should not be burdensome to NF developers. In addition, elastic scaling must ensure *affinity* between packets and their state (*i.e.*, that a packet is directed to the NF instance that holds the state necessary to process that packet), and such affinity must be correctly enforced even in the face of state migrations. A final complication is that some types of state are not partitionable, but *shared* across instances (see §2 for examples). In such cases, elastic scaling must support access to shared state in a manner that ensures the consistency requirements of that state are met, and with minimal disruption to NF throughput and latency.

The core of any elastic scaling solution is how state is organized and abstracted to NF applications. Recent work has explored different options in this regard. Some [33] assume that all state is local, but neither shared or migrated – we call this the *local-only* approach. Others [25,37] support a richer model in which state is exposed to NF developers as either local or remote, and developers can migrate state from remote to local storage, or explicitly access remote state – we call this the *local+remote* approach. Still others [27] assume that *all* state is remote, stored in a centralized store – we call this the *remote-only* approach.

The above were pioneering efforts in exploring the design space for NF state management. But, as we elaborate on in §3, they still fall short of an ideal solution: the *local-only* approach achieves high performance but is limited in the NF functionality that it supports; the *local+remote* approach supports arbitrary NF functionality but complicates NF development and incurs long downtimes from repartitioning state en bloc during scaling events; the *remote-only* approach is elegant but imposes high performance overheads even under normal operation.

In this paper, we propose a new approach to elastic scaling in which state is organized as a *distributed shared object* (DSO) space: objects encapsulate NF state and live in a global namespace, where all NF instances can read/write any object. While DSO is an old idea, it has not to our knowledge been applied to the NFV context. In particular, DSO has not been shown to the meet the elasticity and performance requirements that NFV imposes.

We present S6, a development and runtime framework tailored to NFV. To meet the needs of NFV workloads, S6 extends the DSO concept as follows: (1) for space elasticity, we introduce dynamic reorganization of the DSO keyspace; (2) to minimize the downtime associated with scaling events, we introduce a "smart but lazy" state reorganization; (3) to reduce remote access overheads, we introduce per-packet microthreads and; (4) to optimize

performance without burdening developers, we expose per-object hints via which the developer can inform the DSO framework about appropriate migration or caching policies. S6 hides all those internal complexities of distributed state management under the hood, simplifying NF development.

We present three elastic NFs implemented on top of S6: NAT, PRADS (a network monitoring system [6]), and a subset of the Snort IDS [8]. We show that NFs on S6 elastically scale with minimal performance overhead, and compare them to NFs built using prior approaches. A local-only system like E2 [33] cannot support two of our use-cases (NAT and PRADS) because it does not support shared state. Compared to OpenNF [25], a state-of-the-art framework based on the local+remote approach, S6 achieves 10x - 100x lower latency while sustaining 10x higher throughput during scaling events. Compared to StatelessNF [27], a state-of-the-art framework based on a remote-only approach, S6 achieves 2x - 5x higher throughput under normal operation.

2 Background: NF State Abstractions

The difficulty of elastic scaling arises in how to handle NF state appropriately. NFs keep state about ongoing connections (*e.g.*, TCP connection state, last activity time, number of bytes per user-device, a list of protocols used at a given IP address). NFs read and update a state while processing a packet and reuse the updated state to process subsequent packets. Stateful NF instances should maintain correct NF state collectively to prevent the inconsistent or incorrect behavior. Also, the system must handle general forms of state sharing among instances, as we will present in this Section. Such NFs' behavior requires significantly more care than merely spinning up another NF instance and sending some portion of traffic to it.

We categorize NF state based on whether it is *partitionable*. We say that state is partitionable if it can be distributed across NF instances in a way that state is only locally accessed, assuming a certain traffic load balancing scheme. For example, per-flow state (such as state for individual TCP connections) is partitionable, if traffic is distributed on a flow basis. On the other hand, a counter for the total number of active flows is an example of non-partitionable state, since all NF instances need to update the counter.

Whether state is partitionable or not is important since it determines both the mechanisms needed to manage that state and the achievable performance levels. With partitionable state, we can collocate state with the NF instance that processes it, and hence efficient state migration is key to achieving high performance. If state is not partitionable, high performance requires a different set of techniques: *e.g.*, caching state (when its consistency semantics allows it), placing state to minimize remote accesses, and minimizing the cost of remote state access.

Apps \ State	Partitionable	Non-partitionable
NAT	-	Address mapping entry Available address pool
Firewall	Connection context	-
Load balancer [11, 21]	Connection - server mapping	Server pool usage statistics
Traffic Monitoring [6]	Connection context	Per-host context; Statistics for packets, used protocols and host
IDS/IPS [8, 10, 34]	Connection context	A set of certificates, malicious servers, or infected hosts; Per-host port scanning counter
Web proxy [9]	Connection context	Statistics for cached entry
EPC [3]	User state	SLA/Usage per device/plan
IMS [5]	SIP / RTP sessions	Usage accounting per user

Table 1: Examples of state in popular NF types

Most non-partitionable NF state also provide opportunities for efficient sharing. We can categorize state by whether it is updated mostly by a single or multiple instances. From our observation, single-writer state tends to be read-heavy, thus caching or replication can be effective. When the state is updated by multiple writers simultaneously (*e.g.*, global counters), looser consistency is often tolerable so as to trade freshness of data for performance.

Table 1 lists examples of partitionable and non-partitionable state found in some real-world NFs. We see that both forms of state are common in real-world NFs. For example, traffic monitoring systems [6] maintain state at both the connection (partitionable) and the host (non-partitionable[1]) levels. We also note that state variables, whether partitionable or not, typically relate to each other forming complex data structures. For example, traffic monitoring systems manage a global table of hosts, each referencing a list of its active connections.

3 State Management for Elastic Scaling

State management for elastic scaling of stateful NFs involves many design options, such as where to place state and when to initiate migration. They all affect the overall NF performance, in terms of throughput and latency. Existing approaches cause high performance overhead, either during scaling events (*i.e.*, instances join and leave) or under normal operations (*i.e.*, no ongoing scaling events). In this section, we discuss the limitations of their approaches in §3.1 and propose our new approach in §3.2.

3.1 Limitations of existing approaches

Figure 1 shows the typical components of an NFV architecture as assumed by prior research [27, 27, 33, 37] and industry efforts [13]. An NFV controller [24, 33] manages NF instances that run on servers, while an SDN

[1]No load balancing scheme can ensure data locality of state for both source and destination hosts at the same time.

controller manages the network fabric between servers, including how traffic is load-balanced across the different NF instances. Responsibility of NF state management—initiating migration, placement, etc.—resides in the NF controller. We now discuss existing approaches to NF state management.

Local-only state: Early work on NFV management [22, 24, 33, 35, 38] typically assumed that NF state is partitionable and hence they do not address the issue of shared state. Some (*e.g.*, E2 [33]) address elastic scaling but for NFs with per-flow state only. In such cases, state migration is avoided by directing only new flows to new NF instances. Thus, these early systems do not accommodate NFs with shared or more general (*i.e.*, beyond per-flow) state, which is common in practice (Table 1). Hence, we will not consider solutions based on a "local-only" model further in this paper.[2]

Remote-only state: In this model, all NF state is kept in a standalone centralized store. First proposed by Kablan *et al.* [27], the elegance of this model is that support for state sharing, consistency management, and durability (of state) moves to the centralized store, while the NF instances themselves are stateless and hence can be easily added or removed. StatelessNF [27] uses this approach to build various NFs such as a NAT, Firewall and TCP reassembler.

Unfortunately, this approach comes at the cost of performance. In statelessNF, *all* state accesses are remote: not only do these remote accesses inflate packet latency, they also consume extra CPU cycles and network bandwidth for I/O to the remote store. Our results in §7.2 show that, relative to an NF that uses local state, a remote-only approach can lead to a 2-3x degradation in throughput and a 100-fold increase in packet latency. Problematically, these overheads are incurred even under normal operation (*i.e.*, in the absence of scaling events) and grow with the number of state accesses and the size of messages. Recent work reports that such state accesses are frequent in NFs: *e.g.*, StateAnalyzr [28] reports that typical NFs maintain 10-100s of state variables that are per-flow or shared across flows.

Local+Remote In this model, state is distributed across NFs and exposed to NF developers as either local or remote. All NF state is defined (by the developer) to be either local or remote, and is accessed accordingly. OpenNF [25] and SplitMerge [37] adopts this model. For high performance, partitionable state is typically defined as local state (similar to local-only NFs) while non-partitionable state requires explicit push/pull function calls to synchronize with remote state.

[2]We note that systems such as E2 could be augmented with the state management capabilities that we and others [25, 27, 37] propose.

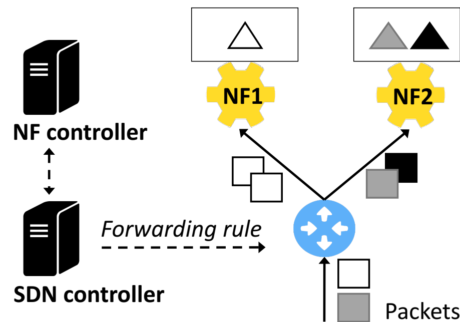


Figure 1: Typical components of an NFV architecture

In this model, state management plays two roles: (1) implementing access to remote state, (2) migrating state upon scaling events so as to not break local memory accesses. This functionality is implemented by a state management framework (such as OpenNF) working in concert with the NFs and SDN controllers. Under normal operation, the local+remote approach has the potential to achieve throughput and latency comparable to the previous models by migrating state to be co-located with the NF instances that access it. Unfortunately, the overhead *during* scaling events is high in this model. The reason stems from the fact that state is explicitly defined and accessed as local or remote. When a new NF instance is launched, all state that may be accessed as local state at the new NF instance must be migrated over to it *before* any access occurs (since otherwise the local access would simply fail).

Scaling events thus result in "stop the world" behavior, which involves the following steps: first, the SDN controller buffers traffic destined for both the old and new instance, by rerouting traffic from the fabric/load-balancer to itself; next, the state management controller coordinates the migration of relevant state from the old to new NF instance; once migration completes, the SDN controller releases buffered traffic and coordinates with the fabric/load-balancer to turn off detouring traffic to the SDN controller. This approach can lead to long pause times during which both old and new NF instances stop processing packets while state is repartitioned. This is also complex to implement due to tight coordination among the SDN controller, NF controller, and the inline switches/load-balancer.[3]

As we show in §7.1, the local+remote approach lead to very long pause times for practical NFs. For example, PRADS implemented on OpenNF incurs a pause time of **490 ms** when migrating only 1,500 flows despite extensive optimization to the process. In practice, the pause time is likely to be even higher considering that a typical 10 Gbps link has tens of thousands of concurrent flows [43].

[3]A subtle additional challenge is that these components often come from different vendors, complicating the adoption of such techniques.

3.2 Our approach: Distributed shared state

The limitations of the aforementioned approaches lead us to consider a new approach, the distributed shared state model, which is familiar from distributed computing. Here, state is distributed among the NFs, and can be accessed by any NF. However, the NF developer makes no distinction between local vs. remote state. Instead, all state variables reside in a shared address space and the state management framework transparently resolves all state access. The framework is also responsible for deciding where state is placed and migrating state across NF instances when appropriate.

Done right, this model can achieve throughput and latency comparable to the local-only model by migrating state to be co-located with the NF instances that access it. This model can also avoid the long pause-times incurred by the local+remote approach. Because there is no distinction between local and remote state, no proactive migration is required during scaling events. The overhead of migration is gradually amortized as packets arrive; *e.g.*, for the same PRADS scenario above, the pause-time can drop to under 1 millisecond (§7). For the same reason, state migration no longer needs tight coordination with traffic load-balancing, hence reducing the system complexity associated with the local+remote model. Finally, the distributed shared state model simplifies NF development. Developers can write NFs on top of a uniform interface for state access, local or remote, outsourcing the underlying details of state lookup, remote access, and migration to the state management framework.

To the best of our knowledge, we are the first to apply distributed shared state to NFV. We highlight two challenges distinct from other application domains. NFs have distinct performance requirements from traditional cloud applications [18, 31]. Furthermore, elasticity makes it more difficult, dynamically reorganizing the structure of state space into the new set of NF instances.

Achieving high performance: NFs have I/O intensive workload, requiring very high throughput on the order of millions of packets/s with sub-millisecond latency. Given these requirements, only a few hundreds or thousands of CPU cycles are available for every packet.

The key to achieving high performance is twofold. Firstly, we should reduce the number of remote accesses by leveraging the state-instance affinity and supporting efficient sharing. As each type of NF state has different access patterns and consistency requirements [28], the question is how to leverage the information while minimizing developer's burden. Secondly, we need to minimize the cost of remote access when it is unavoidable. While we can hide the latency by processing other packets in the meantime, it must be done so without increasing programming complexity. The framework should be able to handle data dependency detection and context stashing [15, 42].

Supporting elastic scaling: As explained above, scaling events at runtime must not incur significant performance degradation. Membership change in NF instance group involves two potential sources of service disruption. First, as input traffic is distributed across the new set of NF instances, a subset of state variables must migrate to maintain locality. Second, in addition to the state variables themselves, their location metadata must be reorganized as well, for scalability of the shared state space. The challenge is how to perform these operations in a distributed fashion, in order to avoid a single point of performance bottleneck. Furthermore, the framework must ensure consistent state access during the process, while minimizing delay in packet processing.

4 S6 Design

S6 is a development and runtime framework for elastic scaling of NFs. S6 makes the following assumptions, which are general enough to apply to a wide variety of deployment scenarios and environments. First, an NF runs as a cluster of virtualized instances, such as VMs and containers. Second, the network somehow distributes input traffic across instances. Lastly, an external NFV controller/orchestrator triggers scaling events to adapt to load change.

S6 does not demand any particular network load balancing mechanism or NFV controller behavior for correctness. Therefore they are out of the scope of this paper. One desirable property is that input traffic be distributed across instances on a flow basis as like most of load-balancers and switches already are doing, so that S6 can leverage the state-instance affinity for high performance. S6 differs from the existing NF state management solutions [25, 37], all of which require sophisticated runtime coordination across NFV controller, SDN controller, and NF instances. S6's decoupling from the load-balancer and SDN controller reduces system complexity.

We summarize the main design components: 1) S6 provides the global DSO shared by all NF instances. We choose 'object' as a basic unit of state. An object encapsulates a set of data and its associated operations, allowing access control and integrity protection of state. All objects in the space are accessible with a uniform API, regardless of where the objects physically reside. 2) Our object abstractions provide NF developers with knobs to specify object access patterns. The S6 framework uses this information to improve performance by reducing the number of remote state accesses. 3) When remote state access is inevitable, S6 mitigates its cost by hiding latency with microthreads; NF worker instances can keep processing other flows only if they have no data dependency on outstanding accesses to remote objects.

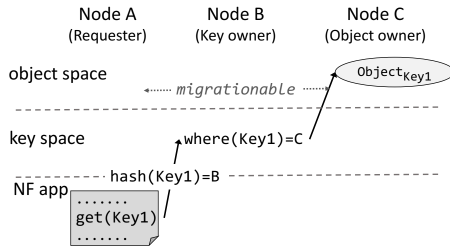


Figure 2: DHT-based Distributed Shared Object Space

4) Upon scaling events, S6 reorganizes the space, while keeping the workers processing traffic. S6 minimizes service disruption with the smart but lazy migration of objects and their metadata. We explain each component in greater detail below.

4.1 DHT-based DSO space architecture

In the DSO space, state objects are uniquely identified with a *key*. Keys can be of any type, such as 5-tuples, host names, and URLs as necessary. When an NF instance requests an object with some key (*e.g.*, extracted from packets), S6 returns a *reference* to the object, rather than the object binary itself. With the reference, the instance can read or update the state by invoking object methods. S6 constructs the DSO space as a DHT-based two-layer structure (Figure 2) of *a key layer* and *an object layer* [40]. Both layers are distributed over NF instances. The key layer keeps track of the current location of every object, rather than directly storing objects. This layer of indirection offers great flexibility in object migration; no broadcast is necessary to locate an object, although it may reside (if it exists) on any instance at the moment. The object layer stores the actual binary of objects. A reference to an object guarantees accessibility, no matter where the object currently is.

When an instance accesses an object for the first time, it hashes the key to identify the *key owner*, the instance who knows the current object location, the *object owner*. The object access request is sent to the key owner first, then the key owner forwards the request to the object owner. Once the location of the object is resolved, the instance caches it so that subsequent object requests can go directly to the object owner. When an object migrates to another instance, the key owner must be notified. The key owner updates the location of the object and invalidates the cached location in workers.

The key owner takes charge of object creation, deletion, and its reference creation; Those requests are serialized and sequentially processed at the key owner. Once the key owner receives object deletion request, it rejects subsequence object access requests until new object creation request comes.

Note that this two-layer structure is only internally managed. S6 hides the complexity of placing and locating objects from NF developers, so that they can focus on the application logic itself.

4.2 Object abstractions: Per-object optimization

We provide an object abstraction that allows developers to hint to the framework about what caching, migration, and optimization strategies are appropriate for each object. Different objects hence have different consistency guarantees depending on their usage. While state management is a generic problem in distributed systems, we focus on NFs' distinct state characteristics and access patterns that we can leverage to achieve good performance.

We first categorize object types into two types depending on whether the object permits updates from multiple flows (thus multiple instances): partitionable objects and non-partitionable objects. Based on the different characteristics for each state type in §2, we introduce appropriate optimization strategies for each object type. APIs in detail and usage examples are in covered in §5.

Partitionable: leveraging state-instance affinity Partitionable objects are primarily used for state that is updated by a single flow; up to one writable reference to the object is allowed. If an instance is holding the writable reference, other instances have to wait for their turn to acquire a writable reference. This is enforced at the key owner since it is a natural serialization point for all reference requests to the object.

In NF contexts, while partitionable objects have high affinity on a single instance, but occasionally, its state affinity may move to other instances. For example, per-flow objects' affinity is decided based on the traffic load-balancing policy of the network, which is not controlled by S6. As NFs frequently update partitionable state, often on a per-packet basis, keeping high state-instance affinity is the key to achieving high performance for partitionable objects.

Partitionable objects are gradually migrated between instances when affinity changes. S6 uses a new request for writeable access from other instances as an affinity change indicator. When a key owner for a partitionable object gets an object access request other than the current object owner, it initiates the object migration process. The current object owner voluntarily releases the reference when the local reference count for the object reaches to zero, then the object is transferred, and the instance becomes a new object owner. Now in the new object owner, all accesses to the object locally happens as the reference points to the object binary in the memory.

Non-partitionable: consistency/performance trading Non-partitionable objects are concurrently accessed from multiple flows simultaneously; multiple writable references to an object may exist. Supporting shared state with high performance in distributed systems is generally difficult or impossible to achieve—if an object is very frequently updated by multiple flows in a strongly-consistent manner, it does not scale and S6 cannot help it. Fortu-

nately, we found that the majority of non-partitionable state in NF applications does not require frequent updates (*e.g.*, one update per flow) or allow trading consistency for performance, as shown in Table 1. S6 provides three optimization mechanisms for non-partitionable objects that NF developers can leverage.

First, S6 supports object access via method call shipping, rather than migrating objects. This prevents objects from bouncing between instances, to avoid wasting in-flight time. The method calls are applied serially in the order of request arrival at the object owner, preserving the internal object consistency. For such non-partitionable objects, NF developers need to design the objects to be commutative, *i.e.*, any order of methods calls should produce an acceptable result. Additionally, S6 supports *untethered update*, which allows remote object update without blocking if there is no need to wait for its completion.

Second, S6 supports abstractions to design objects to enable trade-off between consistency and performance. If a read method on an object class is tolerant of *stale* results, instances can cache the results of the method locally. NF developers can bound the staleness for each read method, so that S6 can periodically refresh the cached results with newer ones.

Third, S6 supports object replication, so that multiple instances can update their local replica. Those replicas are regularly *merged* to the main object at the object owner. NF developers are expected to provide this merge function since it is very object-specific. Shared counters are a prime use case of this local update and merge. Frequent updates are done locally, while (infrequent) reads on the counter cause local numbers to be merged globally.

4.3 Microthreads: Hiding the cost of remote access

Even with above optimizations for objects, blocking remote access is necessary when waiting for migrating objects, refreshing the cache, or dealing with objects with strong consistency. The cost of remote state access is high. Suppose that an NF instance issues an RPC request and waits for its response, in order to process a packet. Assuming 10us round-trip time between NF instances, the latency translates to 30,000 cycles on a 3 GHz processor. We can hide this latency with concurrency; the NF can process other pending packets to keep the CPU busy, as long as they do not have data dependency on the RPC or introduce packet reordering in a flow. Once its response arrives, the NF continues processing the packet(s) that were blocked on it.

We adopt a multi-threaded architecture in favor of ease of NF development to maintain execution contexts of blocked flows. The other option was an event-driven architecture, but it hurts programmability since developers must manually manage to save and restore contexts [15, 42] for every state access. Another issue is that whether a method call would block or not must be visible to the NF developers, which adds additional complexity to the application logic. In contrast, with multi-threaded architecture, developers can program packet processing easily while all thread scheduling is automatically done by the S6 runtime.

To minimize the performance overhead of multi-threading, S6 utilizes cooperative, user-space "microthreads". User-level microthreads are much more lightweight than kernel threads, since non-preemptive scheduling is significantly simpler, and context switching does not involve kernel/user boundary crossing. It also scales up to millions (not thousands) of microthreads thanks for their small footprint.

S6 manages a pool of microthreads to avoid thread construction/destruction cost. A microthread runs for each received packet. Whenever the thread is about to block (*e.g.*, an object is remote and/or in migration, cache entry is being refreshed, data dependency is detected as another microthreads is holding a reference, etc.), the microthread yields to other pending threads and wait to be rescheduled after the blocking condition has been resolved. This non-preemptive scheduling is automatically done by S6 and transparent to the NF developer. When multiple microthreads are ready to resume, S6 schedules one with the longest wait time to avoid packet reordering within a flow and to minimize latency jitter.

4.4 Smart but lazy DSO keyspace reorganization

When the membership of NF instances changes—due to scaling events or node failures—S6 must reorganize the DSO space for the new set of instances. This reorganization involves both object space and key space. As we illustrated in §4.2, the object space is repartitioned automatically and gradually for new state-instance affinity, as NF instances access state objects. Assuming reference locality—most state access is done to a small number of objects—frequently accessed objects are quickly migrated to new object owners, incurring minimal performance impact.

On the other hand, like the object space, S6 ensures that the key space reorganization is also done gradually so as to minimize performance impact. Suppose that we reorganize the DSO key space from S_i to S_{i+1}, which use $h_i(key)$ and $h_{i+1}(key)$ as lookup hashes for finding key owner respectively. Reorganization must not break the coherency of the keyspace, such that any key record is neither lost nor owned by multiple key owners. At the same time, we do not want to pause the entire system for coherency; instead NF instances lazily migrate key ownership from $h_i(k)$ to $h_{i+i}(k)$ in the background as necessary. Our keyspace reorganization algorithm ensures coherency even in the middle of scaling process.

When the scaling process starts, new key access requests go to $h_{i+i}(k)$. The new owner $h_{i+1}(k)$ check if the previous owner $h_i(k)$ has a record for k, and if so, the new key owner pulls the record. During the scaling process, every new key lookup requires two-hop routing. After the keyspace converges to S_{i+1} (all key record migration is completed), key lookups can be done with the normal one-hop routing again.

Dealing with race conditions: One challenge comes from the fact that we cannot assume that all nodes start and finish the scaling process exactly at the same time. For example, if two nodes A (previous owner of k, $h_i(k)$) have not been notified the scaling process and run in 'normal' operation but B (next owner of k, $h_{i+1}(k)$) start 'scaling' operation, both two nodes would claim the ownership for k. This corner case may result in two key records for k created in both node A and B.

To prevent such conflict, S6 performs scaling in two stages: *pre-scaling* and *scaling* stages. The workers transition to scaling stage only if the controller has confirmed that all workers are in pre-scaling state. This barrier ensures that nodes in the 'normal' and nodes in the 'scaling' stage do not coexist. Nodes in pre-scaling stage do not actively transfer key ownership yet, while being aware that other nodes may be in scaling process. Ensuring that there is always a single key owner exists, but not two or none, is done with the following rules:

R1 Preventing double ownership *Suppose that node A (prev owner, $h_i(k)$) is in 'pre-scaling' and node B (next owner, $h_{i+1}(k)$) is in 'scaling'. In this case, $h_i(k)$ should be the single owner for k.*

Since pre-scaling nodes can coexist with nodes still in 'normal' stage, node A should serve $h_i(k)$. Meanwhile, node B $h_{i+i}(k)$ have more contexts about scaling process than A. Until node A goes into 'scaling' stage, it defers claiming the ownership for k, but keeps forwarding requests to A.

R2 Preventing lost ownership *Suppose that node A (prev owner, $h_i(k)$) is in 'scaling' and node B (next owner, $h_{i+1}(k)$) is in 'pre-scaling'. If k is for a new object, $h_{i+1}(k)$ should be the single owner. If k is for an existing object, $h_i(k)$ should be the single owner.*

In this case, no one claim the ownership of k, and the two node forward requests on k to each other. We need to prevent such loop. Let's assume that A receives a request on k. If k is for existing objects and A owns the key since B hasn't claimed the ownership. A keeps serving the requests on k, until B claims ownership of k. If k is for new objects, then B doesn't have any information of k. Therefore B would forward the request to A. A potential loop is prevented by attaching version number to the forwarded requests.

Category	API	Description
Object	`SingleWriter`	Exclusive writeable
	`MultiWriter`	Concurrent writeable
Method	`const stale`	Cached read
	`untethered`	Untethered update
	`merge(Object&)`	Merge two objects
Data Structure	`S6Map<Key, Object>`	Define a map in DSO
	`S6Ref<Object>`	Reference to an object
	`S6Iter<Object>`	Iterator of collections
S6Map (DSO)	`create(Key&,Flag&)`	Create an object
	`get(Key&)`	Retrieve an object
	`remove(S6Ref<>&)`	Remove an object

Table 2: S6 Programming API

State Type	Examples	Object Annotation	Method Annotation
Partitioned	UDP/TCP connection state	`SingleWriter`	-
Non-partitioned freq update	Performance statistics	`MultiWriter`	`untethered` `stale` `merge`
Non-partitioned read-heavy	NAT mapping entry	`SingleWriter`	`stale`
Collection of multi-type state objects	linked-list hashtable	Non-intrusive data structures (§8.1)	

Table 3: Common types of NF state and their annotations

5 Using S6

We introduce our programming model (§5.1) and provide some examples of various NFs (§ 5.2).

5.1 S6 Programming model

Table 2 summarizes the S6 API. From a user's perspective, S6's core components are the *shared object space* and *tasks*.

We provide two types of objects depending on whether the object permits update from multiple writers (NF instances). `SingleWriter` allows exclusive writes from a single instance. `MultiWriter` allows concurrent writes from multiple instances simultaneously. Methods on objects can be annotated appropriately to allows more optimization such as cached read (`const stale`), update-and-forget (`untethered`), or regularly pushing merged local updates (`merge`) into the object owner. Then, S6 supports appropriate optimization on behind based on object type as explained in previous section §4.2. Figure 3 shows an example implementation of an object class used in PRADS [6]. It is exactly same as normal object oriented design only except the additional annotations we introduce. In fact, from our survey of popular NFs in Table 1, we found that most of NF state falls into one of four types shown in Table 3.

S6 provides two types of tasks: data-plane and control-plane. *Data-plane tasks* perform packet processing on input network traffic. *Control-plane tasks* perform out-of-band operations, such as updating configurations

```
class HostAsset: public MultiWriter {
  public:
    void update_service(Service s) untether;
    void update_os(OS os) untether;
    uint64_t first_detect_time() const stale;
    uint64_t last_detect_time() const stale;
    void merge(HostAsset local);
  private:
    addr_t ip;
    uint64_t first_detect_time;
    uint64_t last_detect_time;
    List<Service> service_list;
    List<OS> os_list;
};
```

Figure 3: A sample S6 object definition of PRADS's per-host network asset object

```
S6Map<IPKey, Asset> g_asset;
S6Map<FlowKey, Connection> g_conn;

// data-plane task
FlowKey fkey(sip, dip, sport, dport);
S6Ref<Connection> c = g_conn.create(fkey);
    ...
if (new_os_asset) {
    S6Ref<Asset> asset = g_asset.get(sip);
    asset->update_os_asset(new_os_asset);
}
...

// control-plane task
S6Iter<Asset> *it = g_asset.get_iterator();
while (it->next())
    log_asset(it->key,  it->value);
```

Figure 4: A sample implementation of PRADS tasks

or processing user queries. Both types of tasks have access to the shared object space with a uniform interface. Figure 4 shows an application implementation including one data-plane task for packet processing and one control-plane task for logging the per-host assets.

5.2 Programming NFs

5.2.1 Sample applications

We have chosen various applications to implement or port. Table 4 lists the state objects in those NFs.

Network Monitoring System (PRADS) PRADS [6] is a *Passive Real-time Asset Detection System* in Linux. It allows network administrators to access real-time data on types of protocols, services, and devices on their network.

Intrusion Detection System (Snort-rule) We implement IDS which monitors packets using Snort [8] rules. We borrow the rule compilation and detection code from the original Snort code base.

NAT We implement NAT (Network Address Translator) by following the algorithm described in statelessNF [27], so that have the same per-packet/per-flow access patterns with their implementation.

NF	State	Size (B)*	Update	Access Frequency
PRADS	Flow	160	Exclusive	Per-packet RW
	Statistics	208	Concurrent	Per-packet RW
	Asset	$112+64n$	Concurrent	Rarely R Per-packet W
	Hashtable of flows	$40n$	Concurrent	Per-flow RW
	Hashtable of assets	$32n$	Concurrent	Per-flow RW
IDS	Flow context	$160 \sim 32$k	Exclusive	Per-packet RW
	Whitelisted host	16	Exclusive	Per-packet RW
	Malicious server	$12+28n$	Concurrent	Per-flow RW
	Hash table of Malicious server	$32n$	Concurrent	Per-flow RW
	Hash table of whitelisted host	$32n$	Concurrent	Per-flow RW
NAT	Address Pool	8k per IP	Exclusive	Per-flow RW
	NAT entry	8	Exclusive	Per-packet R Per-flow W

* n is the number of elements in the structure.

Table 4: States, update patterns, and access frequencies of NF applications we use.

5.2.2 Experiences of porting NF applications

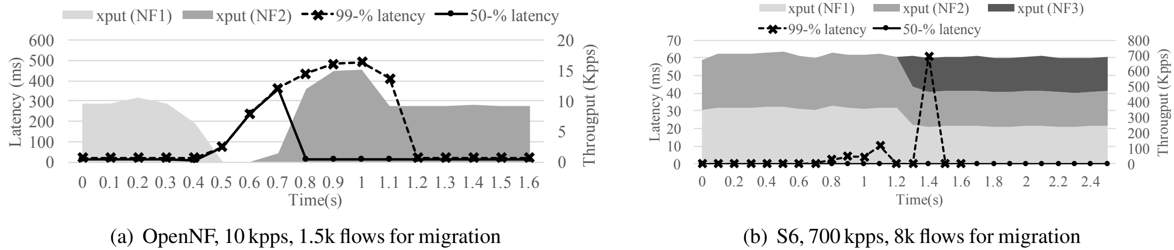
We begin with the assumption that the NF application to port is in an OOP (Object-Oriented Programming) model. Since the baseline code of PRADS is in C, a non-OOP language, our first step is to convert `structs` to C++ objects. Then we start porting these objects in our S6 programming interface.

Porting States Objects: To convert the existing object classes to S6-compatible object classes, we need to (1) identify globally accessible objects, (2) analyze their update patterns, and (3) check the applicability of loose consistency.

In Table 4 we list the states we have identified to be globally accessible and their update patterns: four simple objects (Flow, Statistics, Asset, and Configuration) and two collections of objects (Flow hashtable and Asset hashtable). After identifying the simple objects, we decide their types as `SingleWriter` or `MultiWriter` according to the update pattern. We use `S6Map` to support hash tables for flows and assets, and `S6Iter` to iterate through the list of assets. In case of more complex applications, StateAlyzr [28] can help identifying state variables which need to be shared and their update patterns.

Now the application is compatible with S6 and should run correctly. Next, we turn to performance improvement by loosening the consistency level on objects. We design the Asset and Statistics objects to be commutative, and all their reads as cached reads and all updates as untethered.

Porting Tasks: PRADS has a simple loop processing packets using `libpcap` which is straightforward to port to S6's data plane task. PRADS has other out-of-band tasks from network administrators like generating a log of current assets. We implement these out-of-band operations as control-plane tasks.



(a) OpenNF, 10 kpps, 1.5k flows for migration

(b) S6, 700 kpps, 8k flows for migration

Figure 5: Performance comparison with PRADS scale-out events

6 Implementation

Our implementation of S6 has three main components: the S6 runtime, the S6 compiler, and the NFV controller.

Runtime The S6 runtime plays three roles. First, it manages the DSO space distributed across nodes. It tracks objects location and controls object accesses from multiple instances to support exclusive or concurrent accesses per object type. Second, it manages S6-compatible object references to provide the S6 programming interface. The S6 runtime intermediates every access to objects and performs remote operations or initiate object migration if necessary. Third, it schedules microthreads for data-plane and control-plane tasks. Whenever a microthread is about to block due to remote access, the runtime schedules another pending microthread and continues to process packets without blocking. We use the boost co-routine library [12] to implement non-preemptive, user-level multi-threads.

S6 Compiler While S6 requires a custom programming interface in the source code, there is no convenient way to extend C++ syntax. Instead, we implemented a source-to-source compiler, which translates S6-extended C++ code into plain C++. The generated code abstracts away implementation details of DSO implementation under the hood. For example, when a method is invoked, the generated code checks if the state object is local or remote to call appropriate functions. We implement the compiler on top of clang-3.6 library [14] to perform syntax analysis of a developer's code.

NFV Controller We built a simple NFV controller to manage the S6 instances. It runs an NF cluster by launching S6 instances and initiates scaling-in/out events based on network workloads. The controller also relays out-of-band tasks such as queries or updating configuration from the operators to NF instances.

7 Evaluation

We start our evaluation of S6 with its application-level performance with the scale-out NFs we ported in §5.2. We examine how scaling events impact S6 performance during scaling events in §7.1, and under normal operation in §7.2. Then we show the effectiveness of design choices in S6 with a series of micro-benchmarks in §7.3

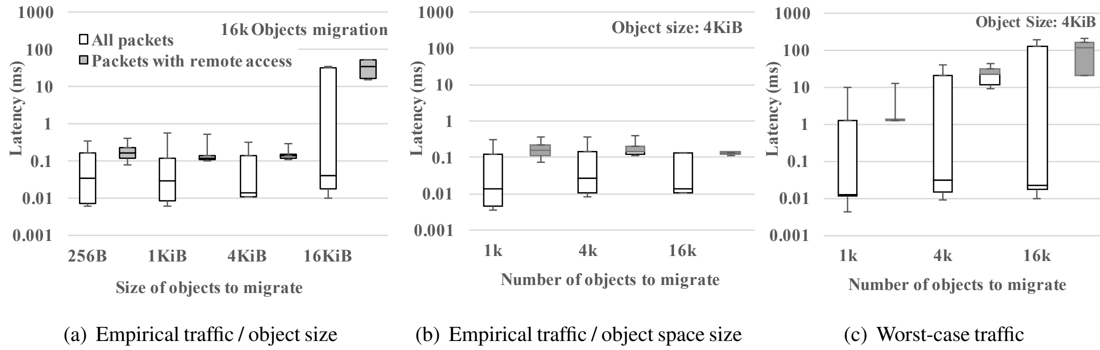
Evaluation setup We use Amazon EC2 c4.xlarge instances (4 cores @ 2.90GHz) for experiments. NF instances run as a Docker container, across the virtual machines in the cluster. Our workload is synthetic TCP traffic based on empirical flow distributions in size and arrival rates. For all experiments shown, we measure the overall throughput and latency measured at input/output ports of each NF. For micro-benchmarks, we use a dedicated Intel Xeon E5-2670 (2×8 cores @ 2.30GHz) server, with a 10 G link for data channel and another 10 G for state channel for inter-instance communication.

7.1 Elastic Scaling

How well S6 performs during scaling events? We compare S6's scaling-out performance with OpenNF using PRADS on each framework. Figure 5(a) shows the throughput and latency of migrating 1.5k flows at 10 kpps workloads using OpenNF. Even with the highest optimization level OpenNF supports, the throughput drops and the latency increases up to hundreds of milliseconds. Not shown, we tested the exact same workload with S6. S6 shows no visible throughput fluctuation and only a few hundreds of *microseconds* increase in latency.

Figure 5(b) shows PRADS scale-out performance on S6 *with a higher input workload*, 700 kpps with 8k concurrent flows. There is not any noticeable throughput degradation (and also zero packet loss). The state channel becomes temporarily congested from object migration, key-space re-partitioning in addition to the shared variable accesses. Still, the peak latency around tens of milliseconds during scaling is transient—within a 0.1 second window—and 10x lower while sustaining 10x higher throughput than OpenNF.

How does workload affect performance during scaling events? We now consider S6 scaling for a synthetic NF, in which we can configure the number of state objects and their size. We send 1 Mpps network load to a single NF instance. Then we initiate a scale-out event, launching another instance and split the traffic. As a result, half of the objects in the original instance move to the new instance as packets arrive. We vary the number of objects and the object size and measure the end-to-end packet processing latency.



(a) Empirical traffic / object size (b) Empirical traffic / object space size (c) Worst-case traffic

Figure 6: Latency (1-25-50-75-99%-iles) during object re-partitioning

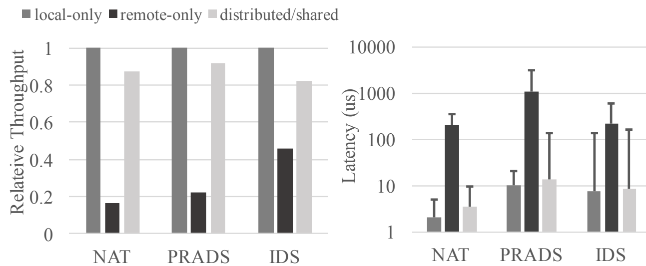


Figure 7: Performance of NFs implemented on different NF state management abstractions

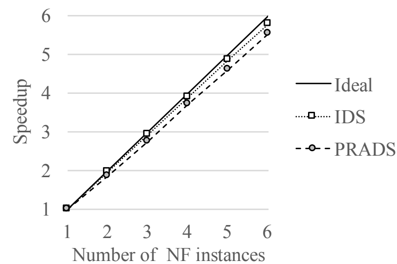


Figure 8: S6 throughput scalability

Figures 6 (a) and (b) show latency during the scaling process. The scaling process takes 100s-1,000s of milliseconds depending on workloads. As shown in the graph, the peak latency is sub-millisecond except for 16 KiB (a) objects, as the state channel to transfer objects among instances gets congested. However, even with the state channel being a bottleneck the peak latency is temporary (re-partitioning ends within a second), and median latency remains under a millisecond. The tail latency comes from packets accessing objects remotely due to gradual migration, and subsequent accesses become local without incurring network round-trips.

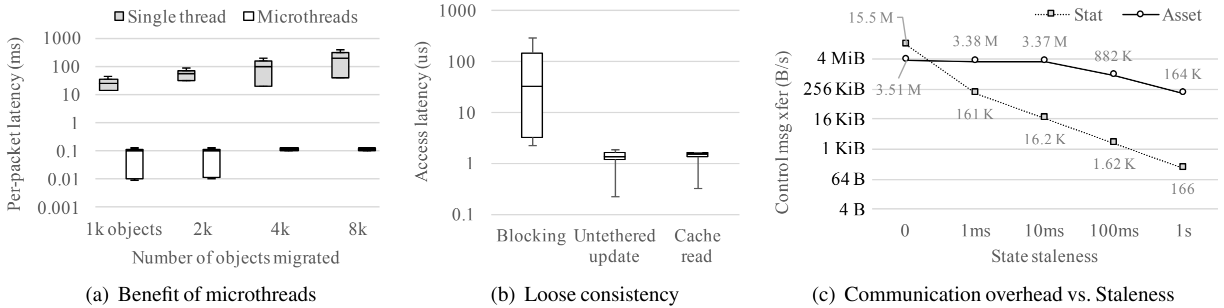
We also evaluate a worst-case scenario with bursty object migration. We generate traffic in a round-robin fashion (*i.e.*, packets are generated sequentially from the flow pool) so that all per-flow state objects migrate back-to-back. Figure 6(c) shows that the peak latency increases as more objects migrate and the state channel becomes more congested. Similarly to the previous graphs, the peak latency lasts only for 500 ms, and the median latency stays under a millisecond.

S6 creates user-level microthreads for non-blocking object migration and key space re-partitioning. The overhead of microthreads was very lightweight for all cases; while S6 can manage up to millions of microthreads, much less is necessary in practice. The maximum number of concurrent microthreads (not shown in the graphs) during migration was about 30k for the 16 Kib case, or a few thousands for other cases.

7.2 Normal Operation

How does S6 compare to existing approaches? We compare S6's performance against the remote-only and local-only options discussed in §3. The local-only model serves as an idealized scenario, as the absence of remote access overhead represents a performance overhead. Since it does not support shared state, we instead replicate non-partitionable state across all instances, thus resulting in incorrect NF behavior. For the remote-only design, we consider StatelessNF [27] as state-of-the-art. While its source code is not publicly available, we implement the algorithms as presented in the paper, including the performance optimization techniques. Our remote-only test uses one remote store and an NF instance; our "distributed/shared" test (S6) uses two NF instances but measures the throughput of only one instance.

Figure 7 shows the throughput and latency of each implementation. The remote-only shows 2-5x lower throughput and 10-100x times higher latency than the ideal (local-only) case since it requires multiple remote state accesses per packet. Another overhead we observed is that depending on workloads and NF types, the state channel (for communication between NF instances and the remote storage) may become more congested than the data channel itself, even with the applications described in the StatelessNF paper. In S6 the only remote access is the first access of a migrating flow context, and all subsequent accesses are local. S6 shows 82-92% of the throughput and comparable latency to the local-only, an ideal case.



(a) Benefit of microthreads (b) Loose consistency (c) Communication overhead vs. Staleness

Figure 9: (a) Per-packet latency during scale-out with and without microthreading, (b) Read/Write latency for remote access (1s maximum staleness), (c) Tradeoff between communication channel overhead vs. accuracy of state

How scalable is S6 with the number of instances? Figure 8 plots the speedup of PRADS and IDS on S6, relative to the baseline throughput of a single instance. The actual aggregated throughput is within 2-8% of ideal linear speedup. We observe little impact on packet processing latency with increasing number of instances.

7.3 Micro-benchmarks

How much do microthreads improve latency relative to a single-threaded approach? In this experiment, we quantify the benefit of microthreading that masks the cost of remote state access on Figure 9(a). To start, we run a single instance which owns all per-flow objects with 1 Mpps input load. Then we split the traffic between two instances, triggering state migration of half of the objects to the second instance. Here we see that the microthreaded architecture improves latency by over three orders of magnitude. Microthreads efficiently pipeline processing packets as they are blocked for object migration. Also, since most packets are processed locally, the number of outstanding microthreads remains small: 130-160 during our experiments.

How much does annotation-based optimization improve state access latency? We compare the performance of different remote access mechanisms in Figure 9(b): 1) blocking RPC, 2) untethered update, and 3) cached read. We run two S6 instances, and 16k shared objects are evenly distributed between two instances. Each instance randomly accesses one of the 16k objects. Thus half of the accesses are local, and the other half are remote. The results show that the latency of untethered updates and cached read is only a few microsecond, since state access can be done with local memory reads/writes; actual synchronization happens in the background. However, in the case of blocking RPC, remote access adds one network round-trip latency for remote objects.

How much does caching reduce communication channel overhead? As we discusses in §4.2, with commutative updates of shared state, we can lower communication channel overhead by allowing bounded staleness. Figure 9(b) quantifies the trade-off, with two different types of shared objects in PRADS. In the case of Stat(istics), a single object is shared by all instances, and every packet triggers at least three updates on it. As we allow more staleness, the required communication channel bandwidth decreases proportionally.

On the other hand, commutativity is not always effective when compared with per-update RPCs. In the case of Assets, State objects—per-host assets—are only shared among flows originated from or destined to the same host. Since updates to an object are not very frequent, periodic synchronization performs no better than individual updates. As shown in the graph, staleness less than 100 ms does not lower the communication channel overhead.

8 Discussion

8.1 State beyond objects

Collections Many NF applications include collection data structures (e.g., linked list, tree, or hash tables). In S6, one can build such collections as non-intrusive containers of references of objects like C++ STL [39]. In non-intrusive data structures, objects do not need to have a special pointer for the container to be a member of it (*e.g.*, a pointer for the next element in list), but the container organizes data structures using references of the objects. One can also specially design a collection structure for efficient concurrent accesses (e.g., RCU [30]).

Framework-level supports on collections will have more opportunities to exploit better locality on its elements. We implements hashtable and read-only iterator on it and leave more framework-level support for data structures as future work.

Multi-object transactions S6 natively supports linearizability – ordering amongst writes to a single object, but not support serializability – ordering with regard to multiple objects. To support multi-object transactions, NF developers can implement a custom lock with exclusively update-able objects. Only a single instance is allowed to have a reference to the lock object; the other instances need to wait until the reference is released from the previous instance. The key owner serializes accesses to the lock object.

8.2 Fault-tolerance

Fault tolerance for middleboxes and network functions has been addressed by prior systems like Pico [36] and FTMB [39]. These systems promise that when an NF fail in a non-scaled out environment, a new NF quickly come back online – with all of the state of the failed NF – and resume processing data.

The most straightforward remediation is to adopt Pico's (checkpoint-based, per-state snapshot) or FTMB's (checkpoint and replay-based, VM snapshot) algorithms at on per node basis. Both systems interpose on accesses to middlebox state during packet processing; these systems also have the ability to interpose on accesses made by S6's RPC calls from other NF instances. Both Pico and FTMB have efficient backup strategies, in that one 'backup' instance can serve as a standby for multiple 'hot' NF instances. S6's knowledge about object access patterns and consistency gives more opportunity to optimize per-object snapshot, balancing between snapshot frequency and amount of logging operations on it.

An alternative approach to fault tolerance could be to extend S6's state management with classic DHT-based failover recovery. Key ownership—and perhaps even data itself—could be replicated thrice across multiple DHT nodes. Hence, if any individual node failed, the rest of the cluster could immediately continue processing incoming flows, accessing the remaining replicated state. Nonetheless, this approach triples intra-cluster traffic, and likely increases read/write latencies. We leave exploration of this approach, its design details, and trade-offs, to future work.

9 Related Work

In §3, we have discussed E2 [33], Split/Merge [37], StatelessNF [27], and OpenNF [25]. We do not revisit them here. There are some specific (not general) NF implementations that internally support horizontal scaling including Maglev (load-balancer) [21], Protego (IPSec gateway) [41], and Bro Cluster [34]. These systems leverage each NF-specific techniques, but cannot be generalized for other types of NFs.

The design and implementation of S6 are heavily inspired by previous work. There are many systems adopting the concepts of DSO to build distributed systems. RPC frameworks such as (CORBA [1], DCOM [2], and RMI [4] provide state access in a uniform manner across heterogeneous languages and software. Thor [17] is a distributed database system that takes care of object distribution, sharing, and caching. Fabric [29] is a distributed application building framework, which focuses on guaranteeing information security among distrust users. All of above systems provide uniform access to objects distributed across nodes, guarantee object consistency, and provide high availability. Yet, none of the above focuses on supporting high-performance requirements such for NFs and elastically adjusting the number of instances on the cluster with minimal interrupts. S6 extends the DSO to support elastic scaling and optimal performance both for under normal operations and during scaling events. We also acknowledge that use of lightweight multi-threading for masking remote access latency can be found in other application domains, *e.g.*, distributed graph processing [31].

Distributed shared state can exist at different levels of abstraction from low-level memory to a higher-level object-oriented model. Distributed key-value stores provide a wider range of state abstractions such as blobs [19, 23, 32] and abstracted data types [7], with properties from ACID to eventual consistency [19]. Partitioned Global Address Space (PGAS) allows multiple machines to share the same virtual address space for their physical memory [16, 20, 26]. This abstraction is useful in supporting machine-level optimizations (*e.g.*, dirty page tracking [16], RDMA [20]) but is too low-level for our context. A single page may contain multiple state variables each with different affinity or consistency semantics, making it impossible to migrate state for optimal state-operation affinity. We choose objects to abstract state, because it allows easy to program various requirements of objects; it is easy to program integrity and control accesses to its encapsulated set of data.

10 Conclusion

We presented S6, a framework for building elastic scaling of NF. S6 extends the DSO model to support elastic scaling of NFs without compromising performance, while the object abstraction transparently hides the complex details of data locality, consistency, and marshaling. S6 introduces a various mean to meet the performance requirements of NFs: "smart but lazy" reorganization of DSO space to minimize the performance overhead during scaling events; micro-threaded architecture to mitigate remote access latency; and programming model to trade performance with freshness per object requirements. Compared to previous work, S6 shows minimal performance overhead during scaling events (10-100x than OpenNF [25]) as well as during normal operations (2-5x than StatelessNF [27]. Our code is available at `https://github.com/NetSys/S6`.

11 Acknowledgments

We thank our shepherd Timothy Roscoe and the anonymous reviewers for their invaluable comments. We also thank Aurojit Panda for the enjoyable discussions and feedback, Keunhong Lee and Junmin Choe for their help on evaluations. This work was funded in part by NRF-2014R1A2A1A01007580, NSF-1553747, NSF-1704941 and Intel corporation.

References

[1] CORBA. http://www.corba.org/.

[2] DCOM. https://msdn.microsoft.com/en-us/library/cc226801.aspx.

[3] Evolved Core Network Implementation of OpenAirInterface. https://gitlab.eurecom.fr/oai/openair-cn.

[4] JavaRMI. http://www.oracle.com/technetwork/articles/javaee/index-jsp-136424.html.

[5] openIMS. http://www.openimscore.org/.

[6] PRADS. http://manpages.ubuntu.com/manpages/wily/man1/prads.1.html.

[7] Redis. https://redis.io/.

[8] Snort++. https://www.snort.org/snort3.

[9] Squid. http://www.squid-cache.org/.

[10] Suricata. https://suricata-ids.org/l.

[11] The Software Load Balancer and Dynamic ADC. http://inlab.de/load-balancer/index.html.

[12] BOOST Coroutine. http://www.boost.org/doc/libs/1_60_0/libs/coroutine/doc/html/index.html, accessed 8 May, 2016.

[13] ETSI NFV. http://www.etsi.org/technologies-clusters/technologies/nfv, accessed April 28, 2016.

[14] ETSI NFV. http://clang.llvm.org/, accessed April 28, 2016.

[15] ADYA, A., HOWELL, J., THEIMER, M., BOLOSKY, W. J., AND DOUCEUR, J. R. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track* (2002), pp. 289–302.

[16] CHAPMAN, B., CURTIS, T., POPHALE, S., POOLE, S., KUEHN, J., KOELBEL, C., AND SMITH, L. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (2010), ACM, p. 2.

[17] DAY, M., LISKOV, B., MAHESHWARI, U., AND MYERS, A. C. References to remote mobile objects in Thor. *ACM Letters on Programming Languages and Systems (LOPLAS) 2*, 1-4 (1993), 115–126.

[18] DEAN, J., AND GHEMAWAT, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[19] 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. *ACM SIGOPS operating systems review 41*, 6 (2007), 205–220.

[20] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (2014), pp. 401–414.

[21] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), pp. 523–535.

[22] FAYAZBAKHSH, S. K., SEKAR, V., YU, M., AND MOGUL, J. C. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013), ACM, pp. 19–24.

[23] FITZPATRICK, B. Distributed caching with memcached. *Linux journal 2004*, 124 (2004), 5.

[24] GEMBER, A., KRISHNAMURTHY, A., JOHN, S. S., GRANDL, R., GAO, X., ANAND, A., BENSON, T., AKELLA, A., AND SEKAR, V. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. *CoRR abs/1305.0209* (2013).

[25] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 163–174.

[26] HOEFLER, T., DINAN, J., THAKUR, R., BARRETT, B., BALAJI, P., GROPP, W., AND UNDERWOOD, K. Remote memory access programming in mpi-3. *ACM Transactions on Parallel Computing 2*, 2 (2015), 9.

[27] KABLAN, M., ALSUDAIS, A., KELLER, E., AND LE, F. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association.

[28] KHALID, J., GEMBER-JACOBSON, A., MICHAEL, R., ABHASHKUMAR, A., AND AKELLA, A. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 239–253.

[29] LIU, J., GEORGE, M. D., VIKRAM, K., QI, X., WAYE, L., AND MYERS, A. C. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 321–334.

[30] 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.

[31] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Confercorence (USENIX ATC 15)* (2015), pp. 291–305.

[32] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., ET AL. The case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review 43*, 4 (2010), 92–105.

[33] PALKAR, S., LAN, C., HAN, S., JANG, K., PANDA, A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 121–136.

[34] PAXSON, V. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks 31*, 23-24 (1999), 2435–2463.

[35] QAZI, Z. A., TU, C.-C., CHIANG, L., MIAO, R., SEKAR, V., AND YU, M. Simple-fying middlebox policy enforcement using sdn. *ACM SIGCOMM computer communication review 43*, 4 (2013), 27–38.

[36] RAJAGOPALAN, S., WILLIAMS, D., AND JAMJOOM, H. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 1:1–1:15.

[37] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 227–240.

[38] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M. K., AND SHI, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 24–24.

[39] SHERRY, J., GAO, P. X., BASU, S., PANDA, A., KRISHNAMURTHY, A., MACIOCCO, C., MANESH, M., MARTINS, J. A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 227–240.

[40] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for Internet Applications. *ACM SIGCOMM Computer Communication Review 31*, 4 (2001), 149–160.

[41] TAN, K., WANG, P., GAN, Z., AND MOON, S. Protego: Cloud-scale multitenant ipsec gateway.

[42] VON BEHREN, R., CONDIT, J., AND BREWER, E. Why Events Are a Bad Idea (for High-concurrency Servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Berkeley, CA, USA, 2003), HOTOS'03, USENIX Association, pp. 4–4.

[43] WOO, S., JEONG, E., PARK, S., LEE, J., IHM, S., AND PARK, K. Comparison of Caching Strategies in Modern Cellular Backhaul Networks. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2013), MobiSys '13, ACM, pp. 319–332.

Iron: Isolating Network-based CPU in Container Environments

Junaid Khalid[†] Eric Rozner[*] Wesley Felter[*] Cong Xu[*]
Karthick Rajamani[*] Alexandre Ferreira[‡] Aditya Akella[†]
[†]*UW-Madison* [*]*IBM Research* [‡]*Arm Research*

Abstract

Containers are quickly increasing in popularity as the mechanism to deploy computation in the cloud. In order to provide consistent and reliable performance, cloud providers must ensure containers cannot adversely interfere with one another. Because containers share the same underlying OS, it is more challenging to provide isolation in a container-based framework than a traditional VM-based framework. And while many schemes can isolate CPU, memory, disk, or network bandwidth in multi-tenant environments, less attention has been paid to how the time spent *processing* network traffic affects isolation on the host server. This paper shows computational overhead associated with the network stack can break isolation in container-based environments. Specifically, a container with heavy network traffic can decrease the computation available to other containers sharing the same server. We propose a scheme, called Iron, that accounts for the time spent in the networking stack on behalf of a container and ensures this processing cannot adversely impact colocated containers through novel enforcement mechanisms. Our results show Iron effectively provides isolation under realistic and adversarial conditions, limiting interference-based slowdowns as high as 6× to less than 5%.

1 Introduction

Today, containers are widely deployed in virtualized environments. Companies such as IBM, Google, Microsoft, and Amazon allow customers to deploy applications and services in containers via public clouds. In addition, serverless computing platforms [8,31,55] rely on containers to deploy user code [38]. Because containers share components of the underlying operating system (OS), it is critical the OS provides *resource isolation* to the container's assigned resources, such as CPU, disk, network bandwidth, and memory. Currently, control groups (or cgroups) in Linux [3] enable resource isolation by allocating, metering, and enforcing resource usage in the kernel.

Resource isolation is an important construct for both application developers and cloud providers. Recent studies indicate today's workloads are heterogeneous and do not easily fit into predetermined bucket allocations [62,69]. Therefore, developers should be able to allocate container resources in a fine-grained manner. For this to be effective, however, a container's provisioned resources must be readily available. When resource availability is compromised due to overprovisioning or ineffective resource isolation, latency-sensitive applications can suffer from performance degradation, which can ultimately impact revenue [7,14,20,49]. In serverless computing, billing is time-based [38] and insufficient resource isolation can cause users to be needlessly overcharged. Cloud providers also rely on resource isolation to employ efficient container orchestration schemes [1,13,69] that enable hyper-dense container deployments per server. However, without hardened bounds on container resource consumption, providers are faced with a trade-off: either underprovision dedicated container resources on each server (and thus waste potential revenue by selling spare compute to lower priority jobs) or allow loose isolation that may hurt customer performance on their cloud.

In this paper, we show containers can utilize more CPU than allocated by their respective cgroup when sending or receiving network traffic, effectively breaking isolation. Modern kernels process traffic via interrupts, and the time spent handling interrupts is often not charged to the container sending or receiving traffic. Without accurately charging containers for network processing, the kernel cannot provide hardened resource isolation. In fact, our measurements indicate the problem can be severe: containers with high traffic rates can cause colocated compute-driven containers to suffer an almost **6**× slowdown. The overhead is high because kernels perform a significant amount of network processing: from servicing interrupts, to protocol handling, to implementing network function virtualizations (e.g., switches, firewalls, rate limiters, etc). Modern datacenter line rates are fast (10-100

Gbps), and studies have shown network processing can incur significant computational overhead [35, 36, 41, 61].

Interference in datacenters is a known problem [49, 53, 64], and researchers have developed schemes to isolate CPU [11, 15, 67] and network bandwidth [42, 56, 58, 65]. In contrast, the recent study of isolating network-based processing has been limited. Prior schemes cannot be applied to modern containerized ecosystems [34] or alter the network subsystem in such a way that interrupt processing becomes less efficient [10, 25].

This paper presents Iron (Isolating Resource Overhead from Networking), a system that monitors, charges, and enforces CPU usage for processing network traffic. Iron implements a careful set of kernel instrumentations to obtain the cost of processing packets at a fine-grained level, while maintaining the efficiency and responsiveness of interrupt handling in the kernel. Iron integrates with the Linux scheduler to charge a container for its traffic. Charging alone cannot provide hardened isolation because processing traffic received by a container after it consumes its CPU allocation can break isolation. As a result, Iron implements a hardware-based scheme to drop packets destined to a container that has exhausted its allocation.

Providing isolation in containerized systems is challenging for many reasons. A container's traffic traverses the entire network stack on the server OS and thus accurate charging requires capturing variations in processing different packet types. A given solution must be computationally light-weight because line rate per-packet operations are prone to high overhead and keeping state across cores can lead to inefficient locking. Finally, limiting interference due to packet receptions is difficult because administrators may not have control over traffic sources. Iron addresses these challenges to effectively enforce isolation for network-based processing. In short, our contributions are as follows:

- A case study showing the computational burden of processing network traffic can be significant. Current cgroup mechanisms do not account for this burden, which can cause an 6× slowdown for some workloads.

- A system called Iron to provide hardened isolation. Iron's charging mechanism integrates with the Linux cgroup scheduler in order to ensure containers are properly charged or credited for network-based processing. Iron also provides a novel packet dropping mechanism to limit the effect, with minimal overhead, of a noisy neighbor that has exhausted its resource allocation.

- An evaluation showing MapReduce jobs can experience over 50% slowdown competing with trace-driven network loads and compute-driven jobs can experience a 6× slowdown in controlled settings. Iron effectively isolates and enforces network-based processing to reduce these slowdowns to less than 5%.

2 Background and Motivation

This section first describes the *interference problem*: that is, how the network traffic of one container can interfere with CPU allocated to another container. Afterwards, we place Iron in the context of past solutions and then empirically examine the impact of interference.

2.1 Network traffic breaks isolation

The interference problem occurs because the Linux scheduler does not properly account for time spent servicing interrupts for network traffic. A brief background on Linux container scheduling, Linux interrupt handling, and kernel packet processing follows.

Linux container scheduling Cgroups limit the CPU allocated to a container by defining how long a container can run (`quota`) over a time period. At a high-level, the scheduler keeps a `runtime` variable that accrues how long the container has run within the current period. When the total runtime of a container reaches its quota, the container is throttled. At the end of a period, the container's runtime is recharged to its quota. The scheduler is discussed in [67].

Linux interrupt handling Linux limits interrupt overhead by servicing interrupts in two parts: a top half (i.e., hardware interrupts) and bottom half (i.e., software interrupts). A hardware interrupt can occur at any time, regardless of which container or process is running. The top half is designed to be light-weight so it only performs the critical actions necessary to service an interrupt. For example, the top half will acknowledge the hardware's interrupt and may directly interface with the device. The top half then schedules the bottom half to execute (i.e., raises a software interrupt). The bottom half is responsible for actions that can be delayed without affecting the performance of the kernel or I/O device. Networking in Linux typically employs *softirqs* (a type of software interrupt) to implement the bottom half. Softirqs are used to transmit deferred transmissions, manage packet data structures, and navigate received packets through the network stack.

Linux's softirq handling directly leads to the interference problem. Software interrupts are checked at the end of hardware interrupt processing or whenever the kernel re-enables softirq processing. Software interrupts run in *process context*. That is, whichever unlucky process is running will have to use its scheduled time to service the softirq. Here, isolation breaks when a container has to use its own CPU time to process another container's traffic.

The kernel tries to minimize softirq handling in process context by limiting the softirq handler to run for a fixed time or budgeted amount of packets. When the budget is exceeded, softirq stops executing and sched-

ules ksoftirqd to run. Ksoftirqd is a kernel thread (it does not run in process context) that services remaining softirqs. There is one ksoftirqd thread per processor. Because ksoftirqd is a kernel thread, the time it spends processing packets is not charged to any container. This breaks isolation by limiting available time to schedule other containers or allowing a container that exhausted its cgroup quota to obtain more processing resources.

Kernel packet processing Consider a "normal" packet transmission in Linux: a packet traverses the kernel from its socket to the NIC. Although this traversal is done in the kernel, it is performed in process context, and hence the time spent sending a packet is charged to the correct container. There are, however, two cases in which isolation can break on the sender. First, when the NIC finishes a packet transmission, it schedules an interrupt to free packet resources. This work is done in softirq context, and hence may be charged to a container that did not send the traffic. The second case arises when there is buffering along the stack, which can commonly occur with TCP (for congestion control and reliability) or with traffic shaping (in qdisc [2]). The packet is taken from the socket to the buffer in process context. Upon buffering the packet, however, the kernel system call exits. Software interrupts are then responsible for dequeuing the packet from its buffer and moving it down the network stack. As before, isolation breaks when softirqs are handled by ksoftirqd or charged to a container that didn't send the traffic.

Receiving packets incurs higher softirq overhead than sending packets. Under reception, packets are moved from the driver's ring buffer all the way to the application socket in softirq context. This traversal may require interfacing with multiple protocol handlers (e.g., IP, TCP), NFVs, NIC offloads (e.g., GRO [16]), or even sending new packets (e.g., TCP ACKs or ICMP messages). In summary, the whole receive chain is performed in softirq context and therefore a substantial amount of time may not be charged to the correct container.

2.2 Putting Iron in context

Previous works can mitigate the interference problem by designing new abstractions to account for container resource consumption or redesigning the OS. Below, Iron's contributions are put in context.

System container abstraction In a seminal paper Banga proposed resource containers [10], an abstraction to capture and charge system resources in use by a particular activity. The work extends Lazy Receiver Processing (LRP) [25]. When a process is scheduled, a receive system call lazily invokes protocol processing in the kernel, and thus time spent processing packets is correctly charged to a process. This approach is inefficient for TCP because at most one window can be consumed between successive system calls [25], and therefore LRP employs a per-socket thread associated with each receiving process to perform asynchronous protocol processing so CPU consumption is charged appropriately.

Although LRP solves the accounting problem, the following issues must be considered. First, as the name implies, LRP only handles receiving traffic and cannot fully capture the overheads of sending traffic. Second, LRP requires a per-socket thread to perform asynchronous TCP processing[1]. Maintaining extra threads leads to additional context switching, which can incur significant overhead for processing large amounts of flows [41]. Third, the scheduler must be made aware of, and potentially prioritize, threads with outstanding protocol processing otherwise TCP can suffer from increased latencies and even drops while it waits for its socket's thread to be scheduled. A similar notion of per-thread softirq processing was proposed in the Linux Real-Time kernel, but ultimately dropped because it increases configuration complexity and reduces performance [30].

Iron explicitly addresses the above concerns. First, Iron correctly accounts for transmissions. Second, Iron seamlessly integrates with Linux's interrupt processing to maintain efficiency and responsiveness. In Linux, all of a core's traffic is processed by that core's softirq handler. Processing interrupts in a shared manner, rather than in a per-thread manner, maintains efficiency by minimizing context switching. Additionally, by servicing hardware interrupts in process context, protocol processing is performed responsively. Linux's design, however, directly leads to the interference problem. Therefore, one contribution of our work is showing accurate accounting for network processing is possible even when interrupt handling is performed in a shared manner.

Redesigning the OS Library OSes [28, 48, 57, 60] redesign the OS by moving network protocol processing from the kernel to application libraries. In these schemes, packet demultiplexing is performed at the lowest level of the network stack: typically the NIC directly copies packets to buffers shared with applications. Since applications process packets from their buffers directly, network-based processing is correctly charged.

Library OSes have numerous practical concerns, however. First, these works face similar challenges as LRP with threaded protocol processing. Second, explicitly removing network processing from the kernel can make management difficult. In multi-tenant datacenters, servers host services such as rate limiting, virtual networking, billing, traffic engineering, health monitoring, and security. With a library OS, admin-defined network processing must be performed in the NIC or in user-level software.

[1] Banga's design uses a per-process asynchronous thread

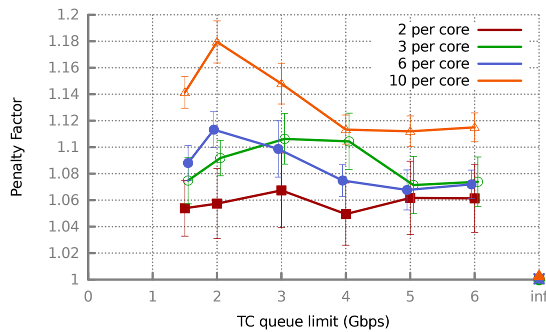


Figure 1: Penalty factor of UDP senders.

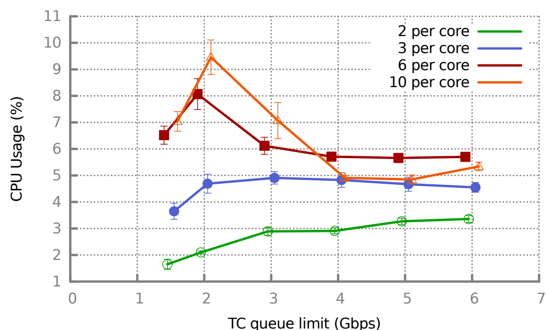


Figure 2: ksoftirqd overhead with UDP senders.

Neither approach is ideal. Application libraries linked by developers make it difficult for admins to insert policies and functionalities at the host. For example, an admin's ability to perform traffic shaping or simply configure a TCP stack may be limited. Furthermore, porting network services to user-level requires every NFV application to track, charge, and enforce network processing to mitigate interference. NIC-based techniques can cost more (to upgrade hosts), scale more poorly in the number of flows and services, and be less flexible and harder to configure than software. And while NICs are becoming more flexible [76], it is likely network management will be dictated by a combination of admin-controlled software and hardware in the future. As such, Iron can help track and enforce software-based network processing. Finally, adapting Library OSes to support multi-tenancy and replacing currently deployed ecosystems can have a high barrier to entry for providers and customers.

2.3 Impact of network traffic

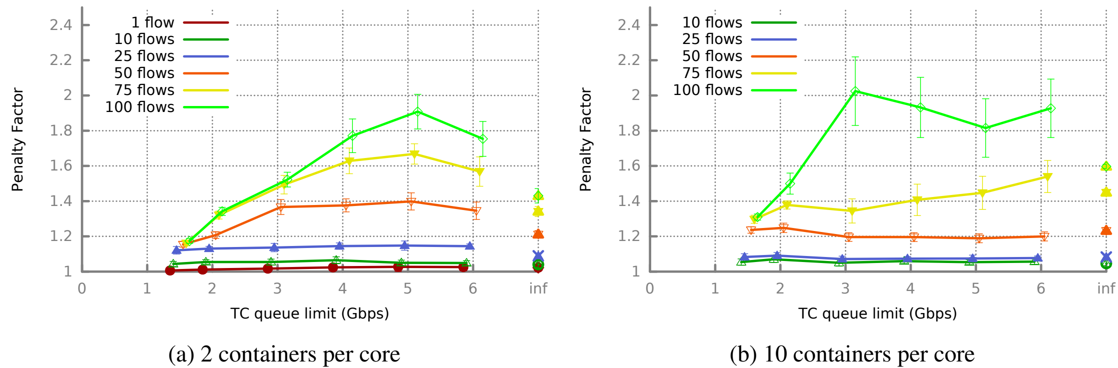
In this section, a set of controlled experiments quantifies the impact of both UDP and TCP network processing on isolation in containerized environments.

Methodology In each experiment, n containers are allocated per core, where n varies from 2, 3, 6, or 10. Each container is configured to obtain an equal share of the core (i.e., $quota = period/n$). This allocation is replicated over all cores. NICs are 25 Gbps, and Section 4 further details methodology. One container per core, denoted the *victim*, runs a CPU-intensive `sysbench` workload [46]. The time to complete each victim's workload is measured under two scenarios. In the first scenario all non-victim containers, henceforth denoted *interferers*, also run `sysbench`. This serves as a baseline case. In the second scenario, the interferers run a simple network flooding application that sends as many back-to-back packets as possible. The victim's completion time is measured under both scenarios, and a *penalty factor* indicates the fraction of time the victim's workload takes when competing with traffic versus competing with `sysbench`. Penalty factors greater than one indicate isolation is broken because traffic is impacting the victim in an adverse way.

For the reception tests, containers are allocated on a single core and all NIC interrupts are serviced on the same core to ensure cores without containers do not process traffic. As before, the victim container runs `sysbench`, but the interferers now run a simple receiver. A multi-threaded sender varies its rate to the core, using 1400 byte packets and dividing flows evenly amongst the receivers. All results are averaged over 10 runs.

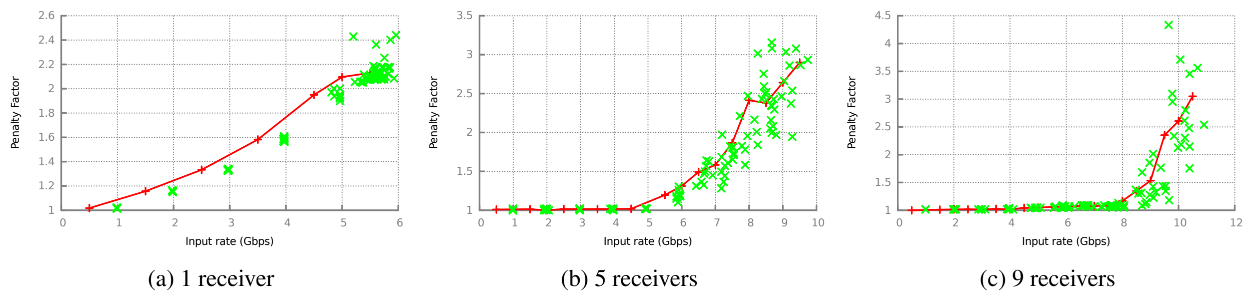
UDP senders These results show the impact when the interfering containers flood 1400 byte UDP traffic. Studies have shown rate limiters can increase computational overhead [61], so the penalty factor is measured when no rate limiters are configured and also when hierarchical token bucket (HTB) [2] is deployed for traffic shaping.

Figure 1 presents the results. Lines denote how many containers are allocated on a core, the x-axis denotes the rate limit imposed on a core, and the y-axis indicates the penalty factor. With n containers per core, each container receives $\frac{1}{n}^{th}$ of the bandwidth allocated to the core. The right-most point labeled "inf" is when no rate limiter is configured. We note the following trends. First, there is no penalty factor with no rate limiting because the application demands are lower than the link bandwidth, so there is no queuing at the NIC. Second, rate limiting causes penalty factors as high as 1.18. The summed application demands can be higher than the imposed rate limit on each core, which means packets are queued in the rate limiter. Softirq handling interferes with the processing time of the victims, leading to high penalty factors. Third, HTB experiences a relatively higher penalty for 1-3 Gbps. When rate limits are 4 Gbps and above, the rate limiter does not shape traffic because senders are CPU-bound and cannot generate more than 4 Gbps of traffic demand. Isolation still breaks because rate limiters maintain state and perform locking (this overhead was also witnessed



(a) 2 containers per core (b) 10 containers per core

Figure 3: Penalty factor of victims with TCP senders.



(a) 1 receiver (b) 5 receivers (c) 9 receivers

Figure 4: Penalty factor when there are 10 containers on 1 core. $i = 1, 5, 9$ of the containers are UDP receivers.

in [43]). For rates below 4 Gbps, senders generate more traffic than the enforced rate and higher overheads occur.

Figure 2 shows the CPU usage of ksoftirqd (on core 0) for the experiment in Figure 1. The trends roughly correspond to the penalty factor overhead. Time spent in ksoftirqd is not attributed to any process, which means that time cannot be issued to other containers. This increases the time it takes for the victim workload to complete. To understand the remaining penalty, we instrumented a run with `perf` [5]. With 10 containers per core and 2 Gbps rate limit, the victim spent 6.99% of its scheduled time servicing softirqs even though it sent no traffic.

TCP senders Figure 3 shows TCP sender performance for 2 and 10 containers per core. Different from the UDP results, the number of flows per core is varied, and flows are divided equally amongst all sending containers on a core. We note the following trends. First, TCP overheads are higher than UDP overheads– in the worst case, the overhead can be as high as 1.95×. TCP overheads are higher because TCP senders receive packets, i.e., ACKs, and also buffer packets at the TCP layer. Both ACK processing and pushing buffered packets to the NIC are completed via softirqs. Therefore, no rate limiting has higher overhead in TCP than UDP. The second interesting trend is overheads increase as the number of flows increase. This occurs for two reasons. First, the number of TCP ACKs increase with flows, and in general, there exists more protocol processing with more flows. Second, a single TCP flow can adapt to the rate limiter, but multiple flows create burstier traffic patterns that increase queuing at the rate limiter.

UDP receivers Figure 4 shows the UDP receiver results. Ten containers are allocated on the core, and if i containers receive UDP traffic, then $10 - i$ containers run `sysbench`. The sender increases its sending rate from 1 Gbps to 12 Gbps at 1 Gbps increments. For each sending rate, 10 trials are run. Each green dot represents the result of a trial. The x-axis indicates the input rate to the core, which can differ from the sending rate due to drops. The red line, provided for reference, averages the penalty factor in 500 Mbps buckets. We varied the number of receivers from 1 to 9, but only show 1, 5, and 9 receivers in the interest of space. We note the following trends. First, the penalty factor for receiving UDP is higher than sending UDP. Packets traverse the whole network stack in softirq context and therefore overheads are larger. Next, as more of the core is allocated to receive (as i increases), the rate at which the server can process traffic increases. As the rate of incoming traffic increases, so does the penalty factor. Under high levels of traffic, the

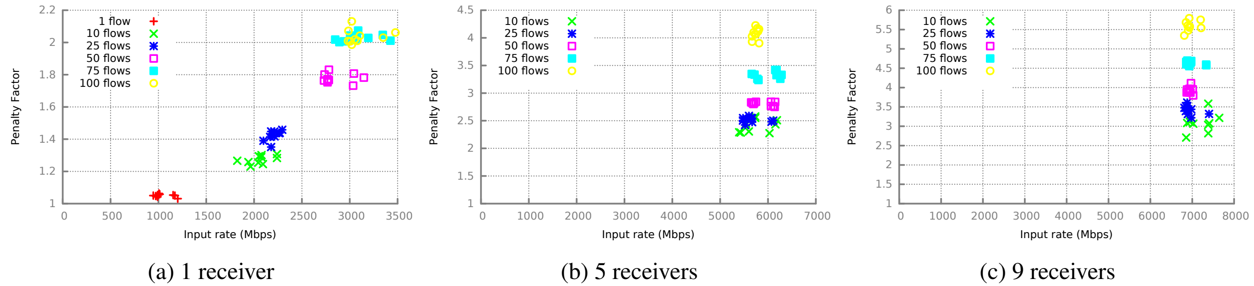

(a) 1 receiver
(b) 5 receivers
(c) 9 receivers

Figure 5: Penalty factor when there are 10 containers on a 1 core. $i = 1,5,9$ of the containers are TCP receivers.

overheads from softirqs cause the victim to take almost 4.5× longer.

TCP receivers Figure 5 shows the results when interfering containers receive TCP traffic. Different from UDP experiments, TCP senders are configured to send as much as they can. TCP will naturally adapt its rate when drops occur (from congestion control) or when receive buffers fill (from flow control). As before, the penalty factor increases as the input rate increases and also when the number of flows increase. In the worst case, interference from TCP traffic causes the victim to take almost six times longer. To further understand this overhead, we instrument `sysbench` with `perf` for nine TCP receivers and 100 flows. Here, ksoftirqd used 54% of the core and `sysbench` spent 60% of its time servicing softirqs. This indicates that isolation techniques must capture softirq overhead in both ksoftirqd and process context.

3 Design

This section details Iron's design. Iron first *accounts* for time spent processing packets in softirq context. After obtaining packet costs, Iron integrates with the Linux scheduler to charge or credit containers for softirq processing. When a container's runtime is exhausted, Iron *enforces* hardened isolation by throttling containers and dropping incoming packets via a hardware-based method.

3.1 Accounting

This section outlines how to obtain per-packet costs in order to ensure accounting is accurate. First, receiver-based accounting is detailed, followed by sender-based accounting. Afterwards, we describe how to assign packets to containers and the state used for accounting.

Receiver-based accounting In Linux, packets traverse the network stack through a series of nested function calls. For example, the IP handler of a packet will directly call the transport handler. Therefore, a function low in the call stack can obtain the time spent processing a packet

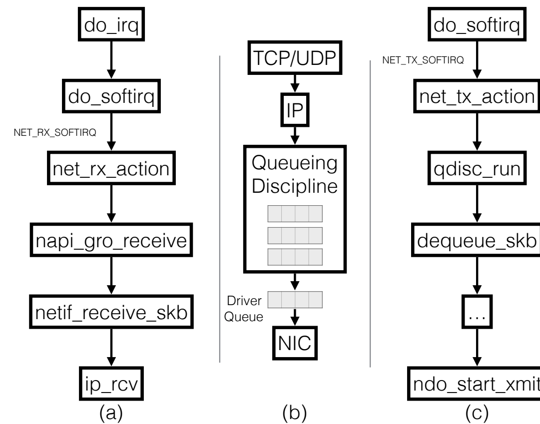


Figure 6: Networking in Linux: (a) subset of receive call stack, (b) send architecture, (c) subset of send call stack.

by subtracting the function start time from the function end time. Figure 6a shows a subset of Linux's receive call stack. Iron instruments `netif_receive_skb` to obtain per-packet costs because it is the first function that handles individual packets outside the driver, regardless of transport protocol[2].

Obtaining the time difference is nontrivial because the kernel is preemptable and functions in the call tree can be interrupted at any time. To ensure only the time spent processing packets is captured, Iron relies on scheduler data. The scheduler keeps the cumulative execution time a thread has been running (*cumtime*), as well as the time a thread was last swapped in (*swaptime*). Coupled with the local clock (*now*), the start and end times can be calculated as: $time = cumtime + (now - swaptime)$.

Besides per-packet costs, there is also a fixed cost associated with processing traffic. That is, there are overheads for entering the function that processes hardware interrupts (`do_IRQ`), processing softirqs, and performing `skb` garbage collection. In Iron, these overheads are lumped together and assigned to packet costs in a weighted fashion. In Linux, six types of softirqs are processed by the

[2]TCP first traverses GRO, but we instrument here for uniformity

softirq handler (`do_softirq`): HI, TX, RX, TIMER, SCSI, and TASKLET. For each interrupt, we obtain the total `do_IRQ` cost, denoted H, and the cost for processing each specific softirq (denoted S_{HI}, S_{TX}, etc). Note software interrupts are processed at the end of a hardware interrupt, so $H > \sum_i S_i$. The overhead associated with processing an interrupt is defined as: $O = H - \sum_i S_i$ and the fair share of the receive overhead within that interrupt is: $O_{RX} = O\frac{S_{RX}}{\sum_i S_i}$. Last, O_{RX} is evenly split amongst packets processed in a given `do_softirq` call to obtain a fixed charge added to each packet.

Finally, we note this scheme is effective in capturing TCP overhead. That is, Iron gracefully handles TCP ACKs and TCP buffering. A TCP flow is handled within a single thread, so when data is received, the thread directly calls a function to send an ACK. When the ACK function returns, the received data continues to be processed as normal. Therefore, ACK overhead is captured by our start and end timestamps. Buffering is also handled correctly. Say packet $i-1$ is lost and thus received packet i is buffered. When retransmitted $i-1$ is received the gap in sequence numbers is filled and TCP will push up the packets to the socket. Correct charging occurs because the cost of moving packet i from the buffer to the socket is captured in the cost of retransmitted packet $i-1$.

Sender-based accounting When sending packets, the kernel has to obtain a lock on a NIC queue. Obtaining a lock on a per-packet basis has high overhead, so packets are often batched for transmission in Linux [18]. Therefore, Iron measures the cost of sending a batch and then charges each packet within the batch for an equal share of the batch cost. The `do_softirq` function calls `net_tx_action` to process transmit softirqs (refer to Figure 6b,c). Then `net_tx_action` calls into the qdisc layer to retrieve packets. Multiple qdisc queues can be dequeued and each queue may return multiple packets. As a result, a linked list of skbs is created and sent to the NIC. Similar to the receiver, `net_tx_action` obtains a start and end time for sending the batch, and O_{TX} is obtained to split the transmission's fixed overheads. Overheads are calculated per core because HTB is work conserving and may dequeue a packet on a different core than it was enqueued.

Container mapping and accounting data structures Iron must identify the container a packet belongs to. On the sender, an skb is associated with its cgroup when enqueued in the qdisc layer. On the receiver, Iron maintains a hash table on IP addresses that is filled when copying packets to a socket.

In Iron, each process maintains a local (per-core) list of packets it processed in softirq context and their individual costs. The per-process structures are eventually merged into a global per-cgroup structure. Iron does this in a way that does not increase locking by merging state when the scheduler obtains a global lock. The per-cgroup structure maintains a variable (`gained`) that indicates if a cgroup should be credited for network processing. Section 3.2 details data structure use.

Algorithm 1 Global runtime refill at period's end

1: **if** gained > 0 **then**
2: runtime $\leftarrow$ runtime + gained
3: gained $\leftarrow 0$
4: **end if**
5: **if** cgroup_idled() and runtime > 0 **then**
6: runtime $\leftarrow 0$
7: **end if**
8: runtime $\leftarrow$ quota + runtime
9: set_timer(now + period)

3.2 Enforcement

This subsection shows how isolation is enforced. Isolation is achieved by integrating accounting data with CPU allocation in Linux's CFS scheduler [67] and dropping packets when a container becomes throttled.

Scheduler integration The CFS scheduler implements CPU allocation for cgroups via a hybrid scheme that keeps both local (i.e., per core) and global state. Containers are allowed to run for a given `quota` within a `period`. The scheduler minimizes locking overhead by updating local state on a fine-grained level and global state on a coarse-grained level. At the global level a `runtime` variable is set to `quota` at the beginning of a period. The scheduler subtracts a `slice` from `runtime` and allocates it to a local core. The `runtime` continues to be decremented until either it reaches zero or the period ends. Regardless, at the end of a period `runtime` is refilled to the quota.

On the local level, a `rt_remain` variable is assigned the `slice` intervals pulled from the global `runtime`. The scheduler decrements `rt_remain` as a task within the cgroup consumes CPU. When `rt_remain` hits zero, the scheduler tries to obtain a slice from the global pool. If successful, `rt_remain` is recharged with a slice and the task can continue to run. If the global pool is exhausted, the local cgroup gets *throttled* and its tasks are no longer scheduled until the period ends.

Iron's global scheduler is presented in Algorithm 1. A global variable `gained` tracks the time a container should get back because it processed another container's softirqs. Line 2 adds `gained` to `runtime`. Next, `runtime` is reset to 0 if the container didn't use its previous allocation because it was limited by its demand (lines 5-7), preserving a CFS policy that disallows unused cycles to be accumulated for use in subsequent periods.

Algorithm 2 Local runtime refill

```
1:  amount ← 0
2:  min_amount ← slice - rt_remain
3:  if cpuusage > 0 then
4:      if cpuusage > gained then
5:          runtime ← runtime - (cpuusage - gained)
6:          gained ← 0
7:      else
8:          gained ← gained - cpuusage
9:      end if
10: else
11:     gained ← gained + abs(cpuusage)
12: end if
13: cpuusage ← 0
14: if runtime = 0 and gained > 0 then
15:     refill ← min(min_amount, gained)
16:     runtime ← refill
17:     gained ← gained - refill
18: end if
19: if runtime > 0 then
20:     amount ← min(runtime, min_amount)
21:     runtime ← runtime - amount
22: end if
23: rt_remain ← rt_remain + amount
```

Last, line 8 refills `runtime`. Note, the runtime input can be negative when a container exceeds its allocated time by sending or receiving too much traffic.

Iron's local algorithm is listed in Algorithm 2. The scheduler invokes this function when `rt_remain` ≤ 0 and after obtaining appropriate locks. The `cpuusage` variable is added to maintain local accounting: positive values indicate the container needs to be charged for unaccounted networking cycles and negative values indicate the container needs a credit for work it did on another container's behalf. Lines 3-9 cover when a container is to be charged, trying to take from `gained` if possible. Lines 10-12 cover the case when a container is to be credited, so `gained` is increased. Lines 14-18 cover a corner case where the runtime may be exhausted, but some credit was accrued and can be used. Lines 19-22 are unchanged: they ensure the container has global runtime left to use. If not, then `amount` remains 0. Line 23 updates the new `rt_remain` by `amount`.

Dropping excess packets While scheduler-based enforcement improves isolation, packets still need to be dropped so a throttled container cannot accrue more network-based processing. Iron does not explicitly drop packets at the sender because throttled containers already cannot generate more outgoing traffic. There exists a corner case when a container has some runtime left and sends a large burst of packets. Currently, the scheduler charges this overage on the next quota refill. We did implement a *proactive* charging scheme that estimates the cost of packet transmission, charges it up-front, and drops packets if necessary. This scheme didn't substantially affect performance, however.

Dropping the receiver's excess packets is more important because a throttled receiver may continue to receive traffic, hence breaking isolation. Iron implements a hardware-based dropping mechanism that integrates with current architectures. Today, NICs insert incoming packets into multiple queues. Each queue has its own interrupt that can be assigned to specified cores. To improve isolation, packets are steered to the core in which their container runs via advanced receive flow steering [39] (FlexNIC [45] also works). Upon reception, the NIC DMAs a packet to a ring buffer in shared memory. Then, the NIC generates an IRQ for the queue, which triggers the interrupt handler in the driver. Modern systems manage network interrupts with NAPI [4]. Upon receiving a new packet, NAPI disables hardware interrupts and notifies the OS to schedule a polling method to retrieve packets. Meanwhile, additionally received packets are simply put in the ring buffer by the NIC. When the kernel polls the NIC, it removes as many packets from the ring buffer as possible, bounded by a budget. NAPI polling exits and interrupt-driven reception is resumed when the number of packets removed is less than the budget.

Our hardware-based dropping mechanism works as follows. First, assume the NIC has one queue per container. Iron augments the NAPI queue structure with a map from a queue to its container (i.e., task_group). When the scheduler throttles a container, it modifies a boolean in task_group. Different from default NAPI, Iron does not poll packets from queues whose containers are throttled. From the kernel's point of view, the queue is stripped from the polling list so that it isn't constantly repolled. From the NIC's point of view, the kernel is not polling packets from the queue, so it stays in polling mode and keeps hardware interrupts disabled. If new packets are received, they are simply inserted into the ring buffer. This technique effectively mitigates receiving overhead because the kernel is not being interrupted or required to do any work on behalf of the throttled container. When the scheduler unthrottles a container, it resets its boolean and schedules a softirq to process packets that may be enqueued.

As a slight optimization, Iron can also drop packets before a container is throttled. That is, if a container is receiving high amounts of traffic and the container is within $T\%$ of its quota, packets can be dropped. This allows the container to use some of its remaining runtime to stop a flood of incoming packets.

Hardware-based dropping is effective when there are a large number of queues per NIC. Even though NICs are

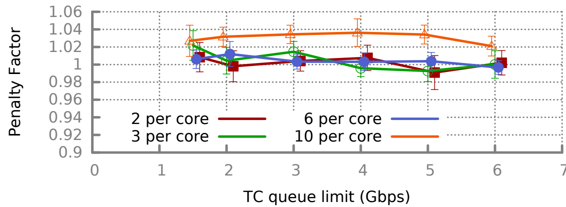


Figure 7: Performance penalty of victim with UDP senders. Compare to Figure 1.

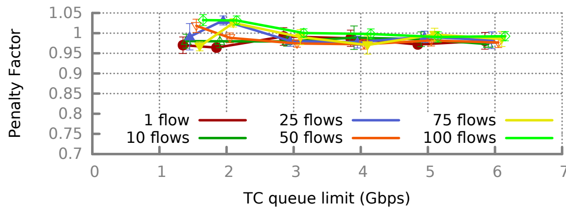


Figure 8: Performance penalty of victim with TCP senders. Compare to Figure 3a.

increasingly outfitted with extra queues (e.g., Solarflare SFN8500-series NICs have 2048 receive queues), in practice the number of queues may not equal the number of containers. Iron can allocate a fixed number of queues per core and then dynamically map problematic containers onto their own queue. Containers without heavy traffic can incur a software-based drop by augmenting the `__netif_receive_skb` function early in the softirq call stack. This dynamic allocation scheme draws inspiration from SENIC [61], which uses a similar approach to scale NIC-based rate limiters. Alternatively, containers can be mapped to queues based on prepurchased bandwidth allocations.

4 Evaluation

This section evaluates the effectiveness of Iron. First, a set of macrobenchmarks show Iron isolates controlled and realistic workloads. Then, a set of microbenchmarks investigates Iron's overhead and design choices.

Methodology The tests are run on Super Micro 5039MS-H8TRF servers with Intel Xeon E3-1271 CPUs. The machines have four cores, with hyper-threading disabled and CPU frequency fixed to 3.2 Ghz. The servers are equipped with Broadcom BCM57304 NetXtreme-C 25 Gbps NICs (driver 1.2.3 and firmware 20.2.25/1.2.2). The servers run Ubuntu 16.04 LTS with Linux kernel 4.4.17. The NICs are set to 25 Gbps for UDP and 10 Gbps for TCP (we noticed instability with TCP at 25 Gbps).

We use `lxc` to create containers and Open Virtual Switch as the virtual switch. Simple UDP and TCP sender and receiver programs create network traffic. The

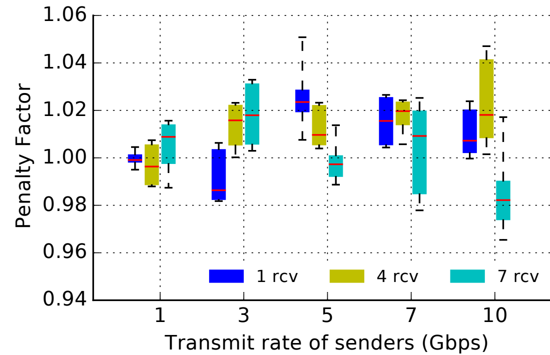


Figure 9: Performance penalty of victim when there are 8 containers on a core. i of the containers are UDP receivers.

`sysbench`'s CPU benchmark is used to measure the computational overhead from competing network traffic. Rate limiters are configured with default burst settings.

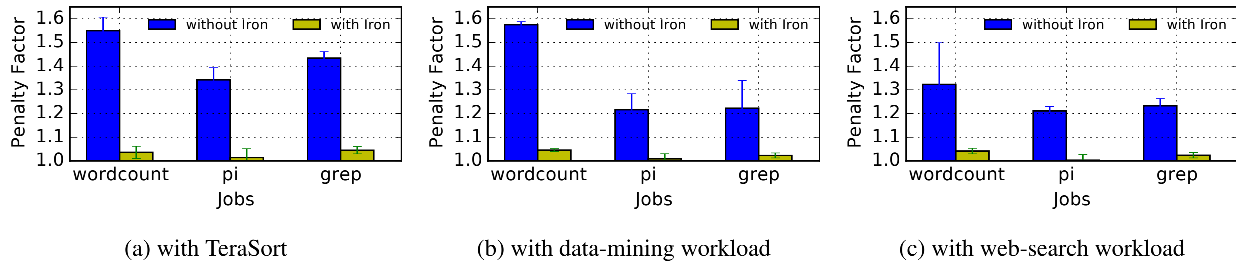
4.1 Macrobenchmarks

Sender-side experiments We run the same experiments in Section 2 to evaluate how well Iron isolates sender interference. Figure 7 shows the impact of UDP senders on `sysbench`. Note this experimental setup is the same as Figure 1. Iron obtains average penalty factors less than 1.01 for 2, 3, and 6 containers, as compared penalty factors as high as 1.11 without Iron. With 10 containers, Iron's penalty factor remains below 1.04, a significant decrease from the maximum of 1.18 without Iron.

Figure 8 shows the performance of Iron with TCP senders, and can be compared to Figure 3a. The maximum penalty factor experienced by Iron is 1.04, whereas the maximum penalty factor without Iron is 1.85. These results show Iron can effectively curtail interference from network-based processing associated with sending traffic.

Receiver-side experiments We rerun the experiments in Section 2 to evaluate how well Iron isolates receiver interference. Even though our NICs support more than eight receive queues, we were unable to modify the driver to expose more queues than cores. Therefore, different from Section 2, a single core is allocated with 8 containers, instead of 10. In these experiments, the number of receiver containers varies from 1, 4, or 7. Containers that are not receivers run an interfering `sysbench` workload. For the UDP experiments, the hardware-based enforcing mechanism was employed, while the TCP experiments utilize our software-based enforcing mechanism.

Figure 9 shows the impact of UDP receivers. The x-axis shows aggregated traffic rate at the sender. This is different from the graphs in Section 2 because Iron drops



(a) with TeraSort

(b) with data-mining workload

(c) with web-search workload

Figure 10: Penalty factor when MapReduce jobs share resources with other workloads.

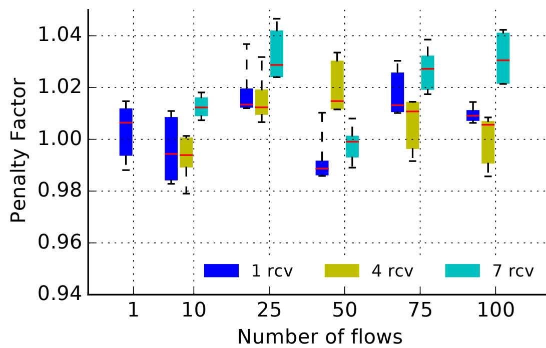


Figure 11: Performance penalty of victim when there are 8 containers on a core. i of the containers are TCP receivers.

packets when container quotas are exceeded, causing received rates to converge. Each number of receivers is indicated by a different bar color. The error bars represent the 5% and 95%. The height of the bars indicate the 25% to 75% and the red horizontal line within each bar is the median. In the previous results without Iron, penalty factors ranged from maximums of 2.45 to 4.45. With Iron, the median penalty factor ranges between 0.98 and 1.02 and never exceeds 1.05. The penalty factor can be lower than 1 when Iron overestimates hard interrupt overheads: overheads, including those occurring after softirq processing, are approximated by using measured values from previous cycles.

Figure 11 shows when interfering containers receive TCP traffic. Unlike UDP, TCP adapts its rate when packet drops occur. Therefore, the software-based rate limiter is effective in reducing interference. In Section 2, the maximum penalty factor ranged from 2.2 to 6. However, with Iron, penalty factors do not exceed 1.05.

Realistic applications Here we evaluate the impact of interference on real applications. We run the experiment on a cluster of 48 containers spread over 6 machines. Each machine has 8 containers (2 per core). The cluster is divided into two equal subclusters such that a container in a subcluster does not share the core with a container from the same subcluster. HTB evenly divides bandwidth between all containers on a machine.

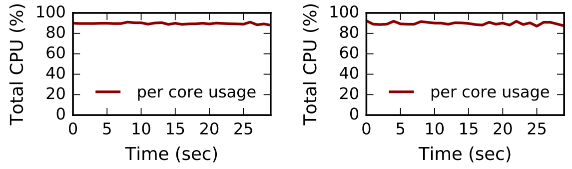
Three MapReduce applications serve as the victims: *pi* computes the value of pi, *wordcount* counts word frequencies in a large file, and *grep* searches for a given word in a file. Three different trace-based interferers run on the other subcluster: the shuffle phase of a *TeraSort* job with a 115GB input file, a *web-search* workload [7] and a *data-mining* workload [6]. For the latter two workloads, applications maintain long-lived TCP connections to every other container in the subcluster, sequentially sending messages to a random destination with sizes distributed from each trace. Figure 10 shows the impact of interference on real applications. Iron obtains an average penalty factor less than 1.04 over all workloads, whereas the average penalty factor ranges from 1.21-1.57 without Iron. These results show Iron can effectively eliminate interference that arises in realistic conditions.

4.2 Microbenchmarks

This subsection evaluates Iron's overhead, the usefulness of runtime packet cost calculation, and the benefits of hardware-based packet dropping.

Performance overhead To measure how accurately Iron limits CPU usage, we allocated 3 containers on a core with each container having 30% of the core. One of the containers ran `sysbench`, while the other two were UDP senders. Figure 12a shows the total CPU used by all containers over a 30 second window. On average, the consumed CPU was around 90.02%. In an ideal case, no more than 90% of the CPU should be utilized. This indicates Iron does not have high overhead in limiting cgroup CPU allocation to its respective limits. We also ran the experiment with a UDP receiver, as shown in Figure 12b. On average Iron ensures an idle CPU of 10.07%, which again shows the effectiveness of our scheme.

Next we analyzed if Iron hurts a network-intensive workload. We instrumented a container to receive traffic and allowed it 100% of the core. Then, at the sender, we generated UDP traffic at 2 Gbps. Using `mpstat`, we measured the CPU consumed by the receiver. The



(a) CPU usage with senders (b) CPU usage with receivers

Figure 12: CPU overhead benchmarks.

Packet type	Average packet cost (usec)
UDP	0.706
TCP	1.670
GRE Tunnel	1.184

Table 1: Average packet processing cost at the receiver.

receiver consumed 35% of the core and received traffic at 1.93 Gbps. Next, we ran the same experiment with the receiver, but this time limited the container to 35% of the core. With Iron limiting the CPU usage, the receiver received traffic at 1.90 Gbps (and used no more than 35% of the CPU). This indicates the overhead of Iron on network traffic is minimal. We ran a similar experiment with the UDP sender and observed no degradation in traffic rate. Unlike the receiver, if a sender is out of CPU cycles it will be throttled, thus generating no extra traffic.

Packet cost variation A simple accounting scheme may charge a fixed packet cost and is likely ineffective because packet processing costs vary significantly at the receiver. Table 1 shows the average packet cost for three classes of traffic. TCP requires bookkeeping for flow control and reliability and results in higher costs than UDP. UDP packets encapsulated in GRE experience extra cost because those packets traverse the stack twice.

Dropping mechanism We compared the impact of software-based versus hardware-based dropping. The UDP sending rate is varied to a receiver with 8 containers on a core (7 are receivers). As shown in Figure 13, both approaches mitigate interference when traffic rates are low. However, when rates are high, the median penalty factor of the software-based rate limiter increases to 1.19, with the 95% approaching a penalty factor of 1.6. The hardware rate-limiter maintains a near-constant penalty factor, regardless of rate.

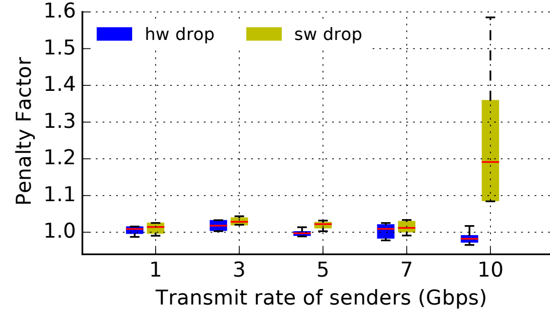


Figure 13: Impact of software and hardware-based packet dropping mechanisms on penalty factor for 7 receivers.

5 Related Work

Here we augment Section 2.2 to further detail prior art.

Isolation of kernel-level resources Many studies have examined how colocated computation suffers from interference [23, 49, 53, 64, 74]. As such, providing resource isolation to underlying server hardware has been a rich area of research. For example, researchers have investigated how to isolate CPU time [11, 15, 67], processor caches [27, 44], memory bandwidth [40, 71], energy [26], and storage [50, 54, 66, 70, 72]. These schemes address problems orthogonal to our work, and none can be generalized to solve the problem Iron solves. While Iron focuses on network-based interrupt processing, Iron's high-level principles of annotating and measuring container-based interrupt overhead can be applied to other interrupts (e.g., timers, storage, etc). For example, tags designed for scheduling I/O storage requests [32, 72] can help account for per-container storage processing overheads. Like Iron, these overheads can be integrated with the Linux CFS scheduler. While developing specific, low-overhead enforcement schemes for other interrupts remains future work, modifying the I/O block scheduler or utilizing software-defined storage mechanisms [66] are promising starts for storage.

A large class of research allocates network bandwidth [9, 33, 42, 56, 58, 59, 63, 65] or isolates congestion control [19, 37] in shared datacenters. In short, these schemes affect network performance but do nothing to control network-based processing time, and thus are complimentary to Iron. Last, some schemes isolate dataplane and application processing on core granularity [12, 41], but do not generalize to support many containers per core nor explicitly study the interference problem.

Resource management and isolation in cloud Determining how to place computation within the cloud has received significant attention. For example, Paragon [21] schedules jobs in an interference-aware fashion to ensure service-level objectives can be achieved. Several other schemes, such as Borg [69], Quasar [22], Heracles [49], and Retro [51] can provision, monitor, and control resource consumption in colocated environments to ensure performance is not degraded due to interference. Iron is largely complementary to these schemes. By provid-

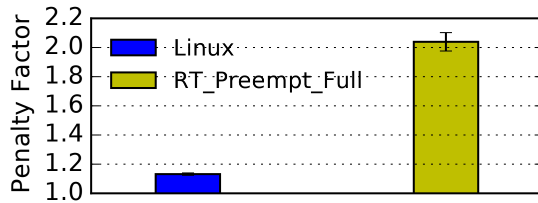


Figure 14: Performance penalty with RT Linux.

ing hardened isolation, Iron allows resource managers to make more informed decisions, so network-heavy jobs cannot impact colocated processor-heavy jobs.

VM network-based accounting Gupta's scheme accounts for processing performed in device drivers for an individual VM [34]. The scheme measures VM-based resource consumption in the hypervisor, integrates with the scheduler to charge for usage, and limits traffic when necessary. Iron differs in many regards. Iron provides performance isolation in container-based environments, instead of VM-based environments. The difference is significant because packets consume more processing time with containers because the network stack lies within the server's kernel, and not the VM's. Gupta relies on a fixed cost to charge for packet reception and transmission, but our results show packet costs vary significantly. Furthermore, because container-based environments incur more processing overhead for traffic, it is important that received traffic is discarded efficiently when necessary. Hence, Iron contains a novel hardware-based enforcement scheme, whereas Gupta's work relies on software.

Shared NFV infrastructure Many works study how to allocate multiple NFVs and their resources on a server [24, 29, 47, 52, 68]. Similar to library OSes, NFV servers require kernel bypass for latency and control. As discussed, kernel bypass approaches cannot easily generalize to solve the interference problem in multi-tenant containerized clouds.

Real-time kernel Real-time (RT) kernels typically aren't used for multi-tenancy, but some RT OSes redesign interrupt processing in a way that could mitigate the interference problem. For example, RT Linux patches the OS so the only type of softirq served in a process's context are those which originated within that process [17]. While this patch doesn't help with receptions, it prevents a container with *no* outgoing traffic from processing interrupts from another container's outgoing traffic. To understand this solution, we ran an experiment with 2 Gbps rate limit and 6 equally-prioritized containers per core: one `sysbench` victim and 5 interferers that flood outgoing UDP traffic. Figure 14 shows the penalty factor for normal Linux and RT Linux (RT_Preempt_Full). The penalty factor of RT Linux is significantly higher than Linux because in RT Linux the victim doesn't process interrupts in its context. Instead, interrupt processing is moved to kernel threads. The processing time used by the kernel threads reduces the time available to the victim. Additionally, RT Linux tries to minimize softirq processing latency and `perf` shows the victim experiences 270× more involuntary context switches as compared to Linux.

Finally, Zhang [75] proposes a RT OS that increases predictability by scheduling interrupts based on process priority and accounting for time spent in interrupts. Zhang's accounting scheme has up to 20% error [75], likely because it is coarse-grained in time and does not use actual, per-packet costs. Overheads in Iron are less than 5% because its accounting mechanism is immediately responsive to actual, per-packet costs. In addition, Iron comprehensively studies the interference problem and introduces enforcement schemes.

Microsoft Windows The scheduler in Windows does not count time spent processing interrupts towards a thread's execution time [73]. This is not sufficient to totally mitigate the interference problem because time spent in interrupt and deferred interrupt processing is not charged to an appropriate thread. Therefore, the large number of cycles consumed by kernel packet processing leave less cycles available to colocated, CPU-heavy threads.

6 Conclusion

This paper presents Iron, a system providing hardened isolation for network-based processing in containerized environments. Network-based processing can have significant overhead, and our case study shows a container running a CPU-intensive task may suffer up to a 6× slowdown when colocated with containers running network-intensive tasks. Iron enforces isolation by accurately measuring the time spent processing network traffic in softirq context within the kernel. Then, Iron relies on an enforcement algorithm that integrates with the Linux scheduler to throttle containers when necessary. Throttling alone is insufficient to provide isolation because a throttled container may receive network traffic. Therefore, Iron contains a hardware-based mechanism to drop packets with minimal overhead. Our scheme seamlessly integrates with modern Linux architectures. Finally, the evaluation shows Iron reduces overheads from network-based processing to less than 5% for realistic and adversarial workloads.

Acknowledgements We thank the reviewers and our shepherd Boon Thau Loo. This work is supported by the National Science Foundation (grants CNS-1302041, CNS-1330308, CNS1345249, and CNS-1717039), and Aditya Akella is also supported by gifts from VMWare, Huawei, and the UW-Madison Vilas Associates.

References

[1] Docker swarm. https://github.com/docker/swarm. Accessed: 2017-09-25.

[2] Linux advanced routing and traffic control howto. http://lartc.org/lartc.html. Accessed: 2017-09-25.

[3] Linux control groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. Accessed: 2017-09-21.

[4] Networking napi. https://wiki.linuxfoundation.org/networking/napi. Accessed: 2017-09-25.

[5] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2017-09-21.

[6] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Matus, F., Pan, R., Yadav, N., Varghese, G., et al. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM* (2014).

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

[8] Amazon Web Servies, Inc. AWS Lambda: Serverless computing. https://aws.amazon.com/lambda/.

[9] Ballani, H., Costa, P., Karagiannis, T., and Rowstron, A. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 242–253.

[10] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *OSDI* (1999), vol. 99, pp. 45–58.

[11] Bartolini, D. B., Sironi, F., Sciuto, D., and Santambrogio, M. D. Automated fine-grained cpu provisioning for virtual machines. *ACM Trans. Archit. Code Optim. 11*, 3 (July 2014), 27:1–27:25.

[12] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (CO, 2014), USENIX Association, pp. 49–65.

[13] Brewer, E. A. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC ’15, ACM, pp. 167–167.

[14] Brutlag, J. Speed Matters for Google Web Search. Tech. rep., 2009. https://services.google.com/fh/files/blogs/google_delayexp.pdf.

[15] Cherkasova, L., Gupta, D., and Vahdat, A. Comparison of the three cpu schedulers in xen. *SIGMETRICS Perform. Eval. Rev. 35*, 2 (Sept. 2007), 42–51.

[16] Corbet, J. Jls2009: Generic receive offload. *Linux Weekly News (LWN)* (Oct. 2009). https://lwn.net/Articles/358910/.

[17] Corbet, J. Software interrupts and realtime. *Linux Weekly News (LWN)* (Oct. 2012). https://lwn.net/Articles/520076/.

[18] Corbet, J. Bulk network packet transmission. *Linux Weekly News (LWN)* (Oct. 2014). https://lwn.net/Articles/615238/.

[19] Cronkite-Ratcliff, B., Bergman, A., Vargaftik, S., Ravi, M., McKeown, N., Abraham, I., and Keslassy, I. Virtualized congestion control. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM ’16, ACM, pp. 230–243.

[20] Dean, J., and Barroso, L. A. The tail at scale. *Communications of the ACM 56* (2013), 74–80.

[21] Delimitrou, C., and Kozyrakis, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS ’13, ACM, pp. 77–88.

[22] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS ’14, ACM, pp. 127–144.

[23] Delimitrou, C., and Kozyrakis, C. Bolt: I know what you did last summer... in the cloud. *SIGARCH Comput. Archit. News 45*, 1 (Apr. 2017), 599–613.

[24] Dobrescu, M., Argyraki, K., and Ratnasamy, S. Toward predictable performance in software packet-processing platforms. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 141–154.

[25] Druschel, P., and Banga, G. Lazy receiver processing (lrp): A network subsystem architecture for server systems. In *OSDI* (1996), vol. 96, pp. 261–275.

[26] Fonseca, R., Dutta, P., Levis, P., and Stoica, I. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI’08, USENIX Association, pp. 323–338.

[27] Funaro, L., Ben-Yehuda, O. A., and Schuster, A. Ginseng: Market-driven llc allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 295–308.

[28] GANGER, G. R., ENGLER, D. R., KAASHOEK, M. F., BRICEÑO, H. M., HUNT, R., AND PINCKNEY, T. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst. 20*, 1 (Feb. 2002), 49–83.

[29] 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* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 1–12.

[30] GLEIXNER, T. [announce] 3.6.1-rt1. *Linux Weekly News (LWN)* (Oct. 2012). https://lwn.net/Articles/518993/.

[31] GOOGLE INC. Cloud functions. https://cloud.google.com/functions/.

[32] GULATI, A., MERCHANT, A., AND VARMAN, P. J. mclock: Handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 437–450.

[33] GUO, C., LU, G., WANG, H. J., YANG, S., KONG, C., SUN, P., WU, W., AND ZHANG, Y. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International Conference* (2010), ACM, p. 15.

[34] GUPTA, D., CHERKASOVA, L., GARDNER, R., AND VAHDAT, A. Enforcing performance isolation across virtual machines in xen. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing* (2006), Springer, pp. 342–362.

[35] HAN, S., MARSHALL, S., CHUN, B.-G., AND RATNASAMY, S. Megapipe: A new programming interface for scalable network i/o. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 135–148.

[36] HE, K., ROZNER, E., AGARWAL, K., FELTER, W., CARTER, J., AND AKELLA, A. Presto: Edge-based load balancing for fast datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 465–478.

[37] HE, K., ROZNER, E., AGARWAL, K., GU, Y. J., FELTER, W., CARTER, J., AND AKELLA, A. Ac/dc tcp: Virtual congestion control enforcement for datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 244–257.

[38] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *Proceedings of HotCloud* (June 2016).

[39] HERBERT, T., AND DE BRUIJN, W. Scaling in the linux networking stack, 2011. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[40] IYER, R., ZHAO, L., GUO, F., ILLIKKAL, R., MAKINENI, S., NEWELL, D., SOLIHIN, Y., HSU, L., AND REINHARDT, S. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS '07, ACM, pp. 25–36.

[41] JEONG, E., WOOD, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mtcp: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 489–502.

[42] JEYAKUMAR, V., ALIZADEH, M., MAZIERES, D., PRABHAKAR, B., KIM, C., AND AZURE, W. Eyeq: Practical network performance isolation for the multi-tenant cloud. In *HotCloud* (2012).

[43] JEYAKUMAR, V., ALIZADEH, M., MAZIÈRES, D., PRABHAKAR, B., KIM, C., AND GREENBERG, A. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI* (2013).

[44] KASTURE, H., AND SANCHEZ, D. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, ACM, pp. 729–742.

[45] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 67–81.

[46] KOPYTOV, A. Sysbench manual. *MySQL AB* (2012).

[47] KULKARNI, S. G., ZHANG, W., HWANG, J., RAJAGOPALAN, S., RAMAKRISHNAN, K. K., WOOD, T., ARUMAITHURAI, M., AND FU, X. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 71–84.

[48] LESLIE, I. M., MCAULEY, D., BLACK, R., ROSCOE, T., BARHAM, P., EVERS, D., FAIRBAIRNS, R., AND HYDEN, E. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J.Sel. A. Commun. 14*, 7 (Sept. 2006), 1280–1297.

[49] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst. 34*, 2 (May 2016), 6:1–6:33.

[50] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)* (Broomfield, CO, October 2014).

[51] MACE, J., BODIK, P., FONSECA, R., AND MUSUVATHI, M. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI* (2015), pp. 589–603.

[52] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 459–473.

[53] MATTHEWS, J. N., HU, W., HAPUARACHCHI, M., DESHANE, T., DIMATOS, D., HAMILTON, G., MCCABE, M., AND OWENS, J. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (New York, NY, USA, 2007), ExpCS '07, ACM.

[54] MCCULLOUGH, J. C., DUNAGAN, J., WOLMAN, A., AND SNOEREN, A. C. Stout: An adaptive interface to scalable cloud storage. In *Proc. of the USENIX Annual Technical Conference–ATC* (2010), pp. 47–60.

[55] MICROSOFT CORP. Azure functions. https://azure.microsoft.com/en-us/services/functions/.

[56] MUNDADA, Y., RAMACHANDRAN, A., AND FEAMSTER, N. Silverline: Data and network isolation for cloud services. In *HotCloud* (2011).

[57] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst. 33*, 4 (Nov. 2015), 11:1–11:30.

[58] POPA, L., KUMAR, G., CHOWDHURY, M., KRISHNAMURTHY, A., RATNASAMY, S., AND STOICA, I. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM* (2012).

[59] POPA, L., YALAGANDULA, P., BANERJEE, S., MOGUL, J. C., TURNER, Y., AND SANTOS, J. R. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 351–362.

[60] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. Rethinking the library os from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (March 2011), ACM.

[61] RADHAKRISHNAN, S., GENG, Y., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. SENIC: Scalable NIC for End-host Rate Limiting. In *NSDI* (2014).

[62] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 7:1–7:13.

[63] RODRIGUES, H., SANTOS, J. R., TURNER, Y., SOARES, P., AND GUEDES, D. O. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *WIOV* (2011).

[64] SHARMA, P., CHAUFOURNIER, L., SHENOY, P., AND TAY, Y. C. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference* (New York, NY, USA, 2016), Middleware '16, ACM, pp. 1:1–1:13.

[65] SHIEH, A., KANDULA, S., GREENBERG, A. G., AND KIM, C. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud* (2010).

[66] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 182–196.

[67] TURNER, P., RAO, B. B., AND RAO, N. Cpu bandwidth control for cfs. In *Proceedings of the Linux Symposium* (2010), pp. 245–254.

[68] VASILESCU, L., OLTEANU, V., AND RAICIU, C. Sharing cpus via endpoint congestion control. In *Proceedings of the Workshop on Kernel-Bypass Networks* (New York, NY, USA, 2017), KBNets '17, ACM, pp. 31–36.

[69] VERMA, A., PEDROSA, L., KORUPOLU, M. R., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[70] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), FAST '07, USENIX Association, pp. 5–5.

[71] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, ACM, pp. 607–618.

[72] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 474–489.

[73] YOSIFOVICH, P., IONESCU, A., RUSSINOVICH, M. E., AND SOLOMON, D. A. *Windows Internals, Part 1: System architecture, processes, threads, memory management, and more*, 7th ed. Microsoft Press, 2017.

[74] ZHANG, W., RAJASEKARAN, S., DUAN, S., WOOD, T., AND ZHUY, M. Minimizing interference and maximizing progress for hadoop virtual machines. *SIGMETRICS Perform. Eval. Rev. 42*, 4 (June 2015), 62–71.

[75] ZHANG, Y., AND WEST, R. Process-aware interrupt scheduling and accounting. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2006), RTSS '06, IEEE Computer Society, pp. 191–201.

[76] ZILBERMAN, N., AUDZEVICH, Y., COVINGTON, G. A., AND MOORE, A. W. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro 34*, 5 (2014), 32–41.

Copa: Practical Delay-Based Congestion Control for the Internet

Venkat Arun and Hari Balakrishnan
M.I.T. Computer Science and Artificial Intelligence Laboratory
Email: {venkatar,hari}@mit.edu

Abstract

This paper introduces Copa, an end-to-end congestion control algorithm that uses three ideas. First, it shows that a *target rate* equal to $1/(\delta d_q)$, where d_q is the (measured) queueing delay, optimizes a natural function of throughput and delay under a Markovian packet arrival model. Second, it adjusts its congestion window in the direction of this target rate, converging quickly to the correct fair rates even in the face of significant flow churn. These two ideas enable a group of Copa flows to maintain high utilization with low queuing delay. However, when the bottleneck is shared with loss-based congestion-controlled flows that fill up buffers, Copa, like other delay-sensitive schemes, achieves low throughput. To combat this problem, Copa uses a third idea: detect the presence of buffer-fillers by observing the delay evolution, and respond with additive-increase/multiplicative decrease on the δ parameter. Experimental results show that Copa outperforms Cubic (similar throughput, much lower delay, fairer with diverse RTTs), BBR and PCC (significantly fairer, lower delay), and co-exists well with Cubic unlike BBR and PCC. Copa is also robust to non-congestive loss and large bottleneck buffers, and outperforms other schemes on long-RTT paths.

1 Introduction

A good end-to-end congestion control protocol for the Internet must achieve high throughput, low queueing delay, and allocate rates to flows in a fair way. Despite three decades of work, these goals have been hard to achieve. One reason is that network technologies and applications have been continually changing. Since the deployment of Cubic [13] and Compound [32, 31] a decade ago to improve on Reno's [16] performance on high bandwidth-delay product (BDP) paths, link rates have increased significantly, wireless (with its time-varying link rates) has become common, and the Internet has become more global with terrestrial paths exhibiting higher round-trip times (RTTs) than before. Faster link rates mean that many flows start and stop quicker, increasing the level of flow churn, but the prevalence of video streaming and large bulk transfers (e.g., file sharing and backups) means that these long flows must co-exist with short ones whose objectives are different (high throughput versus low flow completion time or low interactive delay). Larger BDPs exacerbate the "bufferbloat" problem. A more global Internet leads to flows with very different propagation delays sharing a bottleneck (exacerbating the RTT-unfairness exhibited by many current protocols).

At the same time, application providers and users have become far more sensitive to performance, with notions of "quality of experience" for real-time and streaming media, and various metrics to measure Web performance being developed. Many companies have invested substantial amounts of money to improve network and application performance. Thus, the performance of congestion control algorithms, which are at the core of the transport protocols used to deliver data on the Internet, is important to understand and improve.

Congestion control research has evolved in multiple threads. One thread, starting from Reno, and extending to Cubic and Compound relies on packet loss (or ECN) as the fundamental congestion signal. Because these schemes fill up network buffers, they achieve high throughput at the expense of queueing delay, which makes it difficult for interactive or Web-like applications to achieve good performance when long-running flows also share the bottleneck. To address this problem, schemes like Vegas [4] and FAST [34] use delay, rather than loss, as the congestion signal. Unfortunately, these schemes are prone to overestimate delay due to ACK compression and network jitter, and under-utilize the link as a result. Moreover, when run with concurrent loss-based algorithms, these methods achieve poor throughput because loss-based methods must fill buffers to elicit a congestion signal.

A third thread of research, starting about ten years ago, has focused on important special cases of network environments or workloads, rather than strive for generality. The past few years have seen new congestion control methods for datacenters [1, 2, 3, 29], cellular networks [36, 38], Web applications [9], video streaming [10, 20], vehicular Wi-Fi [8, 21], and more. The performance of special-purpose congestion control methods is often significantly better than prior general-purpose schemes.

A fourth, and most recent, thread of end-to-end congestion control research has argued that the space of congestion control signals and actions is too complicated for human engineering, and that algorithms

can produce better actions than humans. Work in this thread includes Remy [30, 35], PCC [6], and Vivace [7]. These approaches define an objective function to guide the process of coming up with the set of online actions (e.g., on every ACK, or periodically) that will optimize the specified function. Remy performs this optimization offline, producing rules that map observed congestion signals to sender actions. PCC and Vivace perform online optimizations.

In many scenarios these objective-optimization methods outperform the more traditional window-update schemes [6, 35]. Their drawback, however, is that the online rules executed at runtime are much more complex and hard for humans to reason about (for example, a typical Remy controller has over 200 rules). A scheme that uses online optimization requires the ability to measure the factors that go into the objective function, which may take time to obtain; for example, PCC's default objective function incorporates the packet loss rate, but a network running at a low packet loss rate (a desirable situation) will require considerable time to estimate.

We ask whether it is possible to develop a congestion control algorithm that achieves the goals of high throughput, low queueing delay, and fair rate allocations, but which is also *simple* to understand and is *general* in its applicability to a wide range of environments and workloads, and that *performs at least as well* as the best prior schemes designed for particular situations.

Approach: We have developed *Copa*, an end-to-end congestion control method that achieves these goals. Inspired by work on Network Utility Maximization (NUM) [18] and by machine-generated algorithms, we start with an objective function to optimize. The objective function we use combines a flow's average throughput, λ, and packet delay (minus propagation delay), d: $U = \log \lambda - \delta \log d$. The goal is for each sender to maximize its U. Here, δ determines how much to weigh delay compared to throughput; a larger δ signifies that lower packet delays are preferable.

We show that under certain simplified (but reasonable) modeling assumptions of packet arrivals, the steady-state sending rate (in packets per second) that maximizes U is

$$\lambda = \frac{1}{\delta \cdot d_q}, \tag{1}$$

where d_q is the mean per-packet queuing delay (in seconds), and $1/\delta$ is in units of MTU-sized packets. When every sender transmits at this rate, a unique, socially-acceptable Nash equilibrium is attained.

We use this rate as the *target rate* for a Copa sender. The sender estimates the queuing delay using its RTT observations, moves quickly toward hovering near this target rate. This mechanism also induces a property that the queue is regularly almost flushed, which helps all endpoints get a correct estimate of the queuing delay. Finally, to compete well with buffer-filling competing flows, Copa mimics an AIMD window-update rule when it observes that the bottleneck queues rarely empty.

Results: We have conducted several experiments in emulation, over real-world Internet paths and in simulation comparing Copa to several other methods.

1. As flows enter and leave an emulated network, Copa maintains nearly full link utilization with a median Jain's fairness index of 0.86. The median indices for Cubic, BBR and PCC are 0.81, 0.61 and 0.35 respectively (higher the better).
2. In real-world experiments Copa achieved nearly as much throughput and 2-10× lower queueing delays than Cubic and BBR.
3. In datacenter network simulations, on a web search workload trace drawn from datacenter network [11], Copa achieved a $>5\times$ reduction in flow completion time for short flows over DCTCP. It achieved similar performance for long flows.
4. In experiments on an emulated satellite path, Copa achieved nearly full link utilization with an average queuing delay of only 1 ms. Remy's performance was similar, while PCC achieved similar throughput but with $\approx$700 ms of queuing delay. BBR obtained 50% link utilization with $\approx$100ms queuing delay. Both Cubic and Vegas obtained $< 4\%$ utilization.
5. In an experiment to test RTT-fairness, Copa, Cubic, Cubic over CoDel and Newreno obtained Jain fairness indices of 0.76, 0.12, 0.57 and 0.37 respectively (higher the better). Copa is designed to coexist with TCPs (see section §2.2), but when told that no competing TCPs exist, Copa allocated equal bandwidth to all flows (fairness index $\approx$1).
6. Copa co-exists well with TCP Cubic. On a set of randomly chosen emulated networks where Copa and Cubic flows share a bottleneck, Copa flows benefit and Cubic flows aren't hurt (upto statistically insignificant differences) on average throughput. BBR and PCC obtain higher throughput at the cost of competing Cubic flows.

2 Copa Algorithm

Copa incorporates three ideas: first, a target rate to aim for, which is inversely proportional to the measured queueing delay; second, a window update rule that depends moves the sender toward the target rate; and third, a TCP-competitive strategy to compete well with buffer-filling flows.

2.1 Target Rate and Update Rule

Copa uses a congestion window, cwnd, which upper-bounds the number of in-flight packets. On

every ACK, the sender estimates the current rate λ = cwnd/RTTstanding, where RTTstanding is the smallest RTT observed over a recent time-window, τ. We use τ=srtt/2, where srtt is the current value of the standard smoothed RTT estimate. RTTstanding is the RTT corresponding to a "standing" queue, since it's the minimum observed in a recent time window.

The sender calculates the target rate using Eq. (1), estimating the queueing delay as

$$d_q = \text{RTTstanding} - \text{RTTmin}, \tag{2}$$

where RTTmin is the smallest RTT observed over a long period of time. We use the smaller of 10 seconds and the time since the flow started for this period (the 10-second part is to handle route changes that might alter the minimum RTT of the path).

If the current rate exceeds the target, the sender reduces cwnd; otherwise, it increases cwnd. To avoid packet bursts, the sender paces packets at a rate of 2·cwnd/RTTstanding packets per second. Pacing also makes packet arrivals at the bottleneck queue appear Poisson as the number of flows increases, a useful property that increases the accuracy of our model to derive the target rate (§4). The pacing rate is *double* cwnd/RTTstanding to accommodate imperfections in pacing; if it were exactly cwnd/RTTstanding, then the sender may send slower than desired.

The reason for using the smallest RTT in the recent τ=srtt/2 duration, rather than the latest RTT sample, is for robustness in the face of ACK compression [39] and network jitter, which increase the RTT and can confuse the sender into believing that a longer RTT is due to queueing on the forward data path. ACK compression can be caused by queuing on the reverse path and by wireless links.

The Copa sender runs the following steps on each ACK arrival:

1. Update the queuing delay d_q using Eq. (2) and srtt using the standard TCP exponentially weighted moving average estimator.
2. Set $\lambda_t = 1/(\delta \cdot d_q)$ according to Eq. (1).
3. If λ = cwnd/RTTstanding $\leq \lambda_t$, then cwnd = cwnd + $v/(\delta \cdot \text{cwnd})$, where v is a "velocity parameter" (defined in the next step). Otherwise, cwnd = cwnd − $v/(\delta \cdot \text{cwnd})$. Over 1 RTT, the change in cwnd is thus $\approx v/\delta$ packets.
4. The velocity parameter, v, speeds-up convergence. It is initialized to 1. Once per window, the sender compares the current cwnd to the cwnd value at the time that the latest acknowledged packet was sent (i.e., cwnd at the start of the current window). If the current cwnd is larger, then set *direction* to "up"; if it is smaller, then set *direction* to "down". Now, if *direction* is the same as in the previous window, then double v. If not, then reset v to 1. However, start doubling v only after the direction has remained the same for three RTTs. Since direction may remain the same for 2.5 RTTs in steady state as shown in figure 1, doing otherwise can cause v to be >1 even during steady state. In steady state, we want $v=1$.

When a flow starts, Copa performs slow-start where cwnd doubles once per RTT until λ exceeds λ_t. While the velocity parameter also allows an exponential increase, the constants are smaller. Having an explicit slow-start phase allows Copa to have a larger initial cwnd, like many deployed TCP implementations.

2.2 Competing with Buffer-Filling Schemes

We now modify Copa to compete well with buffer-filling algorithms such as Cubic and NewReno while maintaining its good properties. The problem is that Copa seeks to maintain low queuing delays; without modification, it will lose to buffer-filling schemes.

We propose *two distinct modes* of operation for Copa:

1. The *default mode* where $\delta=0.5$, and
2. A *competitive mode* where δ is adjusted dynamically to match the aggressiveness of typical buffer-filling schemes.

Copa switches between these modes depending on whether or not it detects a competing long-running buffer-filling scheme. The detector exploits a key Copa property that the queue is empty at least once every 5·RTT when only Copa flows with similar RTTs share the bottleneck (Section 3). With even one concurrent long-running buffer-filling flow, the queue will not empty at this periodicity. Hence if the sender sees a "nearly empty" queue in the last 5 RTTs, it remains in the default mode; otherwise, it switches to competitive mode. We estimate "nearly empty" as any queuing delay lower than 10% of the rate oscillations in the last four RTTs; i.e., $d_q < 0.1(\text{RTTmax} - \text{RTTmin})$ where RTTmax is measured over the past four RTTs and RTTmin is our long-term minimum as defined before. Using RTTmax allows Copa to calibrate its notion of "nearly empty" to the amount of short-term RTT variance in the current network.

In competitive mode the sender varies $1/\delta$ according to whatever buffer-filling algorithm one wishes to emulate (e.g., NewReno, Cubic, etc.). In our implementation we perform AIMD on $1/\delta$ based on packet success or loss, but this scheme could respond to other congestion signals. In competitive mode, $\delta \leq 0.5$. When Copa switches from competitive mode to default mode, it resets δ to 0.5.

The queue may be nearly empty even in the presence of a competing buffer-filling flow (e.g., because of a recent packet loss). If that happens, Copa will switch

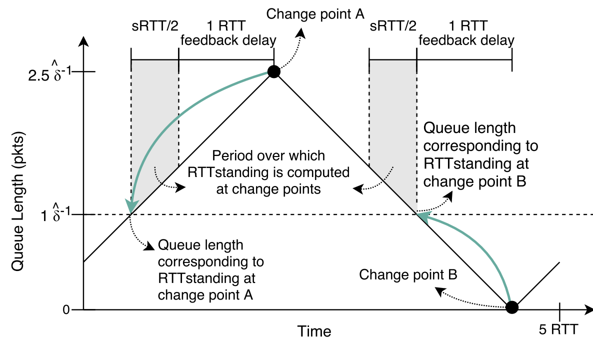


Figure 1: One Copa cycle: Evolution of queue length with time. Copa switches direction at change points A and B when the standing queue length estimated by RTTstanding crosses the threshold of $\hat{\delta}^{-1}$. RTTstanding is the smallest RTT in the last srtt/2 window of ACKs packets (shaded region). Feedback on current actions is delayed by 1 RTT in the network. The slope of the line is $\pm\hat{\delta}$ packets per RTT.

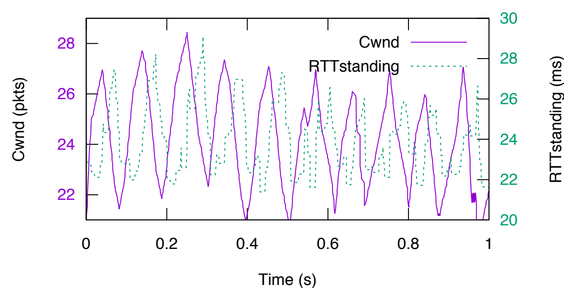


Figure 2: Congestion window and RTT as a function of time for a Copa flow running on a 12 Mbit/s Mahimahi [25] emulated link. As predicted, the period of oscillation is $\approx$5RTT and amplitude is $\approx$5 packets. The emulator's scheduling policies cause irregularities in the RTT measurement, but Copa is immune to such irregularities because the cwnd evolution depends only on comparing RTTstanding to a threshold.

to default mode. Eventually, the buffer will fill again, making Copa switch to competitive mode.

Note that if some Copa flows are operating in competitive mode but no buffer-filling flows are present, perhaps because the decision was erroneous or because the competing flows left the network, Copa flows once again begin to periodically empty the queue. The mode-selection method will detect this condition and switch to default mode.

3 Dynamics of Copa

Figures 1 (schematic view) and 2 (emulated link) show the evolution of Copa's cwnd with time. In steady state, each Copa flow makes small oscillations about the target rate, which also is the equilibrium rate (Section 4). By "equilibrium", we mean the situation when every sender is sending at its target rate. When the propagation delays for flows sharing a bottleneck are similar and comparable to (or larger than) the queuing delay, the small oscillations synchronize to cause the queue length at the bottleneck to oscillate between having 0 and $2.5/\hat{\delta}$ packets every five RTTs. Here, $\hat{\delta}=(\sum_i 1/\delta_i)^{-1}$. The equilibrium queue length is $(0+2.5)\hat{\delta}^{-1}/2=1.25/\hat{\delta}$ packets. When each $\delta = 0.5$ (the default value), $1/\hat{\delta}=2n$, where n is the number of flows.

We prove the above assertions about the steady state using a window analysis for a simplified deterministic (D/D/1) bottleneck queue model. In Section 4 we discuss Markovian (M/M/1 and M/D/1) queues. We assume that the link rate, μ, is constant (or changes slowly compared to the RTT), and that (for simplicity) the feedback delay is constant, RTTmin$\approx$RTT. This means that the queue length inferred from an ACK at time t is $q(t)=w(t-\text{RTTmin})-BDP$, where $w(t)$ is congestion window at time t and BDP is the bandwidth-delay product. Under the constant-delay assumption, the sending rate is cwnd/RTT$=$cwnd/RTTmin.

First consider just one Copa sender. We show that Copa remains in steady state oscillations shown in Figure 1, once it starts those oscillations. In steady state, $v=1$ (v starts to double only after cwnd changes in the same direction for at least 3 RTTs. In steady state, direction changes once every 2.5 RTT. Hence $v=1$ in steady state.). When the flow reaches "change point A", its RTTstanding estimate corresponds to minimum in the $\frac{1}{2}$srtt window of latest ACKs. Latest ACKs correspond to packets sent 1 RTT ago. At equilibrium, when the target rate, $\lambda_t=1/(\delta d_q)$, equals the actual rate, cwnd/RTT, there are $1/\delta$ packets in the queue. When the queue length crosses this threshold of $1/\delta$ packets, the target rate becomes smaller than the current rate. Hence the sender begins to decrease cwnd. By the time the flow reaches "change point B", the queue length has dropped to 0 packets, since cwnd decreases by $1/\delta$ packets per RTT, and it takes 1 RTT for the sender to know that queue length has dropped below target. At "change point B", the rate begins to increase again, continuing the cycle. The resulting mean queue length of the cycle, $1.25/\delta$, is a little higher than $1/\delta$ because RTTstanding takes an extra srtt/2 to reach the threshold at "change point A".

When N senders each with a different δ_i share the bottleneck link, they synchronize with respect to the common delay signal. When they all have the same propagation delay, their target rates cross their actual rates at the same time, irrespective of their δ_i. Hence they increase/decrease their cwnd together, behaving as one sender with $\delta=\hat{\delta}=(\sum_i 1/\delta_i)^{-1}$.

To bootstrap the above steady-state oscillation, target rate should be either above or below the current rate of every sender for at least 1.5 RTT while each $v = 1$. Note, other modes of oscillation are possible, for instance when two senders oscillate about the equilibrium rate 180° out-of-phase, keeping the queue length constant. Nevertheless, small perturbations will cause the target rate to go above/below every sender's current rate, causing the steady-state oscillations described above to commence. So far, we have assumed $v = 1$. In practice, we find that the velocity parameter, v, allows the system to quickly reach the target rate. Then it v remains equal to 1 as the senders hover around the target.

To validate our claims empirically, we simulated a dumbbell topology with a 100 Mbit/s bottleneck link, 20 ms propagation delay, and 5 BDP of buffer in ns-2. We introduce flows one by one until 100 flows share the network. We found that the above properties held throughout the simulation. The velocity parameter v remained equal to 1 most of the time, changing only when flow was far from the equilibrium rate. Indeed, these claims hold in most of our experiments, even when jitter is intentionally added.

We have found that this behavior breaks only under two conditions in practice: (1) when the propagation delay is *much smaller* than the queuing delay and (2) when different senders have very different propagation delays, and the delay synchronization weakens. These violations can cause the endpoints to incorrectly think that a competing buffer-filling flow is present(see §2.2). Even in competitive mode, Copa offers several advantages over TCP, including better RTT fairness, better convergence to a fair rate, and loss-resilience.

Median RTTstanding If a flow achieves an steady-state rate of $\approx \lambda$ pkts/s, the median standing queuing delay, RTTstanding − RTTmin, is $1/(\lambda\delta)$. If the median RTTstanding were lower, Copa would increase its rate more often than it decreases, thus increasing λ. Similar reasoning holds if RTTstanding were higher. This means that Copa achieves quantifiably low delay. For instance, if each flow achieves $1.2 Mbit/s$ rate in the default mode ($\delta = 0.5$), the median equilibrium queuing delay will be 20 ms. If it achieves 12 Mbit/s, the median equilibrium queuing delay will be 2 ms. In this analysis, we neglect the variation in λ during steady state oscillations since it is small.

Alternate approaches to reaching equilibrium. A different approach would be to *directly* set the current sending rate to the target rate of $1/\delta d_q$. We experimented with and analyzed this approach, but found that the system converges only under certain conditions. We proved that the system converges to a constant rate when $C \cdot \sum_i 1/\delta_i <$ (bandwidth delay product), where $C \approx 0.8$ is a dimensionless constant. With ns-2 simulations, we found this condition to be both necessary and sufficient for convergence. Otherwise it oscillates. These oscillations can lead to severe underutilization of the network and it is non-trivial to ensure that we always operate at the condition where convergence is guaranteed.

Moreover, convergence to a constant rate and non-zero queuing delay is not ideal for a delay-based congestion controller. If the queue never empties, flows that arrive later will over-estimate their minimum RTT and hence underestimate their queuing delay. This leads to significant unfairness. Thus, we need a scheme that approaches the equilibrium incrementally and makes small oscillations about the equilibrium to regularly drain the queues.

A natural alternative candidate to Copa's method is additive-increase/multiplicative-decrease (AIMD) when the rate is below or above the target. However, Copa's objective function seeks to keep the queue length small. If a multiplicative decrease is performed at this point, severe under-utilization occurs. Similarly, a multiplicative increase near the equilibrium point will cause a large queue length.

AIAD meets many of our requirements. It converges to the equilibrium and makes small oscillations about it such that the queue is periodically emptied, while maintaining a high link utilization (§5.1). However, if the bandwidth-delay product (BDP) is large, AIAD can take a long time to reach equilibrium. Hence we introduce a velocity parameter §2.1 that moves the rate exponentially fast toward the equilibrium point, after which it uses AIAD.

4 Justification of the Copa Target Rate

This section explains the rationale for the target rate used in Copa. We model packet arrivals at a bottleneck not as deterministic arrivals as in the previous section, but as Poisson arrivals. This is a simplifying assumption, but one that is more realistic than deterministic arrivals when there are multiple flows. The key property of random packet arrivals (such as with a Poisson distribution) is that queues build up even when the bottleneck link is not fully utilized.

In general traffic may be burstier than predicted by Poisson arrivals [28] because flows and packet transmissions can be correlated with each other. In this case, Copa over-estimates network load and responds by implicitly valuing delay more. This behavior is reasonable as increased risk of higher delay is being met by more caution. Ultimately, our validation of the Copa algorithm is through experiments, but the modeling assumption provides a sound basis for setting a good target rate.

4.1 Objective Function and Nash Equilibrium

Consider the objective function for sender (flow) i combining both throughput and delay:

$$U_i = \log \lambda_i - \delta_i \log d_s, \tag{3}$$

where $d_s = d_q + 1/\mu$ is the "switch delay" (total minus propagation delay). The use of switch delay is for technical ease; it is nearly equal to the queuing delay.

Suppose each sender attempts to maximize its own objective function. In this model, the system will be at a Nash equilibrium when no sender can increase its objective function by unilaterally changing its rate. The Nash equilibrium is the n-tuple of sending rates $(\lambda_1,...,\lambda_n)$ satisfying

$$U_i(\lambda_1,...,\lambda_i,...,\lambda_n) > U_i(\lambda_1,...,\lambda_{i-1},x,\lambda_{i+1},...,\lambda_n) \tag{4}$$

for all senders i and any non-negative x.

We assume a first-order approximation of Markovian packet arrivals. The service process of the bottleneck may be random (due to cross traffic, or time-varying link rates), or deterministic (fixed-rate links, no cross traffic). As a reasonable first-order model of the random service process at the bottleneck link, we assume a Markovian service distribution and use that model to develop the Copa update rule. Assuming a deterministic service process gives similar results, offset by a factor of 2. In principle, senders could send their data not at a certain mean rate but in Markovian fashion, which would make our modeling assumption match practice. In practice, we don't, because: (1) there is natural jitter in transmissions from endpoints anyway, (2) deliberate jitter unnecessarily increases delay when there are a small number of senders and, (3) Copa's behavior is not sensitive to the assumption.

We prove the following proposition about the existence of a Nash equilibrium for Markovian packet transmissions. We then use the properties of this equilibrium to derive the Copa target rate of Eq. (1). The reason we are interested in the equilibrium property is that the rate-update rule is intended to optimize each sender's utility independently; we derive it directly from this theoretical rate at the Nash equilibrium. It is important to note that this model is being used not because it is precise, but because it is a simple and tractable approximation of reality. Our goal is to derive a principled target rate that arises as a stable point of the model, and use that to guide the rate update rule.

Lemma 1. *Consider a network with n flows, with flow i sending packets with rate λ_i such that the arrival at the bottleneck queue is Markovian. Then, if flow i has the objective function defined by Eq. (3), and the bottleneck is an M/M/1 queue, a unique Nash equilibrium exists. Further, at this equilibrium, for every sender i,*

$$\lambda_i = \frac{\mu}{\delta_i(\hat{\delta}^{-1}+1)} \tag{5}$$

where $\hat{\delta} = (\sum 1/\delta_i)^{-1}$.

Proof. Denote the total arrival rate in the queue, $\sum_j \lambda_j$, by λ. For an M/M/1 queue, the sum of the average wait time in the queue and the link is $\frac{1}{\mu-\lambda}$. Substituting this expression into Eq. (3) and separating out the λ_i term, we get

$$U_i = \log \lambda_i + \delta_i \log(\mu - \lambda_i - \sum_{j \neq i} \lambda_j). \tag{6}$$

Setting the partial derivative $\frac{\partial U_i}{\partial \lambda_i}$ to 0 for each i yields

$$\delta_i \lambda_i + \sum_j \lambda_j = \mu$$

The second derivative, $-1/\lambda_i^2 - \delta_i/(\mu - \lambda)^2$, is negative.

Hence Eq. (4) is satisfied if, and only if, $\forall i$, $\frac{\partial U_i}{\partial \lambda_i} = 0$. We obtain the following set of n equations, one for each sender i:

$$\lambda_i(1+\delta_i) + \sum_{j \neq i} \lambda_j = \mu.$$

The unique solution to this family of linear equations is

$$\lambda_i = \frac{\mu}{\delta_i(\hat{\delta}^{-1}+1)},$$

which is the desired equilibrium rate of sender i. □

When the service process is assumed to be deterministic, we can model the network as an M/D/1 queue. The expected wait time in the queue is $1/(2(\mu-\lambda)) - \mu/2 \approx 1/2(\mu-\lambda)$. An analysis similar to above gives the equilibrium rate of sender i to be $\lambda_i = 2\mu/(\delta_i(2\hat{\delta}^{-1}+1))$, which is the same as the M/M/1 case when each δ_i is halved. Since there is less uncertainty, senders can achieve higher rates for the same delay.

4.2 The Copa Update Rule Follows from the Equilibrium Rate

At equilibrium, the inter-send time between packets is

$$\tau_i = \frac{1}{\lambda_i} = \frac{\delta_i(\hat{\delta}^{-1}+1)}{\mu}.$$

Each sender does not, however, need to know how many other senders there are, nor what their δ_i preferences are, thanks to the aggregate behavior of Markovian arrivals. The term inside the parentheses in the equation above is a proxy for the "effective" number of other senders, or equivalently the network load, and can be calculated differently.

As noted earlier, the average switch delay for an M/M/1 queue is $d_s = \frac{1}{\mu - \lambda}$. Substituting Eq, (8) for λ in this equation, we find that, at equilibrium,

$$\tau_i = \delta_i \cdot d_s = \delta_i (d_q + 1/\mu), \quad (7)$$

where d_s is the switch delay (as defined earlier) and d_q is the average queuing delay in the network.

This calculation is the basis and inspiration for the target rate. The *does not* model the dynamics of Copa, where sender rates change with time. The purpose of this analysis is to determine a good target rate for senders to aim for. Nevertheless, using steady state formulae for expected queue delay is acceptable since the rates change slowly in steady state.

4.3 Properties of the Equilibrium

We now make some remarks about this equilibrium. First, by adding Eq. (5) over all i, we find that the resulting aggregate rate of all senders is

$$\lambda = \sum \lambda_j = \mu/(1+\hat{\delta}) \quad (8)$$

This also means that the equilibrium queuing delay is $1+1/\hat{\delta}$. If $\delta_i = 0.5$, the number of enqueued packets with n flows is $2n+1$.

Second, it is interesting to interpret Eqs. (5) and (8) in the important special case when the δ_is are all the same δ. Then, $\lambda_i = \mu/(\delta+n)$, which is equivalent to dividing the capacity between n senders and δ (which may be non-integral) "pseudo-senders". δ is the "gap" from fully loading the bottleneck link to allow the average packet delay to not blow up to ∞. The portion of capacity allocated to "pseudo-senders" is unused and determines the average queue length which the senders can adjust by choosing any $\delta \in (0,\infty)$. The aggregate rate in this case is $n \cdot \lambda_i = \frac{n\mu}{\delta+n}$. When δ_is are unequal, bandwidth is allocated in inverse proportion to δ_i. The Copa rate update rules are such that a sender with constant parameter δ is equivalent to k senders with a constant parameter $k\delta$ in steady state.

Third, we recommend a default value of $\delta_i = 0.5$. While we want low delay, we also want high throughput; i.e., we want the largest δ that also achieves high throughput. A value of 1 causes one packet in the queue on average at equilibrium (i.e., when the sender transmits at the target equilibrium rate). While acceptable in theory, jitter causes packets to be imperfectly paced in practice, causing frequently empty queues and wasted transmission slots when a only single flow occupies a bottleneck, a common occurrence in our experience. Hence we choose $\delta = 1/2$, providing headroom for packet pacing. Note that, as per the above equation modeled on an M/M/1 queue, the link would be severely underutilized when there are a small number (≤ 5) of senders. But with very few senders, arrivals at the queue aren't Poisson and stochastic variations don't cause the queue length to rise. Hence link utilization is nearly 100% before queues grow as demonstrated in §5.1.

Fourth, the definition of the equilibrium point is consistent with our update rule in the sense that every sender's transmission rate equals their target rate if (and only if) the system is at the Nash equilibrium. This analysis presents a mechanism to determine the behavior of a cooperating sender: every sender observes a common delay d_s and calculates a common δd_s (if all senders have the same δ) or its $\delta_i d_s$. Those transmitting faster than the reciprocal of this value must reduce their rate and those transmitting slower must increase it. If every sender behaves thus, they will all benefit.

5 Evaluation

To evaluate Copa and compare it with other congestion-control protocols, we use a user-space implementation and ns-2 simulations. We run the user-space implementation over both emulated and real links.

Implementations: We compare the performance of our user-space implementation of Copa with Linux kernel implementations of TCP Cubic, Vegas, Reno, and BBR [5], and user-space implementations of Remy, PCC [6], Vivace [7], Sprout [36], and Verus [38]. We used the developers' implementations for PCC and Sprout. For Remy, we developed a user-space implementation and verified that its results matched the Remy simulator. There are many available RemyCCs and whenever we found a RemyCC that was appropriate for that network, we report its results. We use the Linux qdisc to create emulated links. Our PCC results are for the default loss-based objective function. Pantheon [37], an independent test-bed for congestion control, uses the delay-based objective function for PCC.

ns-2 simulations: We compare Copa with Cubic [13], NewReno [15], and Vegas [4], which are end-to-end protocols, and with Cubic-over-CoDel [26] and DCTCP [1], which use in-network mechanisms.

5.1 Dynamic Behavior over Emulated Links

To understand how Copa behaves as flows arrive and leave, we set up a 100 Mbit/s link with 20 ms RTT and 1 BDP buffer using Linux qdiscs. One flow arrives every second for the first ten seconds, and one leaves every second for the next ten seconds. The mean ± standard deviation of the bandwidths obtained by the flows at each time slot are shown in Figure 3. A CDF of the Jain fairness index in various timeslots is shown in Figure 4.

Both Copa and Cubic track the ideal rate allocation. Figure 4 shows that Copa has the highest median fairness index, with Cubic close behind. BBR and PCC respond much more slowly to changing network

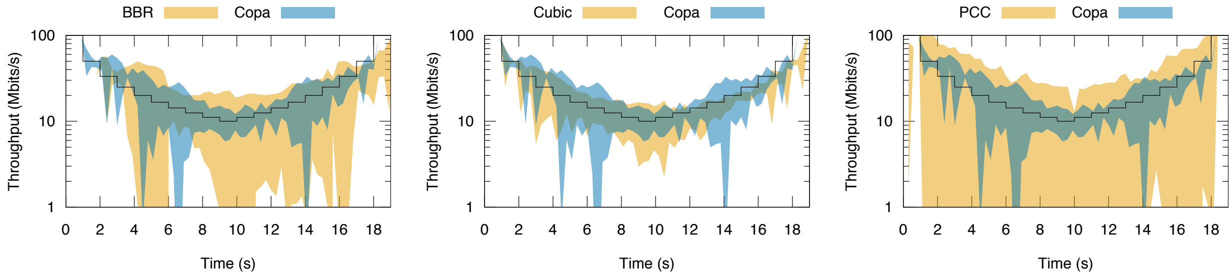


Figure 3: Mean ± std. deviation of throughput of 10 flows as they enter and leave the network once a second. The black line indicates the ideal allocation. Graphs for BBR, Cubic and PCC are shown alongside Copa in each figure for comparison. Copa and Cubic flows follow the ideal allocation closely.

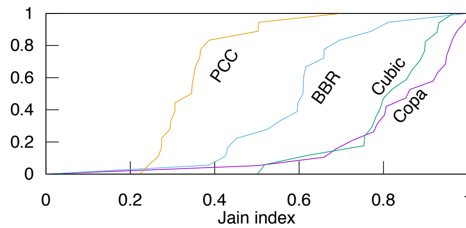


Figure 4: A CDF of the Jain indices obtained at various timeslots for the dynamic behavior experiment (§5.1)

conditions and fail to properly allocate bandwidth. In experiments where the network changed more slowly, BBR and PCC eventually succeeded in converging to the fair allocation, but this took tens of seconds.

This experiment shows Copa's ability to quickly adapt to changing environments. Copa's mode switcher correctly functioned most of the time, detecting that no buffer-filling algorithms were active in this period. Much of the noise and unfairness observed in Copa in this experiment was due erroneous switches to competitive mode for a few RTTs. This happens because when flows arrive or depart, they disturb Copa's steady-state operation. Hence it is possible that for a few RTTs the queue is never empty and Copa flows can switch from default to competitive mode. In this experiment, there were a few RTTs during which several flows switched to competitive mode, and their δ decreased. However, queues empty every five RTTs in this mode as well if no competing buffer-filling flow is present. This property enabled Copa to correctly revert to default mode after a few RTTs.

5.2 Real-World Evaluation

To understand how Copa performs over wide-area Internet paths with real cross traffic and packet schedulers, we submitted our user-space implementation of Copa to Pantheon [37] (`http://pantheon.stanford.edu`), a system developed to evaluate congestion control schemes. During our experiments, Pantheon had nodes in six countries. It creates flows using each congestion control scheme between a node and an AWS server nearest it, and measures the throughput and delay. We separate the set of experiments into two categories, depending on how the node connects to the Internet (Ethernet or cellular).

To obtain an aggregate view of performance across the dozens of experiments, we plot the average normalized throughput and average queuing delay. Throughput is normalized relative to the flow that obtained the highest throughput among all runs in an experiment to obtain a number between 0 and 1. Pantheon reports the one-way delay for every packet in publicly-accessible logs calculated with NTP-synchronized clocks at the two end hosts. To avoid being confounded by the systematic additive delay inherent in NTP, we report the queuing delay, calculated as the difference between the delay and the minimum delay seen for that flow. Each experiment lasts 30 seconds. Half of them have one flow. The other half have three flows starting at 0, 10, and 20 seconds from the start of the experiment.

Copa's performance is consistent across different types of networks. It achieves significantly lower queueing delays than most other schemes, with only a small throughput reduction. Copa's low delay, loss insensitivity, RTT fairness, resistance to buffer-bloat, and fast convergence enable resilience in a wide variety of network settings. Vivace LTE and Vivace latency achieved excessive delays in a link between AWS São Paulo and a node in Columbia, sometimes over 10 seconds. When all runs with > 2000 ms are removed for Vivace latency and LTE, they obtain average queuing delays of 156 ms and 240 ms respectively, still significantly higher than Copa. The Remy method used was trained for a $100\times$ range of link rates. PCC uses its delay-based objective function.

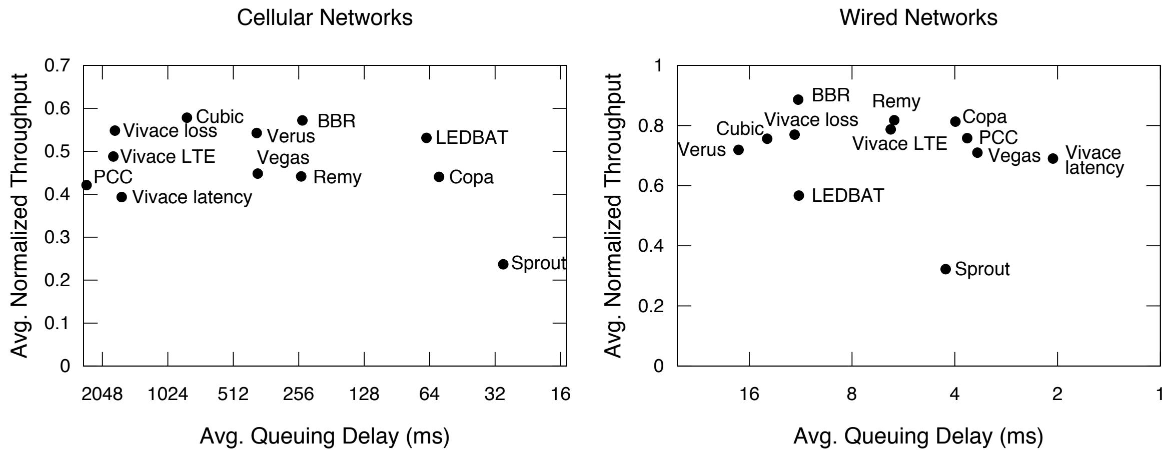


Figure 5: Real-world experiments on Pantheon paths: Average normalized throughput vs. queuing delay achieved by various congestion control algorithms under two different types of Internet connections. Each type is averaged over several runs over 6 Internet paths. Note the very different axis ranges in the two graphs. The x-axis is flipped and in log scale. Copa achieves consistently low queueing delay and high throughput in both types of networks. Note that schemes such as Sprout, Verus, and Vivace LTE designed specifically for cellular networks. Other schemes that do well in one type of network don't do well on the other type. On wired Ethernet paths, Copa's delays are 10× lower than BBR and Cubic, with only a modest mean throughput reduction.

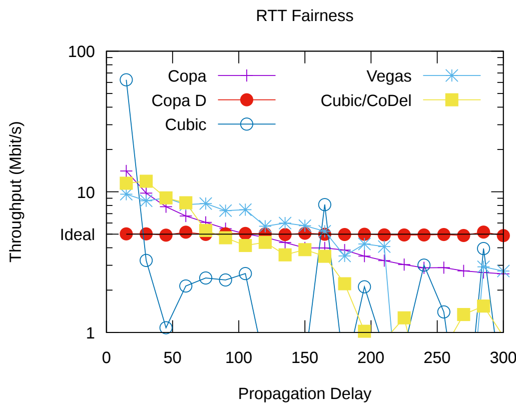


Figure 6: RTT-fairness of various schemes. Throughput of 20 long-running flows sharing a 100 Mbit/s bottleneck link versus their respective propagation delays. "Copa D" is Copa in the default mode without mode switching. "Copa" is the full algorithm. "Ideal" shows the fair allocation, which "Copa D" matches. Notice the log scale on the y-axis. Schemes other than name allocate little bandwidth to flows with large RTTs.

5.3 RTT-fairness

Flows sharing the same bottleneck link often have different propagation delays. Ideally, they should get identical throughput, but many algorithms exhibit significant RTT unfairness, disadvantaging flows with larger RTTs. To evaluate the RTT fairness of various algorithms, we set up 20 long-running flows in ns-2 with propagation delays evenly spaced between 15 ms and 300 ms. The link has a bandwidth of 100 Mbit/s and 1 BDP of buffer (calculated with 300 ms delay). The experiment runs for 100 seconds. We plot the throughput obtained by each of the flows in Figure 6.

Copa's property that the queue is nearly empty once in every five RTTs is violated when such a diversity of propagation delays is present. Hence Copa's mode switching algorithm erroneously shifts to competitive mode, causing Copa with mode switching (labeled "Copa" in the figure) to inherit AIMD's RTT unfriendliness. However, because the AIMD is on $1/\delta$ while the underlying delay-sensitive algorithm robustly grabs or relinquishes bandwidth to make the allocation proportional to $1/\delta$, Copa's RTT-unfriendliness is much milder than in the other schemes.

We also run Copa after turning off the mode-switching and running it in the default mode ($\delta = 0.5$), denoted as "Copa D" in the figure. Because the senders share a common queuing delay and a common target rate, under identical conditions, they will make identical decisions to increase/decrease their rate, but with a time shift. This approach removes any RTT bias, as shown by "Copa D".

In principle, Cubic has a window evolution that is RTT-independent, but in practice it exhibits significant RTT-unfairness because low-RTT Cubic senders are slow to relinquish bandwidth. The presence of the CoDel AQM improves the situation, but significant unfairness remains. Vegas is unfair because several

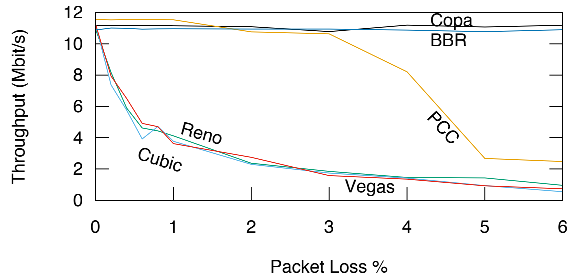


Figure 7: Performance of various schemes in the presence of stochastic packet loss over a 12 Mbit/s link with a 50 ms RTT.

flows have incorrect base RTT estimates as the queue rarely drains. Schemes other than Copa allocate nearly no bandwidth to long RTT flows (note the log scale), a problem that Copa solves.

5.4 Robustness to Packet Loss

To meet the expectations of loss-based congestion control schemes, lower layers of modern networks attempt to hide packet losses by implementing extensive reliability mechanisms. These often lead to excessively high and variable link-layer delays, as in many cellular networks. Loss is also sometimes blamed for the poor performance of congestion control schemes across trans-continental links (we have confirmed this with measurements, e.g., between AWS in Europe and non-AWS nodes in the US). Ideally, a 5% non-congestive packet loss rate should decrease the throughput by 5%, not by 5×. Since TCP requires smaller loss rates for larger window sizes, loss resilience becomes more important as network bandwidth rises.

Copa in default mode does not use loss as a congestion signal and lost packets only impact Copa to the extent that they occupy wasted transmission slots in the congestion window. In the presence of high packet loss, Copa's mode switcher would switch to default mode as any competing traditional TCPs will back off. Hence Copa should be largely insensitive to stochastic loss, while still performing sound congestion control.

To test this hypothesis, we set up an emulated link with a rate of 12 Mbit/s bandwidth and an RTT of 50 ms. We vary the stochastic packet loss rate and plot the throughput obtained by various algorithms. Each flow runs for 60 seconds.

Figure 7 shows the results. Copa and BBR remain insensitive to loss throughout the range, validating our hypothesis. As predicted [22], NewReno, Cubic, and Vegas decline in throughput with increasing loss rate. PCC ignores loss rates up to ≈5%, and so maintains

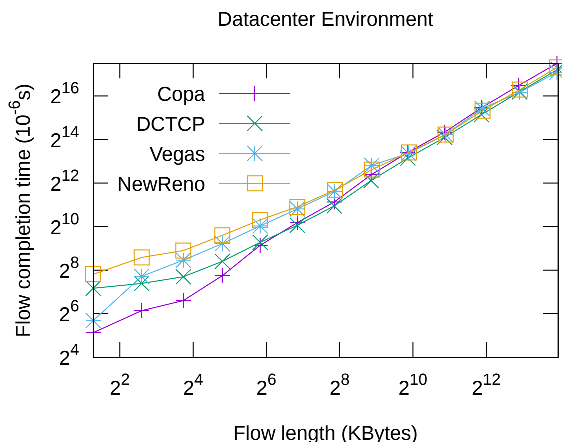


Figure 8: Flow completion times achieved by various schemes in a datacenter environment. Note the log scale.

throughput until then, before falling off sharply as determined by its sigmoid loss function.

5.5 Simulated Datacenter Network

To test how widely beneficial the ideas in Copa might be, we consider datacenter networks, which have radically different properties than wide-area networks. Many congestion-control algorithms for datacenters exploit the fact that one entity owns and controls the entire network, which makes it easier to incorporate in-network support [1, 3, 24, 29, 12]. DCTCP [1] and Timely [23], for example, aim to detect and respond to congestion before it builds up. DCTCP uses routers to mark ECN in packet headers when queue length exceeds a pre-set threshold. Timely demonstrates that modern commodity hardware is precise enough to accurately measure RTT and pace packets. Hence it uses delay as a fine-grained signal for monitoring congestion. Copa is similar in its tendency to move toward a target rate in response to congestion.

Exploiting the datacenter's controlled environment, we make three small changes to the algorithm: (1) the propagation delay is externally provided, (2) since it is not necessary to compete with TCPs, we disable the mode switching and always operate at the default mode with $\delta = 0.5$ and, (3) since network jitter is absent, we use the latest RTT instead of RTTstanding, which also enables faster convergence. For computing v, the congestion window is considered to change in a given direction only if $>2/3$ of ACKs cause motion in that direction.

We simulate 32 senders connected to a 40 Gbit/s bottleneck link via 10 Gbit/s links. The routers have 600 Kbytes of buffer and each flow has a propagation delay of 12 μs. We use an on-off workload with flow lengths drawn from a web-search workload in the

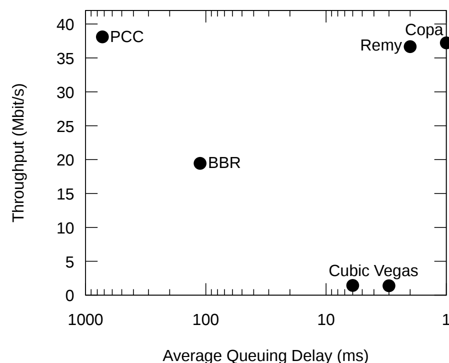


Figure 9: Throughput vs. delay plot for a satellite link. Notice that algorithms that are not very loss sensitive (including PCC, which ignores small loss rates) all do well on throughput, but the delay-sensitive ones get substantially lower delay as well. Note the log-scale.

datacenter [1]. Off times are exponentially distributed with mean 200 ms. We compare Copa to DCTCP, Vegas, and NewReno.

The average flow completion times (FCT) are plotted against the length of the flow in Figure 8, with the *y*-axis shown on a log-scale. Because of its tendency to maintain short queues and robustly converge to equilibrium, Copa offers significant reduction in flow-completion time (FCT) for short flows. The FCT of Copa is between a factor of 3 and 16 better for small flows under 64 kbytes compared to DCTCP. For longer flows, the benefits are modest, and in many cases other schemes perform a little better in the datacenter setting. This result suggests that Copa is a good solution for datacenter network workloads involving short flows.

We also implemented TIMELY [23], but it did not perform well in this setting (over 7 times worse than Copa on average), possibly because TIMELY is targeted at getting high throughput and low delay for long flows. TIMELY requires several parameters to be set; we communicated with the developers and used their recommended parameters, but the difference between our workload and their RDMA experiments could explain the discrepancies; because we are not certain, we do not report those results in the graph.

5.6 Emulated Satellite Links

We evaluate Copa on an emulated satellite link using measurements from the WINDS satellite system [27], replicating an experiment from the PCC paper [6]. The link has a 42 Mbit/s capacity, 800 ms RTT, 1 BDP of buffer and 0.74% stochastic loss rate, on which we run 2 concurrent senders for 100 seconds. This link is challenging because it has a high bandwidth-delay product and some stochastic loss.

Figure 9 shows the throughput v. delay plot for BBR, PCC, Remy, Cubic, Vegas, and Copa. Here we use a RemyCC trained for a RTT range of 30-280 ms for 2 senders with exponential on-off traffic of 1 second, each over a link speed of 33 Mbit/s, which, surprisingly, worked well in this case and was the best performer among the ones available in the Remy repository.

PCC obtained high throughput, but at the cost of high delay as it tends to fill the buffer. BBR ignores loss, but still underutilized the link as its rate oscillated wildly between 0 and over 42 Mbit/s due to the high BDP. These oscillations also causing high delays. Copa is insensitive to loss and can scale to large BDPs due to its exponential rate update. Both Cubic and Vegas are sensitive to loss and hence lose throughput.

5.7 Co-existence with Buffer-Filling Schemes

A major concern is whether current TCP algorithms will simply overwhelm the delay-sensitivity embedded in Copa. We ask: (1) how does Copa affects existing TCP flows?, and (2) do Copa flows get their fair share of bandwidth when competing with TCP (i.e., how well does mode-switching work)?

We experiment on several emulated networks. We randomly sample throughput between 1 and 50 Mbit/s, RTT between 2 and 100 ms, buffer size between 0.5 and 5 BDP, and ran 1-4 Cubic senders and 1-4 senders of the congestion control algorithm being evaluated. The flows are run concurrently for 10 seconds. We report the average of the ratio of the throughput achieved by each flow to its ideal fair share for both the algorithm being tested and Cubic. To set a baseline for variations within Cubic, we also report numbers for Cubic, treating one set of Cubic flows as "different" from another.

Figure 10 shows the results. Even when competing with other Cubic flows, Cubic is unable to fully utilize the network. Copa takes this unused capacity to achieve greater throughput without hurting Cubic flows. In fact, Cubic flows competing with Copa get a higher throughput than when competing with other Cubic flows (by a statistically insignificant margin). Currently Copa in competitive mode performs AIMD on $1/\delta$. Modifying this to more closely match Cubic's behavior will help reduce the standard deviation.

PCC gets a much higher share of throughput because its loss-based objective function ignores losses until about 5% and optimizes throughput. BBR gets higher throughput while significantly hurting competing Cubic flows.

6 Related Work

Delay-based schemes like CARD [17], DUAL [33], and Vegas [4] were viewed as ineffective by much of the community for several years, but underwent a revival in the 2000s with FAST [34] and especially Microsoft's

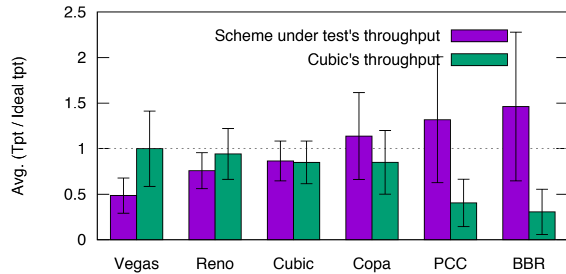


Figure 10: Throughput of different schemes versus Cubic shown by plotting the mean and standard deviation of the ratio of each flow's throughput to the ideal fair rate. The mean is over several runs of randomly sampled networks. The left and right bars show the value for the scheme being tested and Cubic respectively. Copa is much fairer than BBR and PCC to Cubic. It also uses bandwidth that Cubic does not utilize to get higher throughput without hurting Cubic.

Compound TCP [32]. Recently, delay-based control has been used in datacenters by DX [19] and TIMELY [23]. Vegas and FAST share some equilibrium properties with Copa in the sense that they all seek to maintain queue length in proportion to the number of flows. Vegas seeks to maintain between 3 and 6 packets per flow in the queue, and doesn't change its rate if this target is met. Copa's tendency to always change its rate ensures that the queue is periodically empty. This approach has two advantages: (1) every sender gets the correct estimate of minimum RTT, which helps ensure fairness and, (2) Copa can quickly detect the presence of a competing buffer-filling flow and change its aggressiveness accordingly. Further, Copa adapts its rate exponentially allowing it to scale to large-BDP networks. Vegas and FAST increase their rate as a linear function of time.

Network utility maximization (NUM) [18] approaches turn utility maximization problems into rate allocations and vice versa. FAST [34] derives its equilibrium properties from utility maximization to propose an end-to-end congestion control mechanism. Other schemes [24, 14, 24] use the NUM framework to develop with algorithms that use in-network support.

BBR [5] uses bandwidth estimation to operate near the optimal point of full bandwidth utilization and low delay. Although very different from Copa in mechanism, BBR shares some desirable properties with Copa, such as loss insensitivity, better RTT fairness, and resilience to bufferbloat. Experiments §5.2 show that Copa achieves significantly lower delay and slightly less throughput than BBR. There are three reasons for this. First, the default choice of $\delta = 0.5$, intended for interactive applications, encourages Copa to trade a little throughput for a significant reduction in delay. Applications can choose a smaller value of δ to get more throughput, such as $\delta = 0.5/6$, emulating 6 ordinary Copa flows, analogous to how some applications open 6 TCP connections today. Second, BBR tries to be TCP-compatible within its one mechanism. This forced BBR's designers to choose more aggressive parameters, causing longer queues even when competing TCP flows are not present [5]. Copa's use of two different modes with explicit switching allowed us to choose more conservative parameters in the absence of competing flows. Third, both BBR and Copa seek to empty their queues periodically to correctly estimate the propagation delay. BBR uses a separate mechanism with a 10-second cycle, while Copa drains once every 5 RTTs within its AIAD mechanism. As shown in our evaluation in §5.1 and §5.5, Copa is able to adapt more rapidly to changing network conditions. It is also able to handle networks with large-BDP paths better than BBR (§5.6).

7 Conclusion

We described the design and evaluation of Copa, a practical delay-based congestion control algorithm for the Internet. The idea is to increase or decrease the congestion window depending on whether the current rate is lower or higher than a well-defined target rate, $1/(\delta d_q)$, where d_q is the (measured) queueing delay. We showed how this target rate optimizes a natural function of throughput and delay. Copa uses a simple update rule to adjust the congestion window in the direction of the target rate, converging quickly to the correct fair rates even in the face of significant flow churn.

These two ideas enable Copa flows to maintain high utilization with low queuing delay (on average, $1.25/\delta$ packets per flow in the queue). However, when the bottleneck is shared with buffer-filling flows like Cubic or NewReno, Copa, like other delay-sensitive schemes, has low throughput. To combat this problem, a Copa sender detects the presence of buffer-fillers by observing the delay evolution, and then responds with AIMD on the δ parameter to compete well with these schemes.

Acknowledgments

We thank Greg Lauer, Steve Zabele, and Mark Keaton for implementing Copa on BBN's DARPA-funded IRON project and for sharing experimental results, which helped improve Copa. We are grateful to Karthik Gopalakrishnan and Hamsa Balakrishnan for analyzing an initial version of Copa. We thank the anonymous reviewers and our shepherd Dongsu Han for many useful comments. Finally, we would like to thank members of the NMS group at CSAIL, MIT for many interesting and useful discussions. Our work was

funded in part by DARPA's EdgeCT program (award 024890-00001) and NSF grant 1407470.

References

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

[2] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center. In *NSDI*, 2012.

[3] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*, 2013.

[4] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, 1994.

[5] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. Bbr: Congestion-based congestion control. *Queue*, 14(5):50, 2016.

[6] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI*, 2015.

[7] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira. Vivace: Online-learning congestion control. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. USENIX Association, 2018.

[8] J. Eriksson, H. Balakrishnan, and S. Madden. Cabernet: Vehicular Content Delivery using WiFi. In *MobiCom*, 2008.

[9] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *SIGCOMM*, 2013.

[10] M. Ghobadi, Y. Cheng, A. Jain, and M. Mathis. Trickle: Rate limiting youtube video streaming. In *USENIX ATC*, 2012.

[11] 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.

[12] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *NSDI*, pages 1–14, 2015.

[13] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 42(5):64–74, July 2008.

[14] D. Han, R. Grandl, A. Akella, and S. Seshan. Fcp: a flexible transport framework for accommodating diversity. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 135–146. ACM, 2013.

[15] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *SIGCOMM*, 1996.

[16] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[17] R. Jain. A Delay-based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks. In *SIGCOMM*, 1989.

[18] F. P. Kelly, A. Maulloo, and D. Tan. Rate Control in Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, 49:237–252, 1998.

[19] C. Lee, C. Park, K. Jang, S. Moon, and D. Han. Accurate Latency-based Congestion Feedback for Datacenters. In *USENIX ATC*, 2015.

[20] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. Begen, and D. Oran. Probe and adapt: Rate adaptation for HTTP video streaming at scale. *IEEE JSAC*, 32(4):719–733, 2014.

[21] R. Mahajan, J. Padhye, S. Agarwal, and B. Zill. High Performance Vehicular Connectivity with Opportunistic Erasure Coding. In *USENIX ATC*, 2012.

[22] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82, 1997.

[23] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

[24] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 188–201. ACM, 2016.

[25] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX ATC*, 2015.

[26] K. Nichols and V. Jacobson. Controlling Queue Delay. *ACM Queue*, 10(5), May 2012.

[27] H. Obata, K. Tamehiro, and K. Ishida. Experimental evaluation of TCP-STAR for satellite Internet over WINDS. In *International Symposium on Autonomous Decentralized Systems (ISADS)*, 2011.

[28] V. Paxson and S. Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on Networking (ToN)*, 3(3):226–244, 1995.

[29] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *SIGCOMM*, 2014.

[30] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan. An Experimental Study of the Learnability of Congestion Control. In *SIGCOMM*, 2014.

[31] M. Sridharan, K. Tan, D. Bansal, and D. Thaler. Compound TCP: A New TCP congestion control for high-speed and long distance networks. Technical report, Internet-draft draft-sridharan-tcpm-ctcp-02, 2008.

[32] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-speed and Long Distance Networks. In *INFOCOM*, 2006.

[33] Z. Wang and J. Crowcroft. A New Congestion Control Scheme: Slow Start and Search (Tri-S). In *SIGCOMM*, 1991.

[34] D. Wei, C. Jin, S. Low, and S. Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. *IEEE/ACM Trans. on Networking*, 14(6):1246–1259, 2006.

[35] K. Winstein and H. Balakrishnan. TCP ex Machina: Computer-Generated Congestion Control. In *SIGCOMM*, 2013.

[36] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, 2013.

[37] F. Y. Yan, J. Ma, G. Hill, D. Raghavan, R. S. Wahby, P. Levis, and K. Winstein. Pantheon: the training ground for internet congestion-control research. `http://pantheon.stanford.edu/`.

[38] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. Adaptive congestion control for unpredictable cellular networks. In *SIGCOMM*, 2015.

[39] L. Zhang, S. Shenker, and D. D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. 1991.

A Application-Layer Benefits

Many applications benefit from accurate information about path throughput and delay. For example, recently there has been a surge of interest in video streaming, where one of the primary challenges is in estimating the correct bitrate to use. A low estimate hurts video quality while a high estimate risks experiencing a stall in playback. Most algorithms tend to under-estimate rates because stalls hurt the quality of experience more. That, in turn, means they are unable to effectively obtain the true usable path rate.

We showed how every measurement of the queuing delay provides a new estimate of the target rate. Hence, to understand what throughput and delay can be expected from a path, an endpoint only needs to transmit a few packets. The expected performance can be calculated from the measured RTT and queuing delay. These packets can be small, containing only the header and no data, which reduces the bandwidth consumed by probes. Applications can use this information in many ways.

Copa offers a way for applications to obtain rate information. Senders can use the techniques we have developed to measure "expected throughput" – i.e., the rate that a Copa sender will use – by sending only a few small packets, and take an informed decision regarding what quality of content to transfer. As shown in §5.1, Copa's rate estimates are accurate and senders are able to jump directly to the correct rate.

Quick estimation of a transport protocol's expected transmission rate is also useful for selecting good paths or endpoints. For instance, peer-to-peer networks can regularly send tiny packets without payload to monitor the throughput and delay available on the link to a peer. The monitoring is inexpensive, but can enable more informed decisions. Content Distribution Networks using Copa for data delivery can use this too, by routing requests to the appropriate servers. For instance, they can route to minimize the flow-completion time estimated as $2 \cdot \text{RTT} + l/\lambda$, where l is the flow length and λ is the rate estimate.

PCC Vivace: Online-Learning Congestion Control

Mo Dong*, Tong Meng*, Doron Zarchy†, Engin Arslan‡, Yossi Gilad§,
P. Brighten Godfrey* and Michael Schapira†

*UIUC, †Hebrew University of Jerusalem, ‡University of Nevada, Reno, §MIT

Abstract

TCP's congestion control architecture suffers from notoriously bad performance. Consequently, recent years have witnessed a surge of interest in both academia and industry in novel approaches to congestion control. We show, however, that past approaches fall short of attaining ideal performance. We leverage ideas from the rich literature on *online (convex) optimization* in machine learning to design Vivace, a novel rate-control protocol, designed within the recently proposed PCC framework. Our theoretical and experimental analyses establish that Vivace significantly outperforms traditional TCP variants, the previous realization of the PCC framework, and BBR in terms of performance (throughput, latency, loss), convergence speed, alleviating bufferbloat, reactivity to changing network conditions, and friendliness towards legacy TCP in a range of scenarios. Vivace requires only sender-side changes and is thus readily deployable.

1 Introduction

The recent surge of interest in both academia and industry in improving Internet congestion control [8, 11, 13, 19, 21, 22, 24, 25, 31, 32, 36] has made it apparent that today's prevalent congestion control algorithms, the TCP family, fall short of important performance requirements. Indeed, transport rate control faces numerous challenges. First and foremost, a congestion control architecture should be able to efficiently utilize network resources under varying and complex network conditions. This includes optimizing for throughput, loss, and latency, and doing so in a plethora of environments — potentially with non-congestion loss [8], high-RTT cross-continent links, highly dynamic networks such as WiFi and LTE links, etc. Second, congestion control should guarantee quick convergence to stable and fair rates when multiple senders compete over network resources. This desideratum is particularly important for applications like high quality or virtual reality video streaming. Last, a congestion control scheme should be easy and safe (e.g., sufficiently friendly to existing protocols) to deploy.

Traditional algorithms [6, 15, 23] fail to satisfy the first two requirements; their performance can be as high as 10× away from the optimal under non-congestion packet loss [11]. Recent proposals, including Remy [31], PCC [11], and BBR [8], investigate new approaches to this challenge. Remy replaces the human designer with an offline optimization scheme that searches for the best scheme within a certain design space, for a pre-specified range of network conditions. While they can attain high performance, Remy-generated TCPs are inherently prone to degraded performance when the actual network conditions deviate from input assumptions [27].

BBR takes a white-box network-modeling approach, translating change patterns in performance measurements (e.g., increase in delivery rate) to presumed underlying network conditions (e.g., bottleneck throughput and latency). PCC takes a black box approach: a PCC sender observes performance metrics resulting from sending at a specific rate, converts these metrics into a numerical utility value, and adapts the sending rate in the direction that empirically is associated with higher utility. Our experiments indicate that while improving substantially over traditional schemes, both the *specific* realization of PCC in [11], termed "PCC Allegro" (or simply Allegro) henceforth, and BBR's implementation [8], fail to achieve optimal low latency and exhibit far-from-ideal tradeoffs between convergence speed and stability. Specifically, BBR exhibits high rate variance and high packet loss rate upon convergence, whereas PCC Allegro's convergence time is overly long. In addition, when BBR's model of the network does not reflect the complexities of reality, performance can suffer severely. Lastly, they are both highly aggressive towards TCP, although BBR is designed with TCP-friendliness in mind.

To address the above limitations, we draw inspiration from literature on online (convex) optimization [12, 16, 37] to design PCC Vivace, a novel congestion control

scheme. Vivace adopts the high-level architecture of PCC – a utility function framework and a learning rate-control algorithm – but realizes both components differently. First, Vivace relies on a new, learning-theory-informed framework for utility derivation that incorporates crucial considerations such as latency minimization and TCP friendliness. Second, Vivace employs provably (asymptotically) optimal online optimization based on gradient ascent to achieve high utilization of network capacity, swift reaction to changes, and fast and stable convergence. In particular, our contributions are:

(1) A principled framework for transport utility with multiple novel consequences. We prove that for a proper choice of utility functions that incorporate not only throughput and loss (as in Allegro), but also latency, a stable global rate configuration (Nash equilibrium) always exists; show a tradeoff between random loss tolerance and packet loss at convergence with competing senders; and allow controllable capacity allocation among competing senders with heterogeneous utilities (suggesting a future opportunity for centralized network control in an SDN or OpenTCP [13] architecture).

(2) A rate control scheme that utilizes gradient-ascent algorithms from online learning theory to achieve an improved tradeoff between stability and reactivity. We prove that our rate-control scheme guarantees quick convergence to the equilibrium guaranteed by our choice of utility functions, and employ additional techniques to improve rate control in the face of noisy measurements, such as linear regression and low-pass filtering.

(3) Extensive experiments with PCC Vivace, PCC Allegro, BBR, and various TCP variants, in controlled environments, real residential Internet scenarios, and with video-streaming applications. Highlights include: improved performance in rapidly changing conditions (70% less packet loss and 72.5% higher throughput than PCC Allegro, and around 20% median throughput gain over BBR); convergence about 2× faster than Allegro and stability about 2× better than BBR; 57% less video buffering time than BBR with multiple ongoing streams; and significantly improved TCP friendliness.

By no means do we expect that Vivace is the end of the story. Optimizing rate quickly and accurately with limited information in a complex, noisy environment is difficult, and we highlight a simulated LTE environment where a "white-box" model-based approach engineered for this context, namely Sprout [32], outperforms Vivace, as a case for future work. However, Vivace represents a substantial overall advance, showing how a strong learning-theoretic basis yields practical improvements.

2 Rate-Control Through Online Learning

When approached from an online learning perspective, the challenges outlined in § 1 fall naturally into the category of online optimization in machine learning and game theory [16, 37] (a.k.a. "no-regret learning"). Online learning provides a useful and powerful abstraction for decision making under uncertainty. In the online learning setting, a *decision maker* repeatedly selects between available *strategies*. Only after selection is the decision maker aware of the implications of the selected strategy, in terms of a resulting *utility* value. State-of-the-art online learning algorithms provide provable guarantees (namely, the classical "no regret" guarantee [16, 37]) even under *complete* uncertainty about the environment, i.e., without assuming/inferring *anything* about the relation between choices of strategies and the induced utility values. In addition, results in game theory establish that online learning algorithms "play well" together, in the sense that, under the appropriate conditions, global convergence to a stable equilibrium is guaranteed when there are multiple decision makers.

We are thus inspired to apply ideas and machinery from online learning to rate control on the Internet, which can naturally be cast as an online learning task as follows. A traffic sender repeatedly selects between sending rates. After sending at a certain rate for "sufficiently long", the sender learns its performance implications by translating aggregated statistics (e.g., achieved goodput, packet loss rate, average latency) into a numerical utility value, and then adapts the sending rate in response. Importantly, the application of online learning to rate-control is particularly challenging since often only very limited feedback from the network is available to the sender, and so accurately determining the utility derived from sending at a certain rate is sometimes infeasible.

PCC [11] is a promising step towards online-learning-based congestion control. The gist of its architecture is as follows. Time is divided into consecutive intervals, called Monitor Intervals (MIs), each devoted to "testing" the implications for performance of sending at a certain rate. PCC aggregates selective ACKs for packets sent in a MI into the above-mentioned meaningful performance metrics, and feeds these metrics into a utility function that translates them into a numerical value. PCC's rate-control module continuously adjusts the sending rate in the direction that is most beneficial in terms of utility.

However, the specific realization of PCC Allegro in [11] is far from tapping the full potential of online learning. First, Allegro uses a somewhat arbitrary choice of utility function. While [11] proves this choice induces desirable properties in some settings, fair convergence is not provably guaranteed when utility functions are latency-aware, reasoning about fundamental tradeoffs in parameter settings is difficult, and there is no theoretical understanding of what happens when Allegro senders with different utility functions interact with each other.

Second, Allegro inherently ignores the information re-

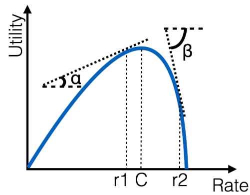


Figure 1: Utility-based rate-control

flected in the utility when deciding on step size. Suppose Allegro's utility function is as described in Figure 1, where C is the capacity of a single link. Then, consider two possible initial rates for the Allegro-sender: r_1 and r_2 ($r_1 < C \ll r_2$). When starting at r_1, the Allegro sender will increase its rate to $r_1(1+\varepsilon)$, for a fixed $\varepsilon > 0$, whereas the initial rate r_2 will be followed by a decrement to $r_2(1-\varepsilon)$. Intuitively, this rate-adjustment is not optimal; a small ε will result in slowly lowering the rate from r_2, leading to long convergence time, whereas choosing too large an ε will increase the rate from r_1 by too much, overshooting the optimum. Indeed, any choice of *fixed* increase/decrease step size is bound to be too much in some circumstances and too little in others, resulting in suboptimal reactivity or convergence.

The combination of Allegro's fairly naive rate control scheme and its ad hoc choice of utility function prevents it from attaining good performance under rapidly changing network conditions, does not alleviate bufferbloat, results in convergence rate/stability tradeoff that is better TCP's yet still suboptimal, leads to high packet loss upon convergence, and is overly aggressive towards TCP. Hence, despite a promising architecture, the operational instantiation of PCC in [11] is still far from optimal.

To address the above limitations, Vivace's design borrows ideas from the rich body of literature on online convex optimization [12, 16, 37] to replace the realization of the two crucial components of PCC's high-level architecture: (1) the utility function framework, and (2) the learning rate-control algorithm. First, Vivace relies on a new, learning-theory-informed framework for utility derivation [12], which guarantees multiple competing Vivace senders will converge to a unique stable rate configuration that is fair and near-optimal. Second, Vivace employs provably optimal gradient-ascent-based no-regret online optimization [37] to adjust sending rates, taking into account not only the direction (increase/decrease) that is more beneficial utility-wise, but also the *extent* to which increasing/decreasing the rate impacts utility.

No-regret learning. The classical objective in online learning theory is *regret minimization*. We give an informal exposition of the implications of no-regret for congestion control here. See [16, 37] for a more complete treatment. A "no-regret" rate-control protocol, such as Vivace, guarantees that its choices of rates are asymptotically (across time) no worse, utility-wise, than sending at what would have been (in hindsight) the *best fixed* rate. No-regret is a useful guarantee for two reasons.

First, no-regret provides a formal performance guarantee for individual senders across *all* network conditions, even highly dynamic or *adversarially* chosen (within the scope of the model). We believe Vivace is the first congestion control scheme to provide such a guarantee.

Second, no-regret provides a powerful lens for theoretical analysis, which we will use to reason formally about convergence with multiple competing no-regret senders, even with heterogeneous utility functions across the senders, and also about tradeoffs between resilience to non-congestion loss and loss upon convergence.

Limitations of no-regret. As no-regret relates performance to the best *fixed* strategy, the quality of this guarantee in a dynamic environment depends on the speed at which the protocol minimizes "regret" [16]. If, from an arbitrary starting state, the regret vanishes to a desired low value within T time units, then the no-regret guarantee applies relative to the best fixed strategy within *every* T units of time. Empirically (§5.1.4), Vivace adapts quickly to changes in network conditions.

Of course, a guarantee of near-optimality relative to the best *dynamic* strategy would be even better. However, such guarantees often entail assumptions about the environment, e.g., that the network behavior exhibits high regularity. Vivace reflects the design choice of avoiding such assumptions. That said, an important direction for future research is to quantify to what extent real-world networks are sufficiently predictable (e.g., via machine learning) to improve rate selection.

PCC Allegro vs. PCC Vivace. Compared with PCC Allegro, PCC Vivace's utility framework (1) incorporates latency awareness, mitigating the bufferbloat problem and the resulting packet loss and latency inflation, (2) extends to heterogeneous senders with different utility functions, enabling flexible network-resource allocation, and (3) induces more friendly behavior towards TCP, and thus is better suited for real-world deployment. In addition, Vivace's rate-control algorithm (1) provides faster, more stable convergence, and (2) reacts more quickly upon changes to network conditions.

3 Vivace's Utility Framework

Vivace divides time into consecutive Monitor Intervals (MIs). At the end of each MI, sender i applies the following utility function to transform the performance statistics gathered at that MI to a numerical utility value:

$$u\left(x_i, \frac{d(RTT_i)}{dT}, L_i\right) = x_i^t - b x_i \frac{d(RTT_i)}{dT} - c x_i \times L_i, \quad (1)$$

where $0 < t < 1, b \geq 0, c > 0$, x_i is sender i's sending rate, and L_i is its observed loss rate. The term $\frac{d(RTT_i)}{dT}$ is the observed "RTT gradient" during this MI, *i.e.*, the increase in latency experienced within this MI. The parameters b, c, t are constants. Intuitively, utility functions of the above

form reward increase in throughput (via x_i^t), and penalize increase in both latency ($bx_i \frac{d(RTT_i)}{dT}$) and loss ($cx_i \times L_i$). We next identify the properties of utility functions within our framework, and refer to [2] for formal analysis.

To see why Vivace's utility function does not consider the absolute value of latency, instead using RTT gradient, consider the following example. A single sender on a link with a large buffer sends at a rate of twice the capacity of the link for a single MI; then, in the next MI, it tries a slightly lower but still over-capacity rate. Such a sender would experience higher absolute latency in the second MI than in the first MI (since the link's queue is only further lengthened), even though lowering the rate was clearly the right choice. To learn within a single MI that lowering the rate is more beneficial, the sender examines the rate at which latency increases or decreases.

The choice of values for the parameters b, c and t in the utility function have crucial implications for the existence of an equilibrium point when multiple Vivace senders compete, and for the latency and congestion loss in such an equilibrium. Due to space limitations, the full proofs of the theorems in this section appear in [2].

3.1 Stability and Fairness

When $t \leq 1$, the family of utility functions in Equation 1 falls into the category of "socially-concave" in game theory [12]. A utility function within this category, when coupled with a theoretical model of Vivace's online-learning rate-control scheme (described in § 4), guarantees high performance from the individual sender's perspective and ensures quick convergence to a global rate-configuration [16, 37]. Specifically, we consider a network model with n senders competing on a bottleneck link with a FIFO queue. The following theorem shows convergence to a fair equilibrium.

Theorem 1. *When n Vivace-senders share a bottleneck link, and each Vivace-sender i's utility function is defined as in Eq. 1, the senders' sending rates converge to a fixed configuration* $(x_1^*, \ldots, x_n^*)$ *such that* $x_1^* = x_2^* = \ldots = x_n^*$*.*

We further analyze the latency in equilibrium. Ideally, upon convergence the latency will not exceed the base RTT, i.e., the RTT when the link buffer is not occupied. Theorem 2 shows how Vivace can achieve that through a proper assignment of value for the parameter b.

Theorem 2. *Let C denote the capacity of the bottleneck link. If* $b \geq tn^{2-t}C^{t-1}$*, then the latency in the unique stable configuration is the base RTT.*

3.2 Random Loss *vs.* Congestion

Non-congestion packet loss (due to lossy wireless links, port flaps on routers, *etc.*) is a common phenomenon in today's Internet [8]. We say a rate-control protocol is *p-loss-resilient* if that protocol does not decrease its sending rate under random loss rate of at most p.

For Vivace to be p-loss-resilient, we need to set c in the above utility function framework to be $c = \frac{tC^{t-1}}{p}$ (details in [2]). However, **enduring more random loss comes at a price**. To simplify the analysis, assume that $b = 0$ (i.e., the utility function is purely loss-based). The following theorem captures a link between random loss resilience and loss due to many competing senders.

Theorem 3. *In a system of n Vivace senders, each p-loss-resilient, the loss rate L of each sender i in equilibrium (with no random loss) satisfies*

$$p = \frac{nL - L + 1}{(1-L)^{1-t} n^{2-t}}. \tag{2}$$

To illustrate Theorem 3 with simplified algebra,[1] suppose that $t = 1$ and $b = 0$. Enduring random loss rate of p implies that the derivative of the utility function u satisfies:

$$\dot{u} = 1 - cp \geq 0, \quad \text{and so:} \quad c \leq 1/p$$

By plugging $t = 1$ in Equation 2, we find that in a system of n Vivace senders sharing a link, the loss rate experienced by each sender under equilibrium (to which Vivace is guaranteed to converge, by Theorem 1), given $n > c$ (if $n < c$, then $L = 0$), is $L = p\frac{n-1/p}{n-1}$.

When $n \to \infty$, the congestion loss rate on convergence approaches the random loss resilience p! Therefore, **withstanding more random loss comes at the cost of suffering more loss upon convergence for a large number of senders**. Our experiments with TCP, BBR, and Allegro, show a similar tradeoff, indicating that this is a barrier for current congestion control frameworks.

3.3 Heterogeneous Senders

So far, our discussion focused on the environment where the senders are homogeneous, i.e., they employ the same utility function. However, our utility framework allows us to reason about interactions between heterogeneous Vivace senders competing over a shared link.

Recent studies on SDN-based traffic engineering [17, 18] and network optimization for big-data systems [9] suggest a need for resource allocation at the transport layer. However, globally allocating network resources to transport-layer connections usually involves complex schemes for rate-limiting at end-hosts, or utilizing in-network isolation mechanisms. OpenTCP [13] proposes allocating bandwidth by tuning TCP parameters or switching between TCP variants. Yet, as TCP has no direct control knobs for global network-resource allocation, OpenTCP resorts to clever "hacks" and complicated feedback loops to indirectly achieve such control.

[1] Our theorems require $t < 1$, but t can be arbitrarily close to 1.

Vivace's utility function framework, in contrast, provides flexibility in resource-allocation. As a concrete example, consider the following loss-based utility:

$$u(x_i, L_i) = x_i - c_i x_i \left(\frac{1}{1 - L_i} - 1 \right) \quad (3)$$

This utility function, similar to that of § 3, induces a unique stable rate configuration to which Vivace senders are guaranteed to converge (more formally, it is also "socially concave"). Now, suppose that n Vivace senders share a link and the goal is to allocate the link's bandwidth C between the senders by assigning a rate of x_i to each sender such that $\sum_{j \in N} x_j = C$. Then we have:

Theorem 4. *A system of n Vivace senders, in which each sender's utility function is of the form in Equation 3, converge to rate configuration $x_1^*, x_2^*, ..., x_n^*$, if for each $i \in [n]$, the loss penalty coefficient in Equation 3 is set to $c_i = \frac{C}{x_i^*}$.*

Hence, one can flexibly adjust the bandwidth allocated to each sender at equilibrium by tuning Vivace's parameters $\{c_i\}$. We experimentally validate this result in § 5.

4 Vivace's Rate Control

Vivace's rate control begins with a **slow start** phase in which the sender doubles sending rate every MI and permanently exits slow start when its empirically-derived utility value decreases for the first time. It then enters the **online learning** phase, which we focus on here.

4.1 Key Idea and Challenges

Vivace's online learning phase employs an online gradient-ascent learning scheme to select transmission rates. This choice of rate-control algorithm is very appealing from an online optimization theory perspective [12, 16, 37]. Specifically, when the utility functions are strictly convex, which is satisfied when $t < 1$ in our utility function formulation (Equation 1), the following two desiderata are fulfilled (see [2] for proofs). (1) Each sender is guaranteed that employing Vivace is (asymptotically) no worse than the optimal *fixed* sending rate in hindsight, termed the "no-regret" guarantee in online learning literature [12, 16, 37]. This is a strong guarantee in that it applies even when the sender is presented with adversarial environments, but it is limited in the sense that it quantifies performance with respect to the actual history of experienced network conditions and not to the conditions that would have resulted from sending at other rates. (2) When multiple senders share the same link, quick convergence to an equilibrium point is guaranteed.

In theory, applying gradient ascent to rate-control means starting at some initial transmission rate and repeatedly estimating γ, by which we denote the gradient (with respect to sending rate) of the utility function, through sampling and changing the rate by $\theta\gamma$, where θ is initially set to be a very high positive number. With time, θ gradually diminishes to 0. But realizing this in practice involves nontrivial operational challenges.

The first challenge is deciding on the extent to which the rate should be increased or decreased. The above theoretical rate-adjustment rule suffers from two serious problems: (a) The initial step size is potentially huge, resulting in a sender jumping between very low (e.g. 1 Mbps) and very high rates (e.g. 500 Mbps) in tens of milliseconds, causing high loss rates and latency inflation. (b) As time goes by, and θ diminishes, changes in rate become small, leading to slow reaction to changed network conditions like newly-free capacity. Second, there are challenges in the basic task of estimating γ. What happens when the environment is noisy, e.g., due to complex interactions between multiple senders, non-congestion loss, or microbursts unrelated to long-term congestion? We next explain how Vivace tackles this.

4.2 Translating Utility Gradients to Rates

Vivace's online learning algorithm begins by computing the gradient of the utility function. Suppose the current sending rate is r. Then, in the next two MIs, the sender will test the rates $r(1+\varepsilon)$ and $r(1-\varepsilon)$, compute the corresponding numerical utility values, u_1 and u_2, respectively, and estimate the gradient of the utility function to be $\gamma = \frac{u_1 - u_2}{2\varepsilon r}$. Then, Vivace utilizes γ to deduce the direction and extent to which rate should be changed, selects the newly computed rate, and repeats the above process.

To convert γ into a change in rate, Vivace starts with fairly low "conversion factor" and increases the conversion factor value as it gains confidence in its decisions. Specifically, initially θ is set to be a conservatively small number θ_0 and so, at first, the rate change is $\Delta_r = \theta_0\gamma$ (i.e., $r_{\text{new}} = r + \theta_0\gamma$). We introduce the concept of *confidence amplifier*. Intuitively, when the sender repeatedly decides to change the rate in the same direction (increase vs. decrease), the confidence amplifier is increased. The confidence amplifier is a monotonically nondecreasing function that assigns a real value $m(\tau)$ to any integer $\tau \geq 0$. After a sender makes τ consecutive decisions to change the rate in the same direction, θ is set to $m(\tau)\theta_0$ (and so the rate is changed by $\Delta_r = m(\tau)\theta_0\gamma$). Setting $m(0) = 1$ implies that initially the change in rate is $\theta_0\gamma$, as described above. When the direction at which rate is adapted is reversed (increase to decrease or vice-versa), τ is set back to 0 (and the above process starts anew).

Sampled utility-gradient can be excessively high due to unreliable measurements or large changes to network conditions between MIs. For instance, a burst of losses when probing $r(1-\varepsilon)$ and no losses when probing $r(1+\varepsilon)$ might result in huge γ and, consequently, a drastic rate change that overshoots the link's capacity. To address this, we introduce a mechanism, called

the *dynamic change boundary* ω. Whenever Vivace's computed rate change (Δ_r) exceeds ωr, the effective rate change is capped at ωr. The dynamic change boundary is initialized to some predetermined value $\omega = \omega_0$, is gradually increased every time $\Delta_r > \omega$, and is decreased when $\Delta_r < \omega$. Specifically, ω is updated to $\omega = \omega_0 + k \cdot \delta$ following k consecutive rate adjustments in which the gradient-based rate-change Δ_r exceeded the dynamic change boundary, for a predetermined constant $\delta > 0$. Whenever $\Delta_r \leq r \cdot \omega$, Vivace recalibrates the value of k in the formula $\omega = \omega_0 + k \cdot \delta$ to be the smallest value for which $\Delta_r \leq r\omega$. k is reset to 0 when the direction of rate adjustment changes (e.g., from increase to decrease).

4.3 Contending with Unreliable Statistics

In general, accurate measurements require long observation time, yet that slows reaction to changing environments. We next discuss the ideas Vivace incorporates to address this challenge.

Estimating the RTT gradient via linear regression. The RTT gradient $\frac{d(RTT_x)}{dt}$ in MI x could be estimated by quantifying the RTT experienced by the first packet and the last packet sent in that MI. To estimate the RTT gradient more accurately, we utilize linear regression. Vivace assembles the 2-dimensional data set of (sampled packet RTT, time of sampling) for the packets in a MI, and uses the linear-regression-generated slope (the "β coefficient") as the RTT gradient.

Low-pass filtering of RTT gradient. Non-congestion-induced latency jitters often occur, e.g., because of recovery from packet losses in the physical layer (especially on wireless links), software packet processing devices, forwarding path flaps, or simply measurement errors due to processing time at the end-host networking stack. When Vivace employs a latency-sensitive utility function, this can result in misinformed decisions. To resolve this, Vivace leverages a low-pass filtering mechanism that treats latency gradient measurement smaller than $flt_{latency}$ as 0, to ignore small, brief latency jitters.

Double checking abnormal measurements. Occasionally, measurements lead to "counterintuitive" observations. We address the specific case that sending faster results in lower loss. While in general we avoid assumptions about the network, even with complex conditions it is highly unlikely that sending faster is the *cause* of lower loss; more likely, this is due to measurement noise or changing conditions (e.g., another sender reducing its rate). In this abnormal situation, Vivace "double checks" by re-running the same pair of rates. If this produces the same outcome in terms of which rate has higher utility, Vivace averages the utility-gradients; otherwise it throws out the original abnormal measurement.[2]

[2] In our experiments, double checking is mostly triggered during changing network conditions and multiflow competition, and lowers the packet loss rate with almost no influence on throughput, *e.g.*, turning on double-checking lowers Vivace's converged congestion loss from close to 7% to 5%. We omit these results due to limited space.

MI timeout. Generally, all information regarding packets sent during an MI will be returned after approximately one RTT. However, in the case of sudden network condition changes, a large number of packets can be lost or delayed. The measurements Vivace did before the sudden change are no longer meaningful. Therefore, if Vivace has not learned the fate of all packets sent in an MI when a certain timeout $T_{timeout}$ (a certain number of RTTs) expires, Vivace halves the sending rate.

4.4 TCP Friendliness

A common requirement from new congestion control schemes is to fairly share bandwidth with existing TCP connections (e.g., CUBIC). Attaining perfect friendliness to TCP can be at odds with achieving high performance, as the congestion control protocol is expected to both not aggressively take over spare capacity freed by a TCP connection when it backs off, and quickly take over spare capacity freed by the very same TCP connection when it terminates. We conjecture that it is fundamentally hard for any loss-based protocol to achieve consistently high performance and at the same time be fair towards TCP. The best is to hope it does not dominate TCP too much. Worse yet, latency-aware protocols can be entirely dominated by today's prevalent loss-based TCP CUBIC. Fast TCP [29], for instance, backs off as latency deviates from the minimal latency due to TCP CUBIC continuously filling the network buffer.

How, then, can a rate-control protocol both optimize latency and avoid being "killed" by loss-based TCP connections? We argue that the combination of Vivace's utility function and its rate control algorithm is a big step in this direction. Informally, Vivace captures the objective that can be expressed as "care about latency when your rate selection makes a difference". To see this, consider the scenario that a Vivace sender is the only sender on a certain link. It tries out two rates that exceed the link's bandwidth, and the buffer for that link is not yet full. Vivace's utility function will assign a higher value to the lower of these rates, since the achieved goodput and loss rate are identical to those attained when sending at the higher rate, but the latency gradient is lower. Thus, in this context, the Vivace sender behaves in a latency-sensitive manner and reduces its transmission rate. Now, consider the scenario that the Vivace sender is sharing a link that is already heavily utilized by many loss-based protocols like TCP CUBIC and the buffer is, consequently, almost always full. When testing different rates, the Vivace sender will constantly perceive the latency gradient as roughly 0, and thus disregard latency and com-

MI duration	1 RTT
sampling step ε	0.05
initial conversion factor θ_0	1
initial dynamic boundary ω_0	0.05
dynamic boundary increment δ	0.1
RTT gradient filter threshold $flt_{latency}$	0.01
MI timeout $T_{timeout}$	4 RTT
confidence amplifier $m(\tau)$	τ $(\tau \leq 3)$ $2\tau - 3$ $(\tau > 3)$

Table 1: Vivace's rate control default parameters

pete against the TCP senders over the link capacity, effectively transforming into a loss-based protocol. Our experimental results in § 5.3 illustrate this intuition.

5 Implementation and Evaluation

We implemented a user-space prototype of Vivace based on UDT [14]. We set the specific parameters for t, b, c based on our theoretical analyses in § 3. We first set $t = 0.9$, satisfying the requirement that $t < 1$ in our family of utility functions, and then adjust the remaining parameters according to that t. We set $b = 900$ so as to achieve (in theory) no inflation in latency with up to 1000 competing senders on a 1000 Mbps bottleneck link, as established in Theorem 2. We set the parameter $c = 11.35$ so as to endure up to 5% random packet-loss rate according to the formula in §3.2 that $c = \frac{tC^{t-1}}{p}$. We believe these are reasonable design choices in practice, and note that our analysis enables tuning this parameter to accommodate other scenarios. Unless stated otherwise, our evaluation of Vivace uses the parameter default values in Table 1. For BBR, we use the *net-next* Linux kernel v4.10 [3].

Setup. We report on our experimental results with Vivace under emulated realistic network conditions, in the Internet, and in emulated application scenarios.

To cleanly separate Vivace's loss-related properties from its latency-related properties, our experiments sometimes involve evaluating Vivace when the latency penalty coefficient is $b = 0$, i.e., studying a purely loss-based variant of Vivace. We refer to this variant of Vivace as "*Vivace-Loss*" and to Vivace with the default parameter assignment as "*Vivace-Latency*".

5.1 Consistent High Performance

5.1.1 Resilience to Random Loss (Fig. 2)

Using Emulab [30], we evaluate the throughput of Vivace with a single flow on a link with 100 Mbps bandwidth, 30 ms RTT, 75 KB buffer, and varying random loss rate, and compare it with Allegro, BBR, and two TCP variants. As shown in Figure 2, both Vivace variants and Allegro achieve more than 90 Mbps throughput when the random loss rate is at most 3%, and remain above 80 Mbps until 3.5% loss rate. After that point, corresponding to the employed 5% loss resistance in the utility functions, their throughput reduces to $\frac{1}{10}$ of link capacity. Vivace does not achieve full capacity at close to 4% random loss due to temporary bursty losses, which may exceed 5% in some monitor intervals.

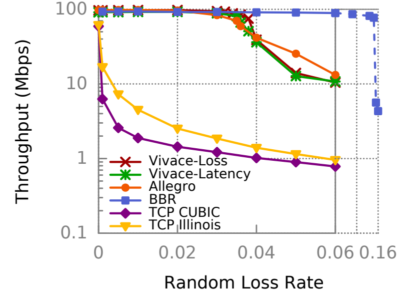


Figure 2: Random loss resilience

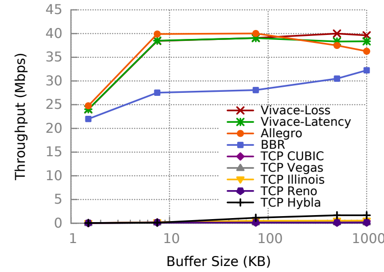


Figure 3: Long RTT tolerance

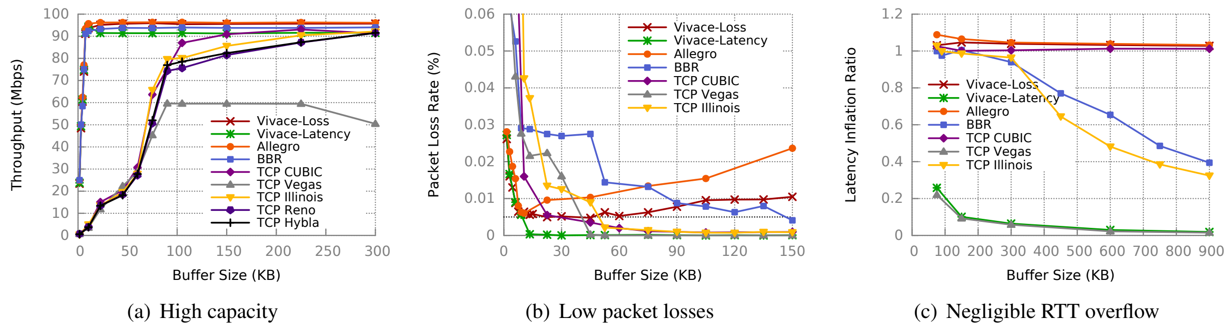
BBR keeps close-to-capacity throughput until 15% loss. By tuning Vivace's utility parameter c, we can achieve similarly high resilience to random loss. However, the theoretical insights (§ 3.2) and experiments we later present (§ 5.2.2) suggest that BBR's higher loss resilience induces comparable congestion loss with multiple competing flows, which we think is a less reasonable design choice. Finally, Figure 2 also shows gains of 20-50× over TCP family protocols.

5.1.2 High Performance on Satellite Links (Fig. 3)

We set up an emulated satellite link (as in [11]) with 42 Mbps bandwidth, 800 ms RTT and 0.74% random loss. Figure 3 shows the throughput achieved. Both Vivace-Loss and Vivace-Latency perform at least similar to Allegro, outperforming all the other protocols. Specifically, Vivace reaches more than 90% link capacity with a 7.5KB buffer, in which case it is more than 40% larger than BBR. When the buffer size increases to 1000KB, the two Vivace flavors are at least 20% better than BBR as well, while the throughput of Allegro starts to fall. We also observed 20-300× higher performance compared to the best-in-class TCP variant.

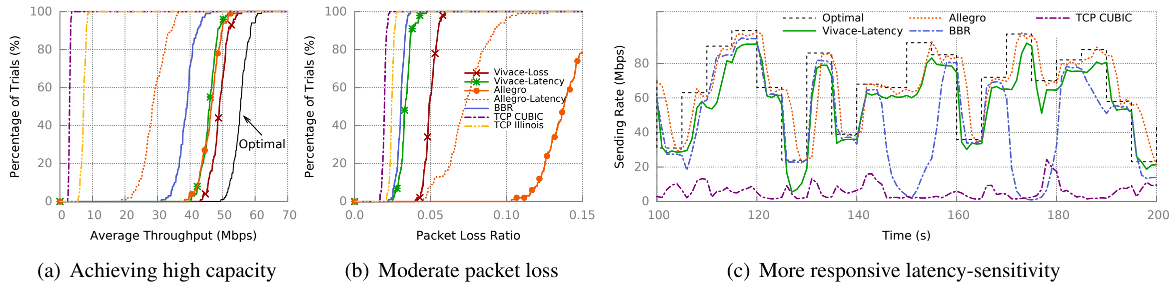
5.1.3 High Throughput without Bufferbloat (Fig. 4)

To demonstrate the effect of Vivace's latency-aware and provably fair utility function framework, we evaluate (using Emulab) its throughput and latency performance on a link with 100 Mbps bottleneck bandwidth, 30 ms RTT, and varying buffer size. On the throughput front, as



(a) High capacity (b) Low packet losses (c) Negligible RTT overflow

Figure 4: Vivace can achieve better performance with shallow buffer



(a) Achieving high capacity (b) Moderate packet loss (c) More responsive latency-sensitivity

Figure 5: Vivace can adapt to rapidly changing network conditions

shown in Figure 4(a), to achieve more than 90 Mbps, Vivace, BBR and Allegro only need a shallow queue of 7.5KB, which is 95% smaller than needed by CUBIC.

We next study how small a buffer each protocol requires to achieve minimal latency inflation and near-lossless data transfer (less than 0.5% loss rate). As shown in Figure 4(b)[3], Vivace-Latency only needs a 13.5 KB buffer to guarantee nearly zero (less than 0.5%) loss, which is 55%, 70%, 74.3%, and 91% smaller than that of CUBIC, Vegas, Illinois, and BBR, respectively. Meanwhile, Vivace-Loss and Allegro exhibit interesting behavior as maximum buffer length grows. Initially, like all protocols, their loss rate decreases as the buffer becomes able to handle the inevitable randomness in packet arrival. But later, loss increases because a larger buffer (which these protocols fill) increases RTT and slows reaction when the buffer occasionally overflows. Even so, Vivace-Loss has lower loss than Allegro in all cases.

Finally, Figure 4(c) compares the latency inflation ratio, computed as the 95th percentile of self-inflicted RTT divided by the maximal possible latency inflation with given buffer size (when the buffer is full). Vivace-Latency's latency inflation ratio is kept small with its absolute RTT overflow always below 2ms. However, both TCP Illinois and BBR have close to 100% inflation ratio (i.e., an almost full buffer) for as large as 300 KB buffer. When the buffer size is as large as 2 BDP (750 KB), Vivace-Latency still has more than 90% less latency inflation ratio than BBR. BBR's performance disadvantage may be due to its white-box assumptions about the buffer size. Vegas achieves good performance on latency inflation by sacrificing the ability to fully utilize the available bandwidth (as shown in Figure 4(a)). As expected, Vivace-Loss, Allegro, and TCP CUBIC have around 100% inflation ratio, because they lack latency awareness.[4] In sum, Vivace achieves superior latency-awareness and high throughput at the same time.

5.1.4 Swift Reaction to Changes (Figures 5-6)

We next demonstrate how Vivace's online learning rate control significantly improves the reactiveness to dynamically changing network conditions.

Emulated changing networks. We start with a network on Emulab where the RTT, bottleneck bandwidth, and random loss rate all change every 5 seconds with uniform distribution ranging from 10-100 ms, 10-100 Mbps, and 0-1%, respectively. For each protocol, we repeat the experiment 100 times with 500 sec duration each, and calculate the cumulative distribution of average throughput and packet loss rate. Allegro's latency-based utility function, which does not guarantee fairness and convergence, is also evaluated (denoted Allegro-Latency).

Figure 5(a) shows Vivace-Loss achieves the highest average throughput. Quantitatively, it reaches 49Mbps in median case, which is 88.3% of the optimal, corresponding to a gain of 5.4%, 25.6%, 72.5%, 5.7×, and 15.3× compared with Allegro, BBR, Allegro-Latency, TCP Illinois, and CUBIC, respectively. Vivace-Latency

[3] For conciseness, we leave out TCP Reno and Hybla, which have similar performance as the presented TCP variants.

[4] Although TCP CUBIC considers per-packet latency, it still cannot avoid severe buffer bloat, which explains its high inflation ratio.

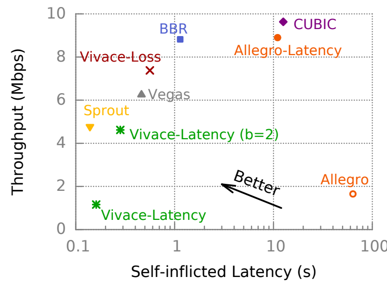


Figure 6: LTE throughput *vs.* self-inflicted latency

performs similarly to Allegro, still with a median gain of 17.9%, 62.0%, 5.3×, and 14.3× over BBR, Allegro-Latency, TCP Illinois, and CUBIC.

To further demonstrate Vivace's reactivity, we compare different protocols' packet loss. As shown in Figure 5(b), the median case loss rates of Vivace-Loss and Vivace-Latency are only 4.9% and 3.3%. Specifically, Vivace-Latency has similar median loss as BBR (but higher throughput), while outperforming Allegro and Allegro-Latency by 55.7% and 75.8%. This is because Allegro's fixed-rate control algorithm reduces rate too slowly when available bandwidth suddenly decreases.

Figure 5(c) illustrates the behavior of several of the protocols across time. BBR occasionally suffers sudden rate degradation; we find this is associated with increases in latency, to which BBR reacts badly. Allegro reduces rate slower than Vivace-Loss when the bandwidth drops, which explains its higher packet loss rate.

LTE networks. An even more challenging network scenario, as suggested by [32, 36], is the LTE environment where very deep queues are accompanied by drastically changing available bandwidth in a matter of milliseconds. On one hand, this extremely dynamic environment requires a long measurement time to prune out random noise. On the other hand, if Vivace takes too long to measure, network conditions may have drastically changed, invalidating previous measurements.

We use Mahimahi [26] to replay the Verizon-LTE trace provided by [32]. We compare Vivace with Allegro-Latency, BBR, CUBIC, Vegas and Sprout [32]. Figure 6 shows the achieved tradeoff between throughput and self-inflicted latency (as defined in [32]). Allegro, due to its overly aggressive behavior, significantly inflates latency, and also fails to deliver good throughput. Vivace-Loss reduces latency by 50.7%, 94.9%, and 95.5% compared to BBR, Allegro-Latency, and TCP CUBIC, at the cost of only 16.3%, 17.1%, and 23.4% smaller throughput, respectively. However, Vivace is still suboptimal. Compared with the best-in-class TCP (Vegas), Vivace-Loss has 17.7% larger throughput, but also 21.6% larger latency. Sprout, which is specifically designed for cellular networks with an explicit cellular link measurement model and receiver-side feedback changes

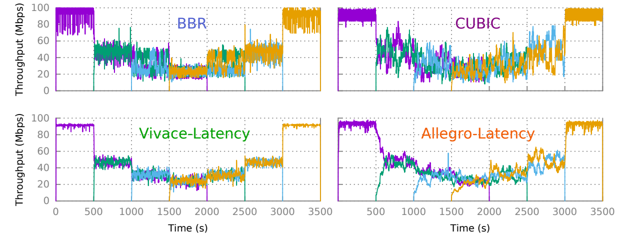


Figure 7: Vivace has fair and stable convergence

to TCP, outperforms Vivace-Loss with 75.2% shorter latency and only 35.6% smaller throughput. Furthermore, Vivace-Latency is impeded by the noisy latency measurements, and achieves the smallest throughput.

We also test with a smaller latency coefficient, $b = 2$. The consequence is supporting fewer competing senders, but the typical resource allocation in LTE networks [34] lowers the possibility of competition by many concurrent flows. This improves performance, with 39.0% lower latency and 26.3% smaller throughput than Vegas, but still falls short of Sprout. Vivace's performance in LTE networks may further improve through better reaction to noisy environments; we leave this to future work.

5.2 Convergence Properties

We next demonstrate that Vivace improves the convergence speed vs. stability tradeoff compared to state-of-the-art protocols. We also experimentally show the tradeoff between congestion loss and random loss resilience.

5.2.1 Convergence Speed and Stability (Fig. 7-8)

Temporal behavior of convergence. We set up a dumbbell topology on Emulab to demonstrate convergence performance with 4 flows sharing a link with 100 Mbps bandwidth, 30 ms RTT, and 75 KB buffer. Figure 7 shows the convergence process of several protocols with 1s granularity. Vivace achieves fair rate convergence among competing flows and is more stable than BBR and CUBIC. Compared with Allegro-Latency, which does not have any convergence guarantee, Vivace's default latency-aware utility function achieves significantly better convergence speed and stability at the same time.

Better convergence speed-stability tradeoff. We measure the quantitative trade-off between speed and stability of convergence, reproducing an experiment in [11].

On a link of 100 Mbps bandwidth and 30 ms RTT, we let an initial flow run for 10 s, which is significantly longer than needed for its convergence, then start a second flow. The convergence time is calculated as the time from the second flow's entry to the earliest time after which it maintains a sending rate within ±25% of its ideal fair share (50 Mbps) for at least 5s. The convergence stability is calculated as the standard deviation of throughput of the second flow after its convergence.

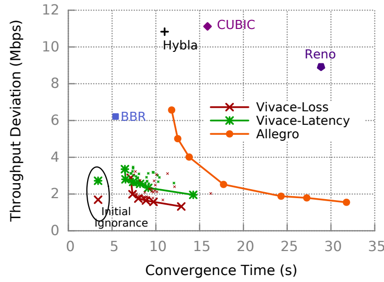


Figure 8: Better tradeoff curve

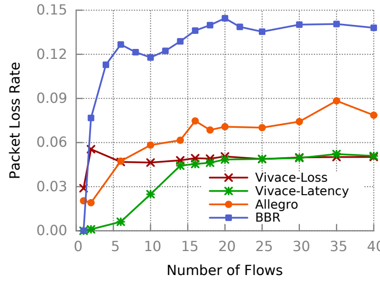


Figure 9: Multi-flow congestion

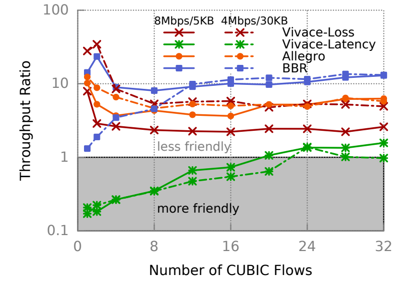


Figure 10: More friendly to TCP

To produce a tradeoff curve in Vivace, we vary parameters that affect response speed within a certain range ($1.0 \leq \theta_0 \leq 1.5$, $0.05 \leq \omega_0, \delta \leq 0.1$); we show all resulting points and highlight the lower-left Pareto front.

Figure 8 illustrates that there is a "virtual wall" in the trade-off plane at around 10s convergence time that neither TCP variants nor Allegro can pass even by trading off stability. Interestingly, BBR and Vivace penetrate that wall. Vivace, by default, achieves significantly better tradeoff points. Both Vivace-latency and Vivace-loss achieve similar convergence stability as Allegro, but using nearly 60% smaller convergence time. With convergence speed slightly higher than BBR, both Vivace variants have around 50% smaller throughput deviation. These improved trade-offs demonstrate the effectiveness of Vivace's no-regret online learning scheme.

However, any of the protocols here could choose to adopt more aggressive slow-start algorithms. As a simple example, we implemented an "initial ignorance" of loss (first 50 lost packets) and latency inflation (gradient smaller than 0.2), resulting in 37% faster convergence, and better stability, than BBR. In general, the startup algorithm is orthogonal to long-term rate control, and more advanced techniques like [22] might improve both BBR and Allegro; we leave this to future work.

5.2.2 Lower Price for Loss Resilience (Figure 9)

As we analyzed in § 3.2, resilience to random loss comes at the cost of sustaining packet-loss after convergence when the number of senders increases. Importantly, this is only the theoretical "minimal price" one has to pay to endure random packet loss within the Vivace framework. Due to the network dynamics from multi-sender interaction and efficiency of rate control algorithm, the actual price can be even higher.

To experimentally evaluate this trade-off, we set up an experiment with 30 ms RTT. We increase the number of concurrent competing flows, while proportionally increasing the FIFO queue link's total bandwidth to maintain a *per flow* 8 Mbps and 25 KB (close to 1 BDP) buffer share on average. Figure 9 shows the average packet loss per flow as the number of flows increases.

We observe that the packet loss rate of Vivace-Loss converges at the theoretical bound of 5%. Even though BBR does not fall into Vivace's online learning analysis framework, it shows a surprisingly similar tradeoff: it endures more random loss but pays a much higher price (14% loss) compared to Vivace. Though one might argue that high congestion loss is fine as long as the final goodput reaches full link utilization, this is often not true, e.g., high loss rate can cause additional delays for key frames, resulting in a lag in interactive or video streaming applications; and the large amount of transmission can cause additional energy burden on mobile devices.

In light of this discovery, we urge future congestion control designs to carefully consider this tradeoff. Though Allegro also achieves 5% random loss resilience similar to Vivace, due to its naïve rate control algorithm, it pays a higher convergence loss (9%) price. BBR, positioned as a latency-aware protocol, also has the same effect. We also observe that though Vivace-latency maintains low loss rate when there is only a single flow, its loss rate still grows as the number of concurrent senders increases. This may be due to factors not modeled in our theoretical analysis, including measurement noise and the fact that the large number of senders are constantly probing rather than staying in a perfect equilibrium. We hope to study this congestion loss in the future.

5.3 Improved Friendliness to TCP

We set up a 30 ms RTT bottleneck link with one flow using a new protocol (BBR, Allegro, or Vivace) and competing with an increasing number of CUBIC flows. As the number of senders increases, we also increase the total bandwidth and bottleneck buffer to maintain the same per-flow share, as before. We used two per-flow share settings: (4Mbps, 30KB) and (8Mbps, 5KB), corresponding to 2BDP and 0.12BDP of buffer size, respectively. Figure 10 shows the ratio between the throughput of the new-protocol flow and average throughput per CUBIC flow. A ratio of 1 in Figure 10 indicates perfect friendliness; larger ratios indicate aggressiveness towards CUBIC.

Vivace-Latency behaves as expected in the design (§4.4). When the number of CUBIC flows is small, since the queue is not always full, Vivace-Latency finds it can

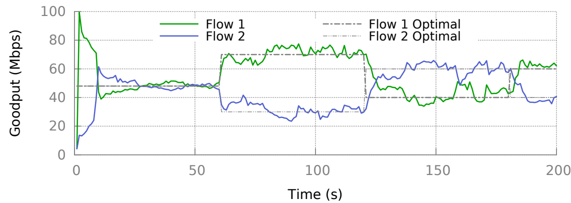


Figure 11: Flexible equilibrium by tuning utility knobs

reduce RTT by reducing its rate, and thus achieves lower throughput than CUBIC flows. However, as the number of CUBIC senders increases, it achieves the best fairness among new generation protocols. Would Vivace-Latency on the Internet still be conservative when the number of competing CUBIC flows is small? Only large scale deployment experiences can tell for sure, but our real world experiments in §5.5.2 strongly suggest positive results.

Among the evaluated protocols, BBR yields the worst TCP friendliness. In [1, 8], BBR's TCP friendliness was noted to be satisfactory, based on a single BBR flow competing with a single CUBIC flow over a large buffer (2 BDP). We successfully reproduced this specific result (the leftmost point in the 4 Mbps/30 KB (2 BDP) BBR line of Figure 10). However, we discovered that as we add more CUBIC flows, BBR becomes increasingly aggressive: it effectively treats all competing CUBIC flows as a single "bundle" with its throughput ratio increasing linearly with the number of CUBIC flows until about 16. Therefore, in practice, BBR can be very unfriendly when there are multiple competing CUBIC flows.

Though Allegro and Vivace-Loss are both loss-based, Vivace is much more agile in its reaction to competing TCP flows, especially under a shallow buffered network: its throughput ratio converges at 2.5 *vs.* Allegro's 8. In addition, though Vivace-Loss dominates CUBIC when the number of concurrent flows is small, the final converged throughput ratio (about 5 at 2 BDP) is less than half of that in BBR. In sum, though perfect TCP friendliness is fundamentally hard, we believe that Vivace provides a viable path towards adoption.

5.4 Flexible Convergence Equilibrium

With its unique utility function framework (§ 3.3), Vivace unleashes the potential to be flexible and centrally controlled. To demonstrate this capability experimentally, we set up a link with 100 Mbps bandwidth, 30 ms RTT, and two competing flows. As shown in Figure 11, we control the two flows' bandwidth share by changing their utility functions at 60s, 120s and 180s. The actual sending rate closely tracks the ideal allocation (dashed lines). This is only to illustrate the basic capability of Vivace; we leave a full-fledged system leveraging this capability to future work.

5.5 Benefits in the Real World

In addition to the above transport-level experiments on controlled networks, we test a video streaming application and performance in the wild Internet.

5.5.1 Video Streaming

We implemented a transparent proxy so that RTSP-over-TCP traffic flows from an OpenRTSP [4] client over a legacy TCP connection ending at a client-side proxy, then over a configurable transport protocol across the bottleneck link to a server-side proxy, and finally over a second legacy TCP connection to an OpenRTSP server; similar proxying occurs in reverse. We compare the streamed video's buffering ratio [10], calculated as the ratio of time spent during buffering relative to total streaming session time, using Vivace-Latency, Allegro-Latency, BBR, and TCP CUBIC. We test with four 4K videos, with 15 Mbps, 30 Mbps, 50 Mbps, and 95 Mbps average bit rate requirements in experiments in Emulab.

We first evaluate the buffering ratio with RTT changing every five seconds to a value uniform-randomly selected between 10 ms and 100 ms. We set up a link with 300 KB buffer and 0.01% random loss rate, with the network bandwidth at least 10% more than the bit rate required to stream the requested video. Fig. 12(a) shows the average buffering ratio of Vivace-Latency stays below 8%, similar to Allegro-Latency – a reduction of at least 86% and 90% compared with BBR and CUBIC.

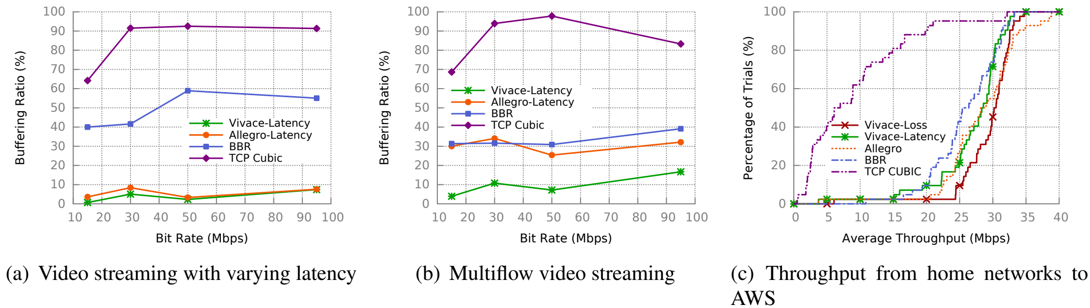
To demonstrate the application-level benefit of Vivace's stable convergence, we set up three competing streaming flows from three client-server pairs. They share a bottleneck link with 75 KB buffer, 100 ms RTT, 0.01% random loss, and adequate bandwidth for all three to stream. As shown in Figure 12(b), Vivace-Latency outperforms Allegro-Latency, BBR, and CUBIC by at least 48%, 57%, and 80%, respectively. We attribute the degraded performance of Allegro-Latency to its inferior latency awareness and reaction, and that of BBR to its high throughput variance among flows.

5.5.2 The Wild Internet

Finally, we evaluate Vivace's real-world performance in the wild Internet. We set up senders at 3 different residential WiFi networks and receivers at 14 Amazon Web Service (AWS) sites, *i.e.*, 42 sender-receiver pairs.[5] As WiFi networks have more noise in latency than wired networks, we use a slightly larger $flt_{latency} = 0.05$ to filter small variation of latency.[6] For each AWS site we test all protocols, and compute the average throughput of each protocol from five 100 sec transmissions. Figure 12(c)

[5]We test only the uplink because the virtualized AWS server impacts performance of our current user-space UDP-based packet pacing.

[6]We expect that in a full implementation, Vivace can automatically adjust $flt_{latency}$ when observing high latency variance.



(a) Video streaming with varying latency (b) Multiflow video streaming (c) Throughput from home networks to AWS

Figure 12: Performance gain in application and live Internet environments

shows the cumulative distribution of average throughput. Similar to the results in controlled networks, Vivace-Loss is slightly better than Vivace-Latency, and they both outperform BBR and CUBIC. Specifically, Vivace-Loss has a median throughput gain of 7.2%, 18.9%, and 4.0× compared with Allegro, BBR, and CUBIC, respectively.

More importantly, even though having the possibility to be overly friendly to TCP, Vivace-Latency successfully achieves 11.6% and 3.7× better throughput than BBR and CUBIC in the median, *i.e.*, CUBIC flows just cannot efficiently utilize the available bandwidth. This serves as a strong validation that Vivace provides a viable deployment path, although larger scale evaluation is desirable. Specifically, more substantial measurements are needed to answer the question of how Vivace behaves on real-world traffic patterns [28].

The root cause of the difference between Vivace-Loss and Allegro is Allegro's slow reaction to network condition variation, *e.g.* packet loss due to available bandwidth reduction or congestion. As a result, it may gain higher throughput occasionally (only after 90th percentile), but it will suffer from a higher loss rate and often yield more aggressive behavior compared to Vivace.

6 Related Work

We have already discussed Remy, and compared with several TCP variants, PCC Allegro, and BBR. We next place Vivace in the context of other related work.

In-network feedback. One class of protocols improve congestion control by providing explicit in-network feedback (e.g. available bandwidth and ECN) for better informed decisions [5, 7, 20]. These protocols yield good performance, but have been proven to be hard to deploy outside of data centers: they require coordinated change of protocols and network devices. Vivace on the other hand is compatible with the TCP message format, only requires deployment at the sender, and is therefore readily deployable.

Specially engineered congestion control. Another class of the recent works targets TCP's poor performance in specific network scenarios like LTE networks [32, 36] or data center networks [5, 24, 33] by leveraging unique insights of particular networks' behavior models, special tools or explicit feedback from the receiver. Some of these works are also moving away from TCP's hardwired control mechanism and are more like BBR; e.g., Timely [24] uses the RTT-gradient as a control signal similar to Vivace, but it still uses a hardwired rate control algorithm that maps events to fixed reactions (e.g., using fixed thresholds and step sizes). Protocols in this class provide significant performance gains, but only target very specific environments. Some of them also require changes at both endpoints [32], which may challenge deployment in practice.

Short flows. Some congestion control protocols aim to optimize performance of very short flows [22, 25]. These are complementary to Vivace, because short-flow optimization in many cases is an "open loop" problem (*i.e.* transfer as much data as possible in first few RTTs with very limited feedback) whereas Vivace targets the "closed loop" phase of data transfer (when meaningful feedback can be gathered with long enough data transfer). In fact Vivace could plausibly utilize [22, 25] as its starting phase, replacing slow-start.

7 Conclusion

We proposed Vivace, a congestion control architecture based on online optimization theory. Vivace leverages a novel latency-aware utility function framework with gradient-ascent-based online learning rate control to achieve provably fast convergence and fairness guarantees. Extensive experimentation reveals that Vivace significantly improves upon the existing state of the art in terms of performance, convergence speed, reactiveness, TCP friendliness, and more. Vivace requires sender-only changes and is hence readily deployable. We leave research questions regarding centralized resource allocation via Vivace's simple interface, and Vivace's integration into comprehensive emulators such as Pantheon [35] and production systems such as QUIC [21] and the Linux kernel, to the future.

We thank our shepherd, Alex Snoeren, and the reviewers for their helpful comments, and Google and Huawei for ongoing support of the PCC project.

References

[1] BBR talk in IETF 97. www.ietf.org/proceedings/97/slides/slides-97-iccrg-bbr-congestion-control-01.pdf.

[2] Full Proof of Theorems. http://www.ttmeng.net/pubs/vivace_proof.pdf.

[3] Linux net-next. https://kernel.googlesource.com/pub/scm/linux/kernel/git/davem/net-next.git/+/v4.10.

[4] OpenRTSP. http://www.live555.com/openRTSP/.

[5] ALIZADEH, M., GREENBERG, A., MALTZ, D., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center TCP. *Proc. of ACM SIGCOMM* (September 2010).

[6] BRAKMO, L., LAWRENCE, S., O'MALLEY, S., AND PETERSON, L. TCP Vegas: New techniques for congestion detection and avoidance. *Proc. of ACM SIGCOMM* (1994).

[7] CAESAR, M., CALDWELL, D., FEAMSTER, N., REXFORD, J., SHAIKH, A., AND VAN DER MERWE, K. Design and implementation of a routing control platform. *Proc. of NSDI* (April 2005).

[8] CARDWELL, N., CHENG, Y., GUNN, C., YEGANEH, S., AND JACOBSON, V. BBR: Congestion-based congestion control. *Queue 14*, 5 (2016), 50.

[9] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient coflow scheduling with varys.

[10] DOBRIAN, F., SEKAR, V., AWAN, A., STOICA, I., JOSEPH, D., GANJAM, A., ZHAN, J., AND ZHANG, H. Understanding the impact of video quality on user engagement. *Proc. of ACM SIGCOMM* (August 2011).

[11] DONG, M., LI, Q., ZARCHY, D., GODFREY, P. B., AND SCHAPIRA, M. PCC: Re-architecting Congestion Control for Consistent High Performance. *Proc. of NSDI* (March 2015).

[12] EVEN-DAR, M. E., MANSOUR, Y., AND NADAV, U. On the convergence of regret minimization dynamics in concave games. *Proc. of ACM symposium on Theory of computing* (2009).

[13] GHOBADI, M., YEGANEH, S., AND GANJALI, Y. Rethinking end-to-end congestion control in software-defined networks. *Proc. of HotNets* (November 2012).

[14] GU, Y. *UDT: a high performance data transport protocol*. University of Illinois at Chicago, 2005.

[15] HA, S., RHEE, I., AND XU, L. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review* (2008).

[16] HAZAN, E. Introduction to online convex optimization. http://ocobook.cs.princeton.edu/OCObook.pdf.

[17] HONG, C., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving high utilization with software-driven WAN. *Proc. of ACM SIGCOMM* (August 2013).

[18] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., AND ZHU, M. B4: Experience with a globally-deployed software defined wan. *ACM Computer Communication Review* (September 2013).

[19] JIANG, J., SUN, S., SEKAR, V., AND ZHANG, H. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. *Proc. of NSDI* (March 2017).

[20] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion control for high bandwidth-delay product networks. *Proc. of ACM SIGCOMM* (August 2002).

[21] LANGLEY, A., RIDDOCH, A., WILK, A., VICENTE, A., KRASIC, C., ZHANG, D., YANG, F., KOURANOV, F., SWETT, I., IYENGAR, J., ET AL. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 183–196.

[22] LI, Q., DONG, M., AND GODFREY, P. Halfback: Running short flows quickly and safely. *Proc. of CoNEXT* (November 2015).

[23] LIU, S., BAŞAR, T., AND SRIKANT, R. TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks. *Performance Evaluation* (2008).

[24] MITTAL, R., LAM, V. T., DUKKIPATI, N., BLEM, E. R., WASSEL, H. M. G., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., AND ZATS, D. TIMELY: RTT-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015* (2015), S. Uhlig, O. Maennel, B. Karp, and J. Padhye, Eds., ACM, pp. 537–550.

[25] MITTAL, R., SHERRY, J., RATNASAMY, S., AND SHENKER, S. Recursively cautious congestion control. *Proc. of NSDI* (March 2014).

[26] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., AND BALAKRISHNAN, H. Mahimahi: Accurate record-and-replay for HTTP. *Proc. USENIX ATC* (August 2015).

[27] SIVARAMAN, A., WINSTEIN, K., THAKER, P., AND BALAKRISHNAN, H. An experimental study of the learnability of congestion control. *Proc. of ACM SIGCOMM* (August 2014).

[28] SUN, Y., YIN, X., JIANG, J., SEKAR, V., LIN, F., WANG, N., LIU, T., AND SINOPOLI, B. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. *Proc. of ACM SIGCOMM* (August 2016).

[29] WEI, D., JIN, C., LOW, S., AND HEGDE, S. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking* (2006).

[30] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, G., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. *Proc. of OSDI* (December 2002).

[31] WINSTEIN, K., AND BALAKRISHNAN, H. TCP ex Machina: computer-generated congestion control. *Proc. of ACM SIGCOMM* (August 2013).

[32] WINSTEIN, K., SIVARAMAN, A., AND BALAKRISHNAN, H. Stochastic forecasts achieve high throughput and low delay over cellular networks. *Proc. of NSDI* (March 2013).

[33] WU, H., FENG, Z., GUO, C., AND ZHANG, Y. ICTCP: Incast congestion control for TCP in data center networks. *Proc. of CoNEXT* (November 2010).

[34] XIE, X., ZHANG, X., KUMAR, S., AND LI, L. E. pistream: Physical layer informed adaptive video streaming over lte. In *Mobicom* (2015).

[35] YAN, F. Y., MA, J., HILL, G., RAGHAVAN, D., WAHBY, R. S., LEVIS, P., AND WINSTEIN, K. Pantheon: the training ground for internet congestion-control research, 2018.

[36] ZAKI, Y., PÖTSCH, T., CHEN, J., SUBRAMANIAN, L., AND GÖRG, C. Adaptive congestion control for unpredictable cellular networks. *Proc. of ACM SIGCOMM* (August 2015).

[37] ZINKEVICH, M. Online convex programming and generalized infinitesimal gradient ascent. In *ICML* (2003), AAAI Press, pp. 928–936.

Multi-Path Transport for RDMA in Datacenters

Yuanwei Lu[†◇], Guo Chen[¶*], Bojie Li[†◇], Kun Tan[‡], Yongqiang Xiong[◇], Peng Cheng[◇], Jiansong Zhang[◇], Enhong Chen[†], Thomas Moscibroda[♯]
[†]*University of Science and Technology of China,*
[◇]*Microsoft Research,* [¶]*Hunan University,* [‡]*Huawei Technologies,* [♯]*Microsoft Azure*

Abstract

RDMA is becoming prevalent because of its low latency, high throughput and low CPU overhead. However, current RDMA remains a single path transport which is prone to failures and falls short to utilize the rich parallel paths in datacenters. Unlike previous multi-path approaches, which mainly focus on TCP, this paper presents a multi-path transport for RDMA, *i.e.* MP-RDMA, which efficiently utilizes the rich network paths in datacenters. MP-RDMA employs three novel techniques to address the challenge of limited RDMA NICs on-chip memory size: 1) a *multi-path ACK-clocking* mechanism to distribute traffic in a congestion-aware manner without incurring per-path states; 2) an out-of-order aware path selection mechanism to control the level of out-of-order delivered packets, thus minimizes the meta data required to them; 3) a *synchronise* mechanism to ensure in-order memory update whenever needed. With all these techniques, MP-RDMA only adds 66B to each connection state compared to single-path RDMA. Our evaluation with an FPGA-based prototype demonstrates that compared with single-path RDMA, MP-RDMA can significantly improve the robustness under failures (2x∼4x higher throughput under 0.5%∼10% link loss ratio) and improve the overall network utilization by up to 47%.

1 Introduction

Modern datacenter applications require high throughput and low latency networks to meet the increasing demands from customers. Compared with conventional software transport, Remote Direct Memory Access (RDMA) implements the entire transport logic in hardware network interface card (NIC) and allows direct access to remote memory, mostly bypassing CPU. Therefore, RDMA provides ultra-low latency ($\sim 1\mu s$) and high throughput (40/100Gbps) with little CPU overhead. Nowadays, RDMA has been deployed in datacenters at scale with RDMA over Converged Ethernet (RoCE) v2 [26, 49]. Existing RDMA is a single path transport, *i.e.*, an RDMA connection only flows along one network path. This single path transport is prone to path failures and also cannot utilize the rich parallel paths in modern datacenters [8, 9, 41]. While many approaches have been proposed to enhance TCP to support multi-path, none has considered RDMA. In this paper, we propose a multi-path transport for RDMA.

However, RDMA is completely implemented in NIC hardware which has very limited computing resource and on-chip memory (*e.g.*, only a few mega-bytes). Although NIC could upload local states in host memory, swapping data between on-chip memory and host memory has a cost and frequent swapping would significantly downgrades performance [32, 33] (also see §2.3). As a consequence, the key design goal for a multi-path RDMA transport is to minimize the memory footprint, which incurs three challenges.

First, a multi-path transport should track the congestion states on each path, so that it can perform congestion-aware load distribution. However, these states grow linearly with the number of sending paths. This may cause a considerable memory overhead even when a modest number of paths are used for one RDMA connection. For example, if we adopt a multi-path transport similar to MPTCP [41], we may add 368 bytes if 8 sub-flows are used. [1] However, the size of these extra states is already 50% more than the entire states of one connection in current RoCE design. [2] As a result, 33.3% fewer concurrent connections can be supported only using on-chip memory, which leads to more frequent swapping and downgrades the performance.

Second, multi-path will cause packets to arrive out-of-order at the receiver. Consequently, the receiver needs additional metadata to track whether a packet has arrived or not. However, if the paths conditions vary greatly, the size of the metadata could be large. Fig. 1 gives the 99.9% tail of the out-of-order degree (OOD) [3] of a network under various scenarios (more details in § 5.2.1). For example, consider the case that one path has degraded to 1Gbps (*e.g.*, due to hardware failures caused link rate auto-negotiation [9, 27]), while other paths re-

*This work was done when Guo Chen was a full-time employee at Microsoft Research.

[1]Each sub-connection needs to maintain states including *rcv_nxt*, *snd_nxt*, *snd_una*, *snd_ssthresh*, *snd_cwnd*, *srtt*, *rttvar*, *rtt_seq*, *map_data_seq*, *map_subseq*, *map_data_len*, ...

[2]Mellanox ConnectX Linux driver [43] maintains all the states of an RDMA connection in a 248B *mlx4_qp_context*.

[3]We define the *out-of-order degree (OOD)* here as the maximal difference between the sequence number of an out-of-order arrived packet and the expected packet sequence number.

main at a normal speed of 40Gbps. If a bitmap structure is used, the size of the bitmap would be 1.2KB. If we naively use fewer bits, any packet with a sequence number out of the range of the bitmap has to be dropped. This would reduce the performance greatly as the throughput is effectively limited by the slowest path. A core design challenge is to keep high performance even if we can only track very limited out-of-order packets.

Finally, the receiver NIC does not have enough memory to buffer out-of-order packets but has to place them into host memory as they arrive. Therefore, the data in host memory may be updated out-of-order. This may cause a subtle issue as some existing applications implicitly assume the memory is updated in the same order as the operations are posted [20, 22, 48]. For example, a process may use a WRITE operation to update a remote memory, and then issues another WRITE operation to set a dirty flag to notify a remote process. If the second WRITE updates memory before the first WRITE, the remote process may prematurely read the partial data and fails. While retaining the memory updating order is trivial for single-path RDMA, it requires careful design in multi-path RDMA to avoid performance downgrade.

This paper presents MP-RDMA, the first multi-path transport for RDMA that addresses all aforementioned challenges. Specifically, MP-RDMA employs a novel *multi-path ACK-clocking* mechanism that can effectively do congestion-aware packets distribution to multiple paths without adding per-path states. Second, we design *an out-of-order aware path selection algorithm* that proactively prunes slow paths and adaptively chooses a set of paths that are fast and with similar delays. This way, MP-RDMA effectively controls the out-of-order level so that almost all packets can be tracked with a small sized bitmap (*e.g.*, 64 bits). Finally, MP-RDMA provides an interface for programmers to ensure in-order memory update by specifying a *synchronise* flag to an operation. A synchronise operation updates memory only when all previous operations are completed. Therefore, two communication nodes can coordinate their behaviors and ensure application logic correctness.

We have implemented an MP-RDMA prototype in FPGA, which can run at the line rate of 40Gbps. We evaluate MP-RDMA in a testbed with 10 servers and 6 switches. Results show that MP-RDMA can greatly improve the robustness under path failures (2x∼4x higher throughput when links have 0.5%∼10% loss rate), overall network utilization (∼47% higher overall throughput) and average flow completion time (up to 17.7% reduction) compared with single-path RDMA. Moreover, MP-RDMA only consumes a small constant (66B) amount of extra per-connection memory, which is comparable to the overhead (∼60B) added by DCQCN [49] to enhance existing single-path RDMA.

In summary, we make the following contributions: 1) We present MP-RDMA, the first transport for RDMA that supports multi-path. 2) We have designed a set of novel algorithms to minimize the memory footprint, so that MP-RDMA is suitable to be implemented in NIC hardware. 3) We have evaluated MP-RDMA on an FPGA-based testbed as well as large-scale simulations.

2 Background and motivation

2.1 RDMA Background

RDMA enables direct memory access to a remote system through NIC hardware, by *implementing the transport entirely in NIC*. Therefore RDMA can provide low latency and high throughput with little CPU involvement on either local or remote end. RoCE v2 [4–6] introduces UDP/IP/Ethernet encapsulation which allows RDMA to run over generic IP networks. Nowadays, production datacenters, *e.g.* Microsoft Azure and Google, have deployed RoCE at scale [26, 39, 49]. Hereafter in this paper, unless explicitly stated otherwise, we refer RDMA to RoCE v2.

In RDMA terminology, an RDMA connection is identified by a pair of work queues, called *queue pair* (QP). A QP consists of a *send* queue and a *receive* queue which are both maintained on NICs. When an application initiates an RDMA *operation* (also called a *verb*) to send or retrieve data, it will post a *work queue element* (WQE) to NIC's send queue or receive queue, respectively. Moreover, to notify the application for operation completion, there is also a completion queue (CQ) associated with each QP. On completing a WQE, a completion queue element (CQE) will be delivered to the CQ. There are four commonly used verbs in RDMA: SEND, RECV, WRITE and READ. Among these, SEND and RECV are *two-sided*, meaning that SEND operation always requires a RECV operation at the other side. READ and WRITE are *one-sided* operations, meaning that applications can directly READ or WRITE pre-registered remote memory without involving remote CPU.

RDMA transport is message-based, *i.e.* an RDMA operation is translated into a *message* for transmission. Then an RDMA message will be divided into multiple equal-sized *segments* which are encapsulated into UDP/IP/Ethernet packet(s). In RoCEv2, all RDMA packets use an identical UDP destination port (4791), while the UDP source port is arbitrary and varies for different connections, which allows load-balancing. An RDMA header is attached to every packet. The header contains a *packet sequence number* (PSN) which provides a continuous sequence number for the RDMA packets in a connection. At the receiver side, RDMA messages are restored according to PSN. Moreover, an RDMA receiver may generate an ACK or a Negative ACK (NACK) to notify the sender for received or lost packets.

RDMA transport requires a lossless network provided by priority-based flow control (PFC) [1, 19]. Specifically, PFC employs a hop-by-hop flow control on traffic with pre-configured priorities. With PFC, when a downstream switch detects that an input queue exceeds a threshold, it will send a PAUSE frame back to the upstream switch. While PFC can effectively prevent switches from dropping packets, the back-pressure behavior may propagate congestion and slow down the entire network. Thus, end-to-end congestion control mechanisms have been introduced into RoCE. For example, DCQCN [49] enhances RoCE transport with explicit congestion notification (ECN) and quantized congestion notification (QCN) [30] to control congestion.

2.2 Need for Multi-Path Transmission

Current RDMA transport mandates a connection to follow one network path. Specifically, packets of one RDMA connection use the same UDP source and destination ports. There are two major drawbacks for such single-path transmission.

First, single path transmission is prone to path failures. Some minor failures along the path can greatly affect the performance of upper-layer applications. For example, silent packet loss is a common failure in datacenter [18, 27]. Since RDMA transport is implemented in hardware which typically lacks resources to realize sophisticated loss recovery mechanism, it is very sensitive to packet loss. As a result, a small loss rate (*e.g.*, 0.1%) along the transmission path can lead to dramatic RDMA throughput degradation (*e.g.*, $<\sim$60%) [49].

Second, single path falls short to utilize the overall network bandwidth. Equal Cost Multi-Path (ECMP) routing is currently the main [26, 44, 45] method to balance RDMA traffic among the datacenter network fabrics. Basically, ECMP hashes different connections to different paths. However, as many prior studies pointed out [15, 26], ECMP is not able to balance traffic well when the number of parallel paths is large [7, 25] due to hash collisions. While some part of the network is highly congested, the rest may often have a low traffic load, reducing the overall network utilization. Therefore, it is important to spread traffic in finer granularity than flow among multiple paths to achieve high network utilization [15, 41].

In literature, a set of mechanisms have been proposed to distribute traffic in finer-grained ways to efficiently utilize the rich network paths in datacenters [8, 9, 14, 15, 21, 28, 34, 40–42]. Unfortunately, most of these previous arts only consider TCP traffic, and none of them explicitly discuss RDMA (see §6 for more discussions). As we will show in §2.3, RDMA is quite different from TCP in many aspects. Therefore, in this paper, we design the first multi-path transport for RDMA.

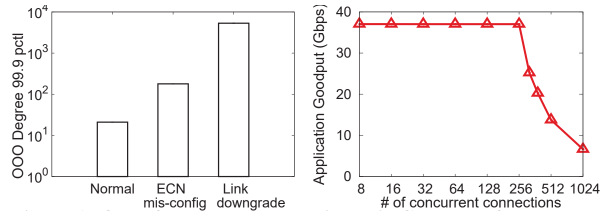


Figure 1: Out-of-order degree under different scenarios.

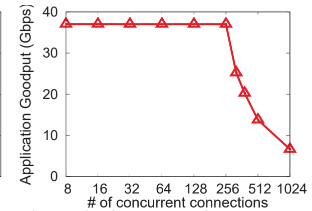


Figure 2: Goodput of concurrent MLNX CX3 Pro WRITEs.

2.3 Challenges for Multi-Path RDMA

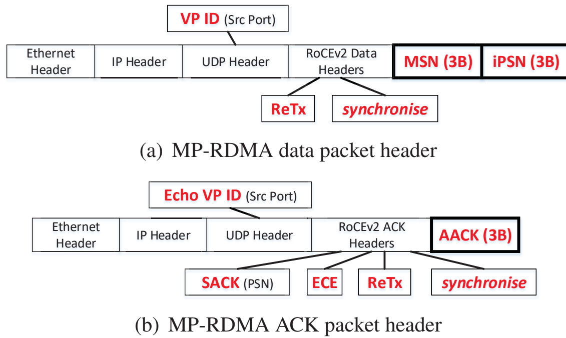
RDMA is implemented in NICs. But usually, on-chip memory in NIC is small and expensive. Populating large memories in NIC hardware is very costly. Not only memory blocks require many transistors and may occupy a large die area, but also significantly increase the power consumption, which may cause further issues like cooling. Thus NICs usually serve as a cache of host memory to store the connection states. If a cache miss happens, RDMA NIC needs to access the host memory via PCIe. Frequent cache misses lead to NIC throughput degradation due to the PCIe bus latency and the contention on the bandwidth. To illustrate the impact of cache misses on application goodput, we use 4 clients with Mellanox ConnectX 3 Pro NICs to initiate RDMA WRITEs to a single server and measure the total goodput. Fig. 2 shows that when the number of concurrent connections is larger than 256, application goodput would drop sharply. This is because to perform WRITE operations, the receiving NIC needs to access corresponding connection states (QP context). When the number of connections is larger than 256, not all states can be stored in NIC's memory. With more concurrent connections, cache misses occur more frequently. This result conforms with previous work [32, 33]. Thus, to avoid performance degradation caused by frequent cache misses, the memory footprint for each RDMA connection should be minimized to support more connections in on-chip memory. This key uniqueness of RDMA brings several challenges for designing MP-RDMA as aforementioned (§1).

3 MP-RDMA design

3.1 Overview

MP-RDMA is a multi-path transport for RDMA while effectively addresses the challenge of the limited on-chip memory in NIC hardware. MP-RDMA employs a novel ACK-clocking and congestion control mechanism to do congestion-aware load distribution without maintaining per-path states (§3.2). Moreover, it uses an out-of-order aware path selection mechanism to control the out-of-order degree among sending paths, thus minimizes the meta data size required for tracking out-of-order packets (§3.3). Finally, MP-RDMA provides a *synchronise* mechanism for applications to ensure in order host memory update without sacrificing throughput (§3.4).

MP-RDMA assumes a PFC enabled network with



(a) MP-RDMA data packet header

(b) MP-RDMA ACK packet header

Figure 3: MP-RDMA packet header format. Fields with red bold text are specific for MP-RDMA.

RED [24] supported. It reuses most of the existing/reserved fields (with thin border) in the UDP and RoCE v2 headers. It extends the existing headers by certain fields (with thick border) (Fig. 3). MP-RDMA controls the transmission paths of a packet by selecting a specific source port in the UDP header and let ECMP pick up the actual path. Since packets with the same source port will be mapped to the same network path, we use a UDP source port to identify a network path, which is termed as a *Virtual Path* (VP). Initially, the sender picks a random VP for a data packet. Upon receiving a data packet, the receiver immediately generates an ACK which encodes the same VP ID (Echo VP ID field). The ACK header carries the PSN of the received data packet (SACK field) as well as the accumulative sequence number at the data receiver (AACK field). ECN signal (ECE field) is also echoed back to the sender.

The data of a received packet is placed directly into host memory. For WRITE and READ operations, the original RDMA header already embeds the address in every data packet, so the receiver can place the data accordingly. But for SEND/RECV operations, additional information is required to determine the data memory placement address. This address is in a corresponding RECV WQE. MP-RDMA embeds a *message sequence number* (MSN) in each SEND data packet to assist the receiver for determining the correct RECV WQE. In addition, an intra-message PSN (iPSN) is also carried in every SEND data packet as an address offset to place the data of a specific packet within a SEND message.

Next, we zoom into each design component and elaborate how they together can achieve high performance with a small MP-RDMA on-chip memory footprint.

3.2 Congestion control and multi-path ACK-clocking

As aforementioned, MP-RDMA performs congestion control without maintaining per-path states, thus minimizing on-chip memory footprint. MP-RDMA uses one congestion window for all paths. The congestion control algorithm is based on ECN. MP-RDMA decreases its *cwnd* proportionally to the level of congestion, which is similar to DCTCP [49]. However, unlike DCTCP that estimates the level of congestion by computing an average ECN ratio, MP-RDMA reacts directly upon ACKs. As packets are rarely dropped in an RDMA network, reacting to every ACK would be precise and reliable. Moreover, it is very simple to implement the algorithm in hardware. MP-RDMA adjusts *cwnd* on a per-packet basis:

For each received ACK:

$$cwnd \leftarrow \begin{cases} cwnd + 1/cwnd & \text{if } ECN = 0 \\ cwnd - 1/2 & \text{if } ECN = 1 \end{cases}$$

Note that on receiving an ECN ACK, *cwnd* is decreased by 1/2 segment instead of cutting by half.

MP-RDMA employs a novel algorithm called *multi-path ACK-clocking* to do congestion-aware packets distribution, which also allows each path to adjust its sending rate independently. The mechanism works as follows: *Initially, the sender randomly spreads initial window (IW) wise of packets to IW initial VPs. Then, when an ACK arrives at the sender, after adjusting cwnd, if packets are allowed, they are sent along the VP carried in the ACK.* In § 3.2.1, fluid models show that with per-packet ECN-based congestion control and multi-path ACK clocking, MP-RDMA can effectively balance traffic among all sending paths based on their congestion level. It is worth noting that MP-RDMA requires per-packet ACK, which adds a tiny bandwidth overhead ($<4\%$) compared to convention RDMA protocol.

MP-RDMA uses a similar way as TCP NewReno [23] to estimate the *inflight* packets when there are out-of-order packets being selectively acked. [4] Specifically, we maintain an *inflate* variable, which increases by one for each received ACK. We use *snd_nxt* to denote the PSN of the highest sent packet and *snd_una* to denote the PSN of the highest accumulatively acknowledged packet. Then the available window (*awnd*) is:

$$awnd = cwnd + inflate - (snd_nxt - snd_una).$$

Once an ACK moves *snd_una*, $inflate$ is decreased by $(ack_aack - snd_una)$. This estimation can be temporarily inaccurate due to the late arrival of the ACKs with SACK PSN between the old *snd_una* and new *snd_una*. However, as *awnd* increases only one per ACK, our ACK clocking mechanism can still work correctly.

3.2.1 Fluid model analysis of MP-RDMA

Now we develop a fluid model for MP-RDMA congestion control. For clarity, we first establish a single-path model for MP-RDMA to show its ability to control

[4] Alternatively, we could use a sender-side bitmap to track *sacked* packets. But the memory overhead of this bitmap could be large for high-speed networks. For example, for 100Gbps network with 100μs delay, the size of the bitmap can be as large as 1220 bits.

the queue oscillation. Then a multi-path model is given to demonstrate its ability in balancing congestion among multiple paths. We assume all flows are synchronized, *i.e.* their window dynamics are in phase.

Single-path model. Consider N long-lived flows traversing a single-bottleneck link with capacity C. The following functions describe the dynamics of $W(t)$ (congestion window), $q(t)$ (queue size). We use $R(t)$ to denote the network RTT, $F(t)$ to denote the ratio of ECN marked packets in the current window of packets. d is the propagation delay. We further use $R^* = d + average_queue_length/C$ to denote the average RTT. MP-RDMA tries to strictly hold the queue length around a fixed value, thus R^* is fixed:

$$\frac{dW}{dt} = \frac{1-F(t-R^*)}{R(t)} - \frac{W(t)}{2R(t)}F(t-R^*) \quad (1)$$

$$\frac{dq}{dt} = N\frac{W(t)}{R(t)} - C \quad (2)$$

$$R(t) = d + \frac{q(t)}{C} \quad (3)$$

The fix point of Equation (1) is: $W(t) = \frac{2(1-F)}{F}$. The queue can be calculated as $q = NW(t) - CR$, which gives:

$$q(t) = \frac{N(1-F)}{F} - \frac{Cd}{2} \quad (4)$$

MP-RDMA requires RED marking at the switch [24]:

$$p = \begin{cases} 0 & \text{if } q \leqslant K_{min} \\ P_{max}\frac{(q-K_{min})}{K_{max}-K_{min}} & \text{if } K_{min} < q \leq K_{max} \\ 1 & \text{if } q > K_{max} \end{cases} \quad (5)$$

Combining Equation (4) and (5) yields the fix point solution (q, W, F). We consider two different ECN marking schemes: 1) standard RED [24]; 2) DCTCP RED ($K_{max} = K_{min}, P_{max} = 1.0$). **With standard RED marking, MP-RDMA achieves a stable queue with small oscillation.** If DCTCP RED is used, as MP-RDMA doesn't use any history ECN information, MP-RDMA can be modeled as a special case of DCTCP with $g = 1$. As a result, The queue oscillation would be large [11].

We use simulations to validate our analysis. 8 flows each with output rate 10Gbps, compete for a 10Gbps bottleneck link. RTT is set to 100μs. For standard RED, we set $(P_{max}, K_{min}, K_{max}) = (0.8, 20, 200)$. For DCTCP RED, we set $(P_{max}, K_{min}, K_{max}) = (1.0, 65, 65)$. According to Fig. 4, with standard RED, MP-RDMA's queue length varies very little compared with theoretical results. And the queue oscillation is much smaller than DCTCP RED. Full throughput is achieved under both marking schemes.

Multi-path model. Now we develop the multi-path model. Let VP_i denote *ith* VP. We assume VP_i has a

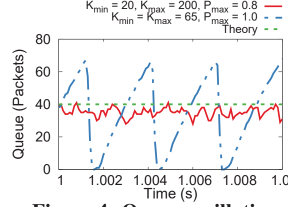


Figure 4: Queue oscillation.

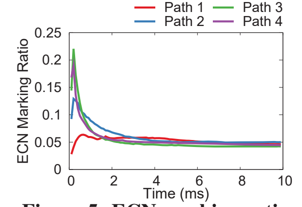


Figure 5: ECN marking ratio.

virtual cwnd denoted by w_i, which controls the number of packets on VP_i. And the total *cwnd* is given as $cwnd = \sum_i w_i$. We use ε to denote the fraction part of *cwnd*, *i.e.* $\varepsilon = cwnd - \lfloor cwnd \rfloor$. We assume ε has a uniform distribution from 0 to 1 (denoted as $U[0,1)$). [5]

An ECN ACK from VP_i will reduce *cwnd* by $1/2$ segment. There could be two situations: If $\varepsilon \geq 1/2$, a new packet can still be clocked out on path VP_i; otherwise, after reduction, the new *cwnd* will prevent a packet from sending to VP_i. Since ε is subject to $U(0,1)$, an ECN ACK reduces w_i by one with probability 50%. On the other hand, a non-ECN ACK increases *cwnd* by $1/cwnd$. If the growth of *cwnd* happens to allow one additional packet, VP_i would get two packets. As ε is subject to $U(0,1)$, such chance would be equal for each incoming non-ECN ACK, *i.e.* $1/cwnd$. In other words, a non-ECN ACK increases w_i by one with probability $w_i/cwnd$.

Based on the above analysis, we can establish the fluid model for our multi-path congestion control. Since a VP is randomly mapped to a physical path, statistically each physical path may get an equal number of VP for a long-lived MP-RDMA connection. Consider N flows, each flow distributes their traffic to M_v virtual paths, which are mapped onto M_p physical paths. We use $Path(j)$ to denote the set of virtual paths that are mapped onto physical path j. Then we have the model:

$(i = 0, 1, ..., M_v - 1;\ j = 0, 1, ..., M_p - 1)$

$$\frac{dw_i}{dt} = \frac{w_i(t)}{cwnd * R_i(t)}[1 - F_i(t - R_i^*)] - \frac{w_i(t)}{2R_i(t)}F_i(t - R_i^*) \quad (6)$$

$$\frac{dq_j}{dt} = N\frac{\sum_{i \in Path(j)} w_i}{R_j} - C_j \quad (7)$$

$$R_j(t) = d_j + \frac{q_j(t)}{C_j} \quad (8)$$

Also, each physical path j has its own RED marking curve as in Eq. (5). Eq. (6) yields the fix point solution: $F_i = \frac{2}{cwnd+2}$. As F_i only depends on the total *cwnd*, this indicates that the marking ratio F_i of each VP will be the same, so will the physical path marking ratio. In

[5] We note that this assumption cannot be easily proven as the congestion window dynamics are very complicated, but our observation on both testbed and simulation experiments verified the assumption. Later we will show that based on this assumption, our experiments and theoretical analysis results match each other very well.

other words, **MP-RDMA can balance the ECN marking ratio among all sending paths regardless of their RTTs, capacities and RED marking curves.** In datacenters where all equal-cost paths have same capacities and RED marking curves, MP-RDMA can balance the load among multiple paths.

We use simulations to validate our conclusion. 10 MP-RDMA connections are established. Each sends at 40Gbps among 8 VPs. The virtual paths are mapped randomly onto 4 physical paths with different rates, *i.e.* 20Gbps, 40Gbps, 60Gbps and 80Gbps. The network base RTT of each path is set to 16μs. For RED marking, all paths has the same $K_{min} = 20$ and $K_{max} = 200$, but with P_{max} set to different values, *i.e.* 0.2, 0.4, 0.6 and 0.8. Fig. 5 shows the ECN marking ratio of the 4 physical paths. ECN marking ratios of the 4 physical paths converge to the same value which validates our analysis.

3.3 Out-of-order aware path selection

Out-of-Order (OOO) is a common outcome due to the parallelism of multi-path transmission. This section first introduces the data structure for tracking OOO packets. Then we discuss the mechanism to control the network OOO degree to an acceptable level so that the on-chip memory footprint can be minimized.

3.3.1 Bitmap to track out-of-order packets

MP-RDMA employs a simple bitmap data structure at the receiver to track arrived packets. Fig. 6 illustrates the structure of the bitmap, which is organized into a cyclic array. The head of the array refers to the packet with $PSN = rcv_nxt$. Each slot contains two bits. According to the message type, a slot can be one of the four states: 1) *Empty*. The corresponding packet is not received. 2) *Received*. The corresponding packet is received, but not the *tail* (last) packet of a message. 3) *Tail*. The packet received is the tail packet of a message. 4) *Tail with completion*. The packet received is the tail packet of a message that requires a completion notification.

When a packet arrives, the receiver will check the PSN in the packet header and find the corresponding slot in the bitmap. If the packet is the tail packet, the receiver will further check the opcode in the packet to see if the message requires a completion notification, *e.g.*, SEND or READ response. If so, the slot is marked as *Tail with completion*; Otherwise, it is marked as *Tail*. For non-tail packets, the slots are simply set to *Received*. The receiver continuously scans the tracking bitmap to check if the head-of-the-line (HoL) message has been completely received, *i.e.*, a continuous block of slots are marked as *Received* with the last slot being either *Tail* or *Tail with completion*. If so, it clears these slots to *Empty* and moves the head point after this HoL message. If the message needs a completion notification, the receiver pops a WQE from the receive WQ and pushes a CQE in the CQ.

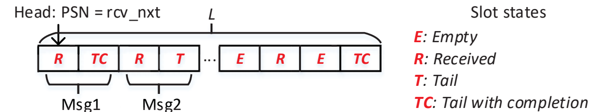


Figure 6: Data structure to track OOO packets at the receiver.

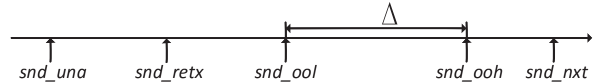


Figure 7: MP-RDMA window structure at the sender.

3.3.2 Out-of-order aware path selection

MP-RDMA employs only limited slots in the tracking bitmap, *e.g.*, $L = 64$, to reduce the memory footprint in NIC hardware. Therefore, if an out-of-order packet holds a PSN larger than $(rcv_nxt + L)$, the receiver has to drop this packet, which hurts the overall performance. MP-RDMA controls the degree of out-of-order packets (OOD) by a novel path selection algorithm, so that most packets would arrive within the window of the tracking bitmap. The core idea of our out-of-order aware path selection algorithm is to actively prune the slow paths and select only fast paths with similar delay.

Specifically, we add one new variable, snd_ooh, which records the highest PSN that has been sacked by an ACK. For the sake of description, we define another variable $snd_ool = snd_ooh - \Delta$, where $\Delta \leq L$ is a tunable parameter that determines the out-of-order level of MP-RDMA. The algorithm works as follows: *When an ACK arrives at the sender, the sender will check if the SACK PSN is lower than* snd_ool*. If so, the sender reduces cwnd by one and this ACK is not allowed to clock out a packet to the VP embedded in the ACK header.*

The design rationale is straightforward. We note that snd_ooh marks an out-of-order packet that goes through the fast path. In order to control the OOD, we need to prune all slow paths that causes an OOD larger than Δ. Clearly, an ACK acknowledges a PSN lower than snd_ool identifies such a slow path with the VP in the header. Note that PSN alone may not correctly reflect the sending order of a retransmitted packet (sent later but with lower PSN). Therefore, to remove this ambiguity, we explicitly tagged a bit in packet header to identify a retransmitted packet and echoed back in its ACK (ReTx in Fig. 3). For those ReTx ACKs, we simply regard their data packets have used good paths.

3.4 Handling synchronise operations

As discussed in §2, NIC hardware does not have enough memory to store out-of-order packets and has to place them into host memory. One possible way is to allocate a separate re-ordering buffer in host memory and temporarily store the out-of-order packets there. When the HoL message is completely received, the NIC can copy the message from the re-ordering buffer into the right memory location. This, however, causes a signification overhead as a packet may traverse PCIe bus

twice, which not only consumes double PCIe bandwidth resource but also incurs a long delay. We choose to directly place out-of-order packets' data into application memory. This approach is simple and achieves optimal performance in most cases. However, to support applications that rely on the strict order of memory updates, *e.g.*, key-value store using RDMA WRITE operations [20], MP-RDMA allows programmers to specify a *synchronise* flag on an operation, and MP-RDMA ensures that a synchronise operation updates the memory only after all previous operations are completed.

One straightforward approach is to delay a synchronise operation until the initiator receives acknowledgements or data (for READ verbs) of all previous operations. This may cause inferior performance as one additional RTT will be added to every synchronise operation. We mitigate this penalty by delaying synchronise operations only an interval that is slightly larger than the maximum delay difference among all paths. In this way, the synchronise operations should complete just after all its previous messages with high probability. With the out-of-order aware path selection mechanism (§3.3), this delay interval can be easily estimated as

$$\Delta t = \alpha \cdot \Delta / R_s = \alpha \cdot \Delta / \left(\frac{cwnd}{RTT} \right),$$

where Δ is the target out-of-order level, R_s is the sending rate of the RDMA connection and α is a scaling factor. We note that synchronise messages could still arrive before other earlier messages. In these rare cases, to ensure correctness, the receiver may drop the synchronise message and send a NACK, which allows the sender to retransmit the message later.

3.5 Other design details and discussions

Loss recovery. For single-path RDMA, packet loss is detected by the gap in PSNs. But in MP-RDMA, out-of-order packets are common and most of them are not related to packet losses. MP-RDMA combines loss detection with the out-of-order aware path selection algorithm. In normal situations, the algorithm controls OOD to be around Δ. However, if a packet gets lost, OOD will continuously increase until it is larger than the size of the tracking bitmap. Then, a NACK will be generated by the receiver to notify the PSN of the lost packet. Upon a NACK, MP-RDMA enters recovery mode. Specifically, we store the current *snd_nxt* value into to a variable called *recovery* and set *snd_retx* to the NACKed PSN (Fig.7). In the recovery mode, an incoming ACK clocks out a retransmission packet indicated by *snd_retx*, instead of a new packet. If *snd_una* moves beyond *recovery*, the loss recovery mode ends.

There is one subtle issue here. Since MP-RDMA enters recovery mode only upon bitmap overflow, if the application does not have that much data to send, RTO is triggered. To avoid this RTO, we adopt a scheme of FUSO [18] that early retransmits unacknowledged packets as new data if there is no new data to transmit and *awnd* allows. In rare case that the retransmissions are also lost, RTO will eventually fire and the sender will start to retransmit all unacknowledged packets.

New path probing. MP-RDMA periodically probes new paths to find better ones. Specifically, every RTT, with a probability p, the sender sends a packet to a new random VP, instead of the VP of the ACK. This p balances the the chance to fully utilize the current set of good paths and to find even better paths. In our experiment, we set p to 1%.

Burst control. Sometimes for a one returned ACK, the sender may have a burst of packets ($\geq$2) to send, *e.g.*, after exiting recovery mode. If all those packets are sent to the ACK's VP, the congestion may deteriorate. MP-RDMA forces that one ACK can clock out at most two data packets. The rest packets will gradually be clocked out by successive ACKs. If no subsequent ACKs return, these packets will be clocked out by a *burst_timer* to random VPs. The timer length is set to wait for outstanding packets to be drained from the network, *e.g.* 1/2 RTT.

Path window reduction. If there is no new data to transfer, MP-RDMA gracefully shrinks *cwnd* and reduce the sending rate accordingly following a principle called *"use it or lose it"*. Specifically, if the sender receives an ACK that should kick out a new packet but there is no new data available, *cwnd* is reduced by one. This mechanism ensures that all sending paths adjust their rates independently. If path window reduction mechanism is not used, the sending window opened up by an old ACK may result in data transmission on an already congested path, thus deteriorating the congestion.

Connection restart. When applications start to transmit data after idle (*e.g.* 3 RTTs), MP-RDMA will restart from IW and restore multi-path ACK clocking. This is similar to the restart after idle problem in TCP [29].

Interact with PFC. With our ECN-based end-to-end congestion control, PFC will seldom be triggered. If PFC pauses all transmission paths [26, 49], MP-RDMA will stop sending since no ACK returns. When PFC resumes, ACK clocking will be restarted. If only a subset of paths are paused by PFC, those paused paths will gradually be eliminated by the OOO-aware path selection due to their longer delay. We have confirmed above arguments through simulations. We omit the results here due to space limitation.

4 Implementation

4.1 FPGA-based Prototype

We have implemented an MP-RDMA prototype using Altera Stratix V D5 FPGA board [12] with a PCIe Gen3

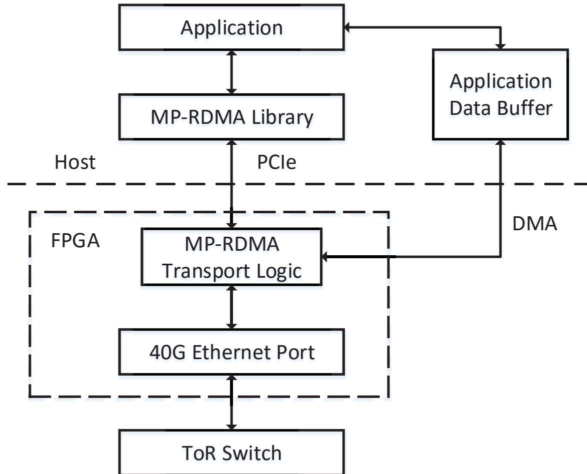


Figure 8: System architecture.

x8 interface and two 40G Ethernet ports. Fig.8 shows the overview of the prototype architecture. There are two major components: 1) MP-RDMA transport logic, and 2) MP-RDMA library. The entire transport logic is implemented on FPGA with ClickNP framework [35]. We have developed 14 ClickNP elements with ~2K lines of OpenCL code. Applications call MP-RDMA library to issue operations to the transport. FPGA directly DMAs packet data from/to the application buffer via PCIe.

Table 1 summarizes all extra states incurred per connection by MP-RDMA for multi-path transport compared to existing RoCE v2. Collectively, MP-RDMA adds additional 66 bytes. This extra memory footprint is comparable to other single-path congestion control proposals to enhance RoCE v2. For example, DCQCN [49] adds ~60 bytes for its ECN based congestion control.

4.2 Validation

We now evaluate the basic performance of the FPGA-based prototype. We measure the processing rate and latency for sending and receiving under different message sizes. Specifically, the sending/receiving latency refers to the time interval between receiving one ACK-/data packet and generating a new data/ACK packet.

To measure the processing rate for sending logic, we use one MP-RDMA sender to send traffic to two MP-RDMA receivers, creating a sender bottleneck, vice versa for measuring the receiving logic. As shown in Fig.9, our implementation achieves line rate across all message sizes for receiving. For sending, when message size is smaller than 512 bytes, the sender cannot reach the line rate. This is because sender logic is not fully pipelined due to memory dependencies. However, our sending logic processing rate is still 10.4%~11.5% better than commodity Mellanox RDMA NIC (ConnectX-3 Pro) [37, 38]. When message size is larger, *i.e.* >512B, the sender logic can sustain the line-rate of 40Gbps. The prototype also achieves low latency. Specifically, the sending and receiving latency is only 0.54μs and 0.81μs for 64B messages respectively.

Table 1: MP-RDMA States

Functionality	Variable	Size (B)
Congestion control	$cwnd$	4
	$inflate$	4
	snd_una	3
	snd_nxt	3
	rcv_nxt	3
OOO-aware path selection	snd_ooh	3
	L	1
Loss recovery	snd_retx	3
	$recovery$	3
Path probing	$MaxPathID$	2
	p	1
Tracking OOO packets	$bitmap$ data	16
	$bitmap$ head	1
Burst Control	$burst_timer$	3
Connection restart	$restart_timer$	3
Synchronise message	α	1
RTT measurement	$srtt$, $rttvar$, rtt_seq	12
Total	N/A	66

5 Evaluation

In this section, we first evaluate MP-RDMA's overall performance. Then, we evaluate properties of MP-RDMA algorithm using a series of targeted experiments.

Testbed setup. Our testbed consists of 10 servers located under two ToR switches as shown in Fig.10. Each server is a Dell PowerEdge R730 with two 16-core Intel Xeon E5-2698 2.3GHz CPUs and 256GB RAM. Every server has one Mellanox ConnectX-3 Pro 40G NIC as well as an FPGA board that implements MP-RDMA. There are four switches connecting the two ToR switches forming four equal-cost cross-ToR paths. All the switches are Arista DCS-7060CX-32S-F with Trident chip platform. The base cross-ToR RTT is 12μs (measured using RDMA ping). This means the bandwidth delay product for a cross-ToR network path is around 60KB. We enable PFC and configure RED with $(P_{max}, K_{min}, K_{max}) = (1.0, 20KB, 20KB)$ as it provides good performance on our testbed. The initial window is set to be one BDP. We set $\Delta = 32$ and the size of the bitmap $L = 64$.

5.1 Benefits of MP-RDMA

5.1.1 Robust to path failure

1) Lossy paths. We show that MP-RDMA can greatly improve RDMA throughput in a lossy network [27].

Setup: We start one RDMA connection from T0 to T1, continuously sending data at full speed. Then, we manually generate random drop on Path 1, 2 and 3. We leverage the switch built-in iCAP (ingress Content-

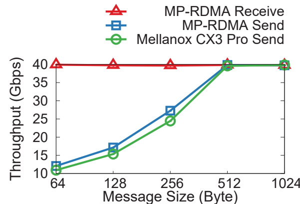


Figure 9: Prototype ability.

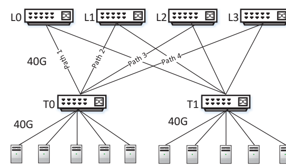
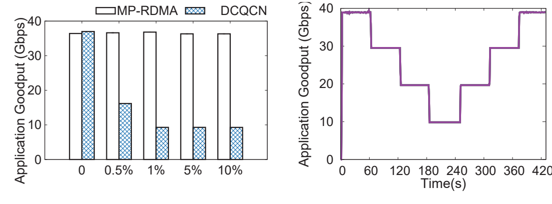


Figure 10: Testbed Topology.



(a) Adaptive to random drop. (b) Reaction to paths failure.

Figure 11: MP-RDMA robustness.

Aware Processor) [2] functionality to drop packets with certain IP ID (*e.g.*, ID *mod* 100 == 0). We compare the goodput between MP-RDMA and single-path RDMA (DCQCN). Each result is the average of 100 runs.

Results: Fig.11(a) illustrates that MP-RDMA always achieves near to optimal goodput (∼38Gbps excluding header overhead) because it always avoids using lossy path. Specifically, the credits on lossy paths are gradually reduced and MP-RDMA moves its load to Path 4 (good path). However, DCQCN has 75% probability to transmit data on lossy paths. When this happens, DCQCN's throughput drops dramatically due to its go-back-N loss recovery mechanism. Specifically, the throughput of the flow traversing lossy path drops to ∼10Gbps when the loss rate is 0.5%, and drops to near zero when loss rate exceeds 1%. This conforms with the results in [36, 49]. As a result, DCQCN can achieve only ∼17.5Gbps average goodput when loss rate is 0.5%. When the loss rate exceeds 0.5%, DCQCN achieves only ∼25% average goodput compared with MP-RDMA. Improving the loss recovery mechanism (*e.g.*, [36]) is a promising direction to further improve the performance of MP-RDMA and DCQCN, but it is not the focus of this paper.

2) Quick reaction to link up and down. We show that MP-RDMA can quickly react to path failure and restore the throughput when failed paths come back.

Setup: We start one MP-RDMA connection from T0 to T1 and configure each path to be 10Gbps. At time 60s, 120s, and 180s, P1, P2, and P3 are disconnected one by one. At time 250s, 310s, and 370s, these paths are restored to healthy status one by one.

Results: Fig.11(b) shows that, upon each path failure, MP-RDMA quickly throttles the traffic on that path, meanwhile fully utilizes other healthy paths. This is because there are no ACKs returning from the failed paths which leads to zero traffic on those paths. While the ACK clocking for healthy paths is not impacted, those paths are fully utilized and are used to recover the lost packets on failed paths. When paths are restored, MP-RDMA can quickly fully utilize the newly recovered path. Specifically, for each restored path, it takes only less than 1s for this path to be fully utilized again. This is benefited from the path probing mechanism of MP-RDMA, which periodically explores new VPs and restores the ACK-clocking on those paths.

5.1.2 Improved Overall Performance

Now, we show that with multi-path enabled, the overall performance can be largely improved by MP-RDMA.

1) Small-Scale Testbed. Now we evaluate the throughput performance on our testbed.

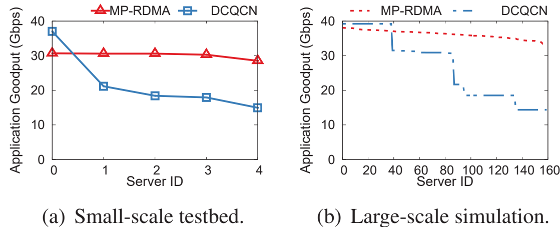
Setup: We generate a permutation traffic [15, 41], where 5 servers in T0 setup MP-RDMA connects to 5 different servers in T1 respectively. Permutation traffic is a common traffic pattern in datacenters [26, 49] and in the following, we use this pattern to study the thoughput, latency and out-of-order behavior of MP-RDMA. We compare the overall goodput (average of 10 runs) of all these 5 connections of MP-RDMA with DCQCN.

Results: The results show that MP-RDMA can well utilize the link bandwidth, achieving in total 150.68Gbps goodput (near optimal excluding header overhead). Due to the coarse-grained per-connection ECMP-based load balance, DCQCN only achieves in total 102.46Gbps. MP-RDMA gains 47.05% higher application goodput than DCQCN. Fig.12(a) shows the goodput of each RDMA connection (denoted by its originated server ID) in one typical run. The 5 flows in MP-RDMA fairly share all the network bandwidth and each achieves ∼30Gbps. However, in DCQCN, only 3 out of 4 paths are used for transmission while the other one path is idle, which leads to much lower (<20Gbps) and imbalanced throughput.

2) Large-Scale Simulation on Throughput. Now we evaluate throughput performance at scale with NS3 [3].

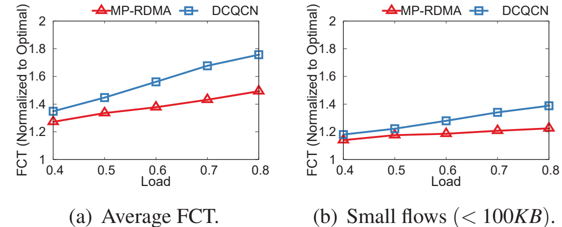
Setup: We build a leaf-spine topology with 4 spine switches, 32 leaf switches and 320 servers (10 under each leaf). The server access link is 40Gbps and the link between leaf and spine is 100Gbps, which forms a full-bisection network. The base RTT is 16us. For the single-path RDMA (DCQCN), we use the simulation code and parameter settings provided by the authors. We use the same permutation traffic [15, 41] as before. Half of the servers act as senders and each sends RDMA traffic to one of the other half servers across different leaf switches. In total there are 160 RDMA connections. For MP-RDMA, the ECN threshold is set to be 60KB.

Results: Fig.12(b) shows the goodput of each RDMA connection. MP-RDMA achieves much better overall performance than DCQCN with ECMP. To be specific, the average throughput of all servers of MP-RDMA is 34.78% better than DCQCN. Moreover, the performance across multiple servers is more even in MP-RDMA,



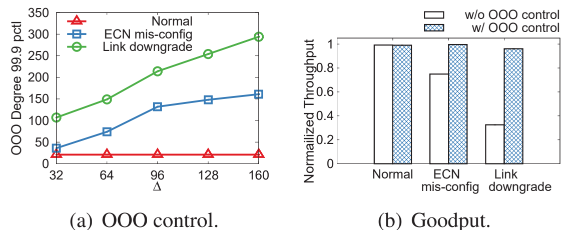
(a) Small-scale testbed. (b) Large-scale simulation.

Figure 12: Overall throughput compared with DCQCN.



(a) Average FCT. (b) Small flows ($< 100KB$).

Figure 13: FCT performance compared with DCQCN.



(a) OOO control. (b) Goodput.

Figure 14: Out-of-order control algorithm performance.

where the lowest connection throughput can still achieve 32.95Gbps. However, in DCQCN, many unlucky flows are congested into a single path, leading to a very low throughput (*e.g.*, <15Gbps) for them.

3) Large-Scale Simulation on FCT.

Setup: We use the same leaf-spine topology and generate flow size according to a web search workload [10]. The source and destination of each flow are randomly picked from all the servers. We further assume that flows arrive according to a Poisson process and vary the inter-arrival time of flows to form different levels of load.

Results: In this experiment, at start up, each connection uses 54 virtual paths. As time goes by, a long flow will result in using about 60∼70 virtual paths. Fig. 13 shows the normalized FCT performance. For average FCT, MP-RDMA is 6.0%∼17.7% better than DCQCN. For large flows (>10MB), throughput is the dominate factor. As MP-RDMA avoids hash collision, they achieve 16.7%∼77.7% shorter FCT than DCQCN. We omit the figure due to space limitation. For small flows (<100KB), MP-RDMA also achieves a little bit better FCT (3.6%∼13.3% shorter) than DCQCN (Fig. 12(b)). This advantage is from finer grained load balance and accurate queue length control of congestion control (§3.2.1) in MP-RDMA.

5.2 MP-RDMA deep-dive

5.2.1 OOO-aware path selection

Now, we show MP-RDMA's OOO-aware path selection algorithm can well control the OOO degree, and achieve good application throughput.

Setup: We use the same traffic as in §5.1.2, and measure the OOO degrees in three different scenarios: 1) *Normal*, in which all paths RED marking parameters are configured as $(P_{max}, K_{min}, K_{max}) = (1.0, 20KB, 20KB)$; 2) *ECN mis-config*, in which the RED of path 4 is misconfigured as $(P_{max}, K_{min}, K_{max}) = (1.0, 240KB, 240KB)$; 3) *link-degradation*, in which path 4 degrades from 40Gbps to 1Gbps due to failure caused auto-negotiation.

Results: First we set bitmap length L to infinite to cover the maximum OOD. Then, we evaluate how MP-RDMA can control the OOD to different extent with different Δ. Fig. 14(a) shows the 99.9th percentile of OOD using different Δ under various scenarios. OOO-aware path selection can well control the OOD. Specifically, compared to MP-RDMA without OOO control, [6] Δ=32 can effectively reduce the OOD 99.9th by ∼5x and ∼50x under *ECN mis-configuration* and *link-degradation* respectively. A proper Δ can control the OOD to a small range, which means that we can use a very small L in practice under various network conditions.

Next, we consider a bitmap with $L = 64$. We set Δ=32 correspondingly. Fig. 14(b) shows the throughput normalized to the ideal case when all connections fairly share the full bandwidth. With OOO control, in *ECN mis-config* case, MP-RDMA achieves optimal throughput. Even in more extreme *link-degradation* case, the overall application throughput is only 3.94% less than the optimal. However, if MP-RDMA uses the same L=64 bitmap but without OOO control, its throughput significantly degrades by 25.1% and 67.5% under these two cases respectively, due to severe OOO.

5.2.2 Congestion-aware Path Selection

Now, we show MP-RDMA's ability to do congestion-aware traffic distribution.

Setup: We configure each path to 10G and start one MP-RDMA long connection sending unlimited data at the maximum rate. Normally, the traffic is evenly balanced among the four parallel paths. Then after ∼30s, we start another special MP-RDMA flow which is manually forced to use only Path 4 (denoted as SP-RDMA). The SP-RDMA flow will cause a sudden congestion on Path 4. We evaluate how MP-RDMA perceives the congestion and moves the load away from Path 4.

Results: Fig. 15 shows the throughput of the MP-RDMA flows on each of the four paths. Before the SP-RDMA flow joins, each path has a throughput stable at ∼10Gbps. After the SP-RDMA joins on Path 4, the throughput of the MP-RDMA flows on Path 4 quickly falls to near zero. Meanwhile, the throughput on other

[6] Without OOO control, the 99.9th OOD is 179 and 5324 for the two abnormal scenarios, respectively.

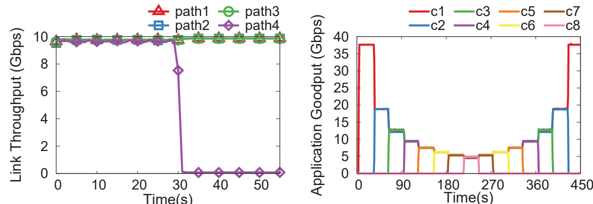


Figure 15: Path selection.

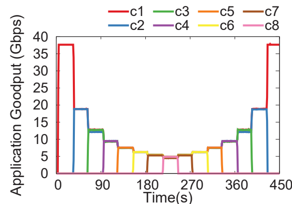


Figure 16: MP-RDMA Fairness.

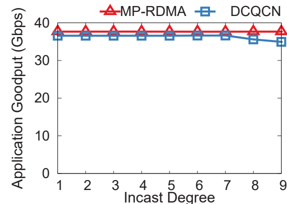


Figure 17: Incast performance.

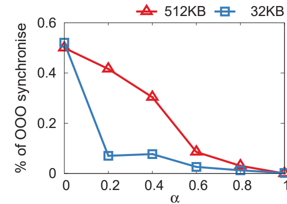


Figure 18: α impact.

3 paths all remains at around 10Gbps. This indicates that MP-RDMA can quickly perceive the congestion on Path 4, and moves the load away from this path. Also, since the congestion conditions on other paths remain unchanged, MP-RDMA does not adjust the load on them. Here we don't focus on the fairness between SP-RDMA and MP-RDMA connections.

5.2.3 Fairness of MP-RDMA

Setup: In this experiment, two physical servers under one ToR establish multiple MP-RDMA connections to another server under the same ToR creating a single bottleneck. 8 MP-RDMA connections are started one by one with an interval of 30s, and then leaves the network one after another with the same time interval. We measure the application goodput of each connection.

Results: Fig. 16 shows that all flows evenly share the network, and get the fair share quickly. Specifically, each connection's throughput quickly converges to $\sim\frac{40}{n}$Gbps, when n varies from 1 to 8 and then 8 to 1. The Jain fairness index [31] is within 0.996 - 0.999 (1 is optimal) under various number of concurrent flows.

5.2.4 Incast

Next we evaluate MP-RDMA's congestion control under more stressed scenario, *i.e.*, incast.

Setup: The traffic pattern mimics a disk recovery service [49] where failed disks are repaired by fetching backups from several other servers. Specifically, a receiver host initiates one connection with each of the N randomly selected sender host, simultaneously requesting 1Gb data from each sender. Following the convention in DCQCN [49], we vary the incast degree from 1 to 9. The experiment is repeated five times. We evaluate the overall application goodput at the receiver end.

Results: Fig. 17 shows that MP-RDMA achieves similar near-optimal incast performance as DCQCN. To be specific, when incast degree increases from 1 to 9, the total goodput of the 5 connections remains stable, at around 37.65Gbps. Note that MP-RDMA achieves a little (∼3%) higher goodput than DCQCN. We cannot ascertain the exact root cause of this, but we believe this may be an implementation issue with the Mellanox NIC instead of an algorithm issue with DCQCN.

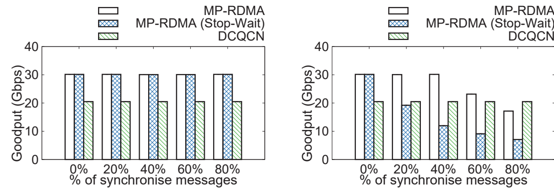
5.2.5 Synchronise Mechanism Performance

In this section, we evaluate the impact of *sychronise* mechanism on application performance.

Setup: The same permutation traffic in §5.1.2 is used to emulate a typical network congestion. *Synchronise* messages will be delayed for a while and then send out. This results in burst traffic and causes large delay fluctuations. We stress test the mechanism under the case when the load is as high as 0.8. We first study the setting of parameter α by measuring the amount of out-of-order *synchronise* messages under different α. Then α is set to a value that ensures all the *synchronise* messages are in order. The average goodput for the 5 connections under various ratio of synchronise messages are measured. Two different message sizes are evaluated, *i.e.*, 512KB (RDMA-based RPC application [48]) and 32KB (a more challenging smaller size). The results are compared with DCQCN, which achieves only ∼20Gbps in average (due to ECMP hash collision). We also evaluate MP-RDMA (Stop-Wait), in which a synchronise message is sent only when all previous messages are completed.

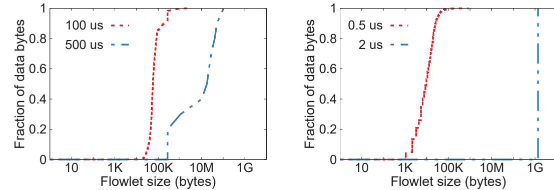
Results: As shown in Fig 18, larger α leads to less OOO *synchronise* messages. Under the same α, OOO is severer for larger message size due to the more congested network. When α is 1.0, no OOO occurs in our tests. As such, we set α to 1.0 for the following experiment.

Fig. 19 shows the result for *synchronise* mechanism impact on throughput. When message size is large (*e.g.*, 512KB), both MP-RDMA and MP-RDMA (Stop-Wait) can achieve ∼30Gbps goodput, which is ∼48% higher than single-path DCQCN across all *synchronise* ratios. This is because the Δt for sending *synchronise* messages is ∼0.5 RTT for MP-RDMA and ∼1 RTT for MP-RDMA (Stop-Wait). Both are rather small compared with the transmission delay for a 512KB message. Thus the impact of Δt is amortized. When message size is smaller (*i.e.*, 32KB), Δt is larger compared with the message transmission delay. Thus the goodput drops as the *synchronise* message ratio grows. However, with our optimistic transmission algorithm (§3.4), MP-RDMA still achieves good performance. Specifically, MP-RDMA gets 13%∼49% higher throughput than DCQCN under 0∼60% *synchronise* ratio. When the synchronise ratio grows to 80%, MP-RDMA performs 16.28% worse. Note that this is already the worst case performance for MP-RDMA because the traffic load is at its peak, *i.e.* 100%. More results (omitted due to space limitation) show that, when the load is lighter, MP-RDMA performs very close to DCQCN under high *synchronise* ra-



(a) 512KB message. (b) 32KB message.

Figure 19: Synchronise mechanism performance.



(a) TCP. (b) RDMA.

Figure 20: Flowlet characteristics in TCP and RDMA.

tio. On the contrary, the naive MP-RDMA (Stop-Wait) only achieves less than 50% throughput of MP-RDMA.

6 Related Work

Various multi-path transmission mechanisms propose to utilize parallel network paths in datacenters [8, 9, 14, 15, 21, 28, 34, 40–42]. Most of them consider only TCP, and cannot be directly used for RDMA.

Load-balance routing: Previous approaches such as [8, 9, 14, 15, 21, 28, 34, 40, 42] propose to balance traffic over a set of paths at the routing layer. In order to handle out-of-order packets, some of them, *e.g.*, [15, 28], utilize a dedicated reordering buffer under the transport layer. However, these schemes are hard to implement in NIC hardware. Other work, *e.g.*, [9, 34], try to proactively avoid out-of-order delivery. Most of them utilize *flowlets*. If the inactive gap between flowlets is long enough, flowlets can be distributed to different paths without causing out-of-order. However, for RDMA which is implemented in hardware and usually smoothed with a rate-shaper, it is quite hard to find flowlets. To validate this, we study the flowlet characteristics of RDMA and compare it with TCP on our testbed. We measure the size of flowlets with various inactive intervals. For each experiment, we run 8 RDMA/TCP flows with size 2GB. Fig. 20 shows that it is really difficult to observe flowlets in RDMA traffic. When the inactive interval is larger than $2\mu s$, the flowlet size is strictly 2GB. In contrast, TCP does have flowlets. When we set the inactive gap to $100\mu s$, we observe many flowlets with size $\sim$60KB. We conclude that flowlet-based load balancing schemes may not work well for RDMA traffic. A recent work [47] reports that flowlet can be used to do load balance for DCQCN traffic. This might be true for applications with an on-off traffic pattern, but not for applications that are throughput intensive. Moreover, as flowlets cannot guarantee out-of-order free, it's not clear how out-of-order RDMA packets would impact the performance in [47].

Multi-path transport: MPTCP modifies TCP to enable multi-path transmission [16, 17, 41]. The core idea is to split the original transport into several sub-flows and distribute packets among these sub-flows according to their congestion states. Thereby MPTCP adds additional states proportional to the number of sub-flows and explores a large re-ordering buffer at the transport layer to handle out-of-order packets. As aforementioned, this design adds considerable memory overhead and is difficult to implement in hardware.

Generally, the multi-path ACK-clocking of MP-RDMA resembles the PSLB algorithm [13] in the sense that both schemes adjust their load on multiple paths in a per-ACK fashion. However, MP-RDMA independently adjusts the load on each path while PSLB dynamically moves the load of slow paths to fast paths.

Recently Mellanox proposed a multi-path support for RDMA [46]. However, it is just a fail-over solution using two existing single-path RoCE connections (hot standby). The goal and the techniques of the Mellanox multi-path RDMA are completely different from MP-RDMA, which is a new multi-path RDMA transport.

7 Conclusion

This paper presents MP-RDMA, a multi-path transport for RDMA in datacenters. It can efficiently utilize the rich network paths in datacenters while keeping on-chip memory footprint low. MP-RDMA employs novel *multi-path ACK-clocking* and *out-of-order aware path selection* to choose best network paths and distribute packets among them in a congestion-aware manner. In total, MP-RDMA requires only a small constant (66B) amount of extra memory for each RDMA connection no matter how many network paths are used. Our FPGA-based prototype validates the feasibility for MP-RDMA's hardware implementation. Our evaluations on a small-scale testbed as well as large-scale simulation illustrate the effectiveness for MP-RDMA in utilizing the rich network paths diversity in datacenters.

Acknowledgements

We thank our shepherd Michael Freedman and the anonymous reviewers for their valuable comments and suggestions. We are grateful to Wei Bai, Ran Shu and Henry Xu for their comments on the modeling. We thank Larry Luo for his discussion to improve the quality of the paper. Yun Wang, Wencong Xiao and Sukhan Lee helped us to improve the writing. This research was partially funded by the National Natural Science Foundation of China (Grants No. U1605251).

References

[1] 802.1Qbb - Priority-based Flow Control. *http://www.ieee802.org/1/pages/802.1bb.html*.

[2] Arista 7050x and 7050x2 switch architecture (a day in the life of a packet). https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7050X_Switch_Architecture.pdf.

[3] Ns3: A discrete-event network simulator for internet systems. https://www.nsnam.org/.

[4] *InfiniBand architecture volume 1, general specifications, release 1.2.1*. InfiniBand Trade Association, 2008.

[5] *InfiniBand architecture volume 2, physical specifications, release 1.3*. InfiniBand Trade Association, 2012.

[6] *Supplement to InfiniBand architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (IP routable RoCE)*. InfiniBand Trade Association, 2012.

[7] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[9] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 503–514, New York, NY, USA, 2014. ACM.

[10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[11] M. Alizadeh, A. Javanmard, and B. Prabhakar. Analysis of dctcp: stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 73–84. ACM, 2011.

[12] Altera. Stratix v fpgas. https://www.altera.com/products/fpga/stratix-series/stratix-v/overview.html.

[13] J. Anselmi and N. Walton. Decentralized proportional load balancing. *SIAM Journal on Applied Mathematics*, 76(1):391–410, 2016.

[14] T. Benson, A. Anand, A. Akella, and M. Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 8. ACM, 2011.

[15] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz. Per-packet load-balanced, low-latency routing for clos-based data center networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 49–60, New York, NY, USA, 2013. ACM.

[16] Y. Cao, M. Xu, and X. Fu. Delay-based congestion control for multipath tcp. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2012.

[17] Y. Cao, M. Xu, X. Fu, and E. Dong. Explicit multipath congestion control for data center networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 73–84. ACM, 2013.

[18] G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. L. Luo, Y. Xiong, X. Wang, et al. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016.

[19] Cisco. Priority flow control: Build reliable layer 2 infrastructure. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-542809_ns783_Networking_Solutions_White_Paper.html.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[21] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM 2013*.

[22] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[23] S. Floyd, A. Gurtov, and T. Henderson. The newreno modification to tcp's fast recovery algorithm. 2004.

[24] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (ToN)*, 1(4):397–413, 1993.

[25] 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 '09*, 2009.

[26] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 202–215. ACM, 2016.

[27] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. volume 45, pages 139–152. ACM, 2015.

[28] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. Presto: Edge-based load balancing for fast datacenter networks. *ACM SIGCOMM Computer Communication Review*, 45(4):465–478, 2015.

[29] A. Hughes and J. Touch. J. heidemann," issues in tcp slow-start restart after idle. *Work in Progress*, 1, 1998.

[30] IEEE. Ieee 802.1qau - congestion notification. http://www.ieee802.org/1/pages/802.1au.html.

[31] R. Jain, A. Durresi, and G. Babic. Throughput fairness index: An explanation. Technical report, Tech. rep., Department of CIS, The Ohio State University, 1999.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[33] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, GA, 2016. USENIX Association.

[34] N. Katta, M. Hira, A. Ghag, C. Kim, I. Keslassy, and J. Rexford. Clove: How i learned to stop worrying about the core and love the edge. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 155–161. ACM, 2016.

[35] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, and P. Cheng. Clicknp: Highly flexible and high-performance network processing with reconfigurable hardware. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 1–14. ACM, 2016.

[36] Y. Lu, G. Chen, Z. Ruan, W. Xiao, B. Li, J. Zhang, Y. Xiong, P. Cheng, and E. Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking*, pages 22–28. ACM, 2017.

[37] Mellanox. Connectx-3 pro en single/dual-port adapters 10/40/56gbe adapters w/ pci express 3.0. http://www.mellanox.com/page/products_dyn?product_family=162&mtag=connectx_3_pro_en_card.

[38] Mellanox. Roce vs. iwarp competitive analysis. http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf.

[39] R. Mittal, N. Dukkipati, E. Blem, H. M. G. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. Timely: Rtt-based congestion control for the datacenter. *acm special interest group on data communication*, 45(4):537–550, 2015.

[40] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 307–318. ACM, 2014.

[41] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

[42] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIGCOMM Computer Communication Review*, 44(4):407–418, 2015.

[43] L. C. Reference. Linux/include/linux/mlx4/qp.h. http://lxr.free-electrons.com/source/include/linux/mlx4/qp.h.

[44] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.

[45] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM Computer Communication Review*, 45(4):183–197, 2015.

[46] M. Technologies. Multi-path rdma. https://www.openfabrics.org/images/eventpresos/workshops2015/DevWorkshop/Tuesday/tuesday_04.pdf.

[47] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI*, pages 407–420, 2017.

[48] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. G ra m: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 408–421. ACM, 2015.

[49] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 523–536. ACM, 2015.

Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization

Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat

Google, Inc.

Abstract

This paper presents our design and experience with Andromeda, Google Cloud Platform's network virtualization stack. Our production deployment poses several challenging requirements, including performance isolation among customer virtual networks, scalability, rapid provisioning of large numbers of virtual hosts, bandwidth and latency largely indistinguishable from the underlying hardware, and high feature velocity combined with high availability.

Andromeda is designed around a flexible hierarchy of flow processing paths. Flows are mapped to a programming path dynamically based on feature and performance requirements. We introduce the *Hoverboard* programming model, which uses gateways for the long tail of low bandwidth flows, and enables the control plane to program network connectivity for tens of thousands of VMs in seconds. The on-host dataplane is based around a high-performance OS bypass software packet processing path. CPU-intensive per packet operations with higher latency targets are executed on coprocessor threads. This architecture allows Andromeda to decouple feature growth from fast path performance, as many features can be implemented solely on the coprocessor path. We demonstrate that the Andromeda datapath achieves performance that is competitive with hardware while maintaining the flexibility and velocity of a software-based architecture.

1 Introduction

The rise of Cloud Computing presents new opportunities and challenges for networking. Cloud providers must support virtual networks with high performance and a rich set of features such as load balancing, firewall, VPN, QoS, DoS protection, isolation, and NAT, all while operating at a global scale. There has been substantial research in network support for Cloud Computing, in particular in high-speed dataplanes [17, 34], virtualized routing infrastructure [6, 22, 23, 31], and NFV middleboxes [14, 16]. Typical research efforts focus on point problems in the space, rather than the challenges of bringing a working system together end to end. We developed *Andromeda*, the network virtualization environment for Google Cloud Platform (GCP). We use this experience to show how we divide functionality across a global, hierarchical control plane, a high-speed on-host virtual switch, packet processors, and extensible gateways.

This paper focuses on the following topics:

- The Andromeda Control plane is designed for agility, availability, isolation, and scalability. Scale up and down of compute and rapid provisioning of virtual infrastructure means that the control plane must achieve high performance and availability. Andromeda scales to networks over 100,000 VMs, and processes network updates with a median latency of 184ms. Operations on behalf of one virtual network, e.g., spinning up 10k VMs, should not impact responsiveness for other networks.
- The Andromeda Dataplane is composed of a flexible set of flow processing paths. The *Hoverboard path* enables control plane scaling by processing the long tail of mostly idle flows on dedicated gateways. Active flows are processed by our on-host dataplane. The on-host *Fast Path* is used for performance-critical flows and currently has a 300ns per-packet CPU budget. Expensive per-packet work on-host is performed on the *Coprocessor Path*. We found that most middlebox functionality such as stateful firewalls can be implemented in the on-host dataplane. This improves latency and avoids the high cost of provisioning middleboxes for active flows.
- To remain at the cutting edge, we constantly deploy new features, new hardware, and performance improvements. To maintain high deployment velocity without sacrificing availability, Andromeda supports transparent VM live migration and non-disruptive dataplane upgrades.

We describe the design of Andromeda and our experience evolving it over five years. Taken together, we have improved throughput by 19x, CPU efficiency by 16x, latency by 7x, and maximum network size by 50x, relative to our own initial production deployment. Andromeda

also improved velocity through transparent VM migration and weekly non-disruptive dataplane upgrades, all while delivering a range of new end-customer Cloud features.

2 Overview

2.1 Requirements and Design Goals

A robust network virtualization environment must support a number of baseline and advanced features. Before giving an overview of our approach, we start with a list of features and requirements that informed our thinking:

• At the most basic level, network virtualization requires supporting isolated virtual networks for individual customers with the illusion that VMs in the virtual network are running on their own private IP network. VMs in one virtual network should be able to communicate with one another, to internal Cloud provider services, to third party providers, and to the Internet, all subject to customer policy while isolated from actions in other virtual networks. While an ideal, our target is to support the same throughput and latency available from the underlying hardware.

• Beyond basic connectivity, we must support constantly evolving network features. Examples include billing, DoS protection, tracing, performance monitoring, and firewalls. We added and evolved these features and navigated several major architectural shifts, such as transitioning to a kernel bypass dataplane, all without VM disruption.

• A promise of Cloud Computing is higher availability than what can be provisioned in smaller-scale deployments. Our network provides global connectivity, and it is a core dependency for many services; therefore, it must be carefully designed to localize failures and meet stringent availability targets.

• Operationally, we have found live virtual machine migration [5, 11, 18, 30] to be a requirement for both overall availability and for feature velocity of our infrastructure. Live migration has a number of stringent requirements, including packet delivery during the move from one physical server to another, as well as minimizing the duration of any performance degradation.

• Use of GCP is growing rapidly, both in the number of virtual networks and the number of VMs per network. One critical consideration is *control plane scalability*. Large networks pose three challenges, relative to small networks: they require larger routing tables, the routing tables must be disseminated more broadly, and they tend to have higher rates of churn. The control plane must be able to support networks with tens or even hundreds of thousands of VMs. Additionally, low programming latency is important for autoscaling and failover. Furthermore, the ability to provision large networks quickly makes it possible to run large-scale workloads such as MapReduce inexpensively, quickly, and on demand.

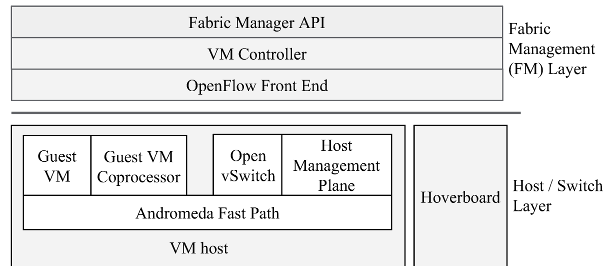


Figure 1: Andromeda Stack

2.2 Design Overview

Our general design approach centers around hierarchical data and control planes. The control plane is designed around a global hierarchy coupled with the overall Cloud cluster management layer. For example, configuring Andromeda is only one step among many in configuring compute, storage, access control, etc. For isolation, we run separate control stacks in every *cluster*. A cluster is a collection of colocated machines with uniform network connectivity that share the same hardware failure domain. The Andromeda control plane maintains information about where every VM in the network currently runs, and all higher-level product and infrastructure state such as firewalls, load balancers, and routing policy. The control plane installs selected subsets of this state in individual servers through a hierarchy of controllers.

The dataplane consists of a set of flexible user-space packet processing paths. The VM host *Fast Path* is the first path in the dataplane hierarchy and targets raw performance over flexibility. The Fast Path has a per-packet CPU budget of 300ns. Achieving this goal requires limiting both the complexity of Fast Path work and the amount of Fast Path state required to process packets. High performance, latency-critical flows are processed end to end on the Fast Path. Andromeda forwards other flows from the Fast Path to *Hoverboards* or *Coprocessors* for additional processing. On-host software Coprocessors running in per-VM floating threads perform per-packet work that is CPU-intensive or without strict latency targets. Coprocessors decouple feature growth from Fast Path performance, providing isolation, ease of programming, and scalability.

Andromeda sends packets that do not match a flow rule on the VM host to Hoverboards, dedicated gateways that perform virtual network routing. The control plane dynamically selects only active flows to be installed on VM hosts based on current communication patterns. Hoverboards process the long tail of mostly idle flows. Since typically only a small subset of possible VM pairs in a network communicate, only a small fraction of network configuration is required at an individual VM host. Avoiding the need to install full forwarding information on every host improves per-server memory utilization and control-plane CPU scalability by over an order of magnitude.

3 Control Plane

The Andromeda control plane consists of three layers:

Cluster Management (CM) Layer: The CM layer provisions networking, storage, and compute resources on behalf of users. This layer is not networking-specific, and is beyond the scope of this paper.

Fabric Management (FM) Layer: The FM layer exposes a high-level API for the CM Layer to configure virtual networks. The API expresses user intent and abstracts implementation details, such as the mechanism for programming switches, the encapsulation format, and the network elements responsible for specific functions.

Switch Layer: In this layer, two types of software switches support primitives such as encapsulation, forwarding, firewall, and load balancing. Each VM host has a virtual switch based on Open vSwitch [33], which handles traffic for all VMs on the host. Hoverboards are standalone switches, which act as default routers for some flows.

3.1 FM Layer

When the CM layer connects to a FM controller, it sends a `full` update containing the complete FM configuration for the cluster. Subsequent updates are diffs against previously sent configuration. The FM configuration consists of a set of *entities* with a known type, a unique name, and parameters defining entity properties. Figure 2 lists some examples of FM entities.

The FM API is implemented by multiple types of controllers, each responsible for different sets of network devices. Presently, VM Controllers (VMCs) program VM hosts and Hoverboards, while Load-Balancing Controllers [12] program load balancers. This paper focuses on VMCs.

VMCs program VM host switches using a combination of OpenFlow [3, 28] and proprietary extensions. VMCs send OpenFlow requests to proxies called *OpenFlow Front Ends (OFEs)* via RPC – an architecture inspired by Onix [25]. OFEs translate those requests to OpenFlow. OFEs decouple the controller architecture from the OpenFlow protocol. Since OFEs maintain little internal state, they also serve as a stable control point for VM host switches. Each switch has a stable OFE connection without regard for controller upgrade or repartitioning.

OFEs send switch events to VMCs, such as when a switch connects to it, or when virtual ports are added for new VMs. VMCs generate OpenFlow programming for switches by synthesizing the abstract FM programming and physical information reported in switch events. When a VMC is notified that a switch connected, it reconciles the switch's OpenFlow state by reading the switch's state via the OFE, comparing that to the state expected by the VMC, and issuing update operations to resolve any differences.

Network: QoS, firewall rules, ...
VM: Private IP, external IPs, tags, ...
Subnetwork: IP prefix
Route: IP prefix, priority, next hop, ...

Figure 2: Examples of FM Entities

Multiple VMC partitions are deployed in every cluster. Each partition is responsible for a fraction of the cluster hosts, determined by consistent hashing [20]. The OFEs broadcast some events, such as switch-connected events, to all VMC partitions. The VMC partition responsible for the host switch that generated the event will then subscribe to other events from that host.

3.2 Switch Layer

The switch layer has a programmable software switch on each VM host, as well as software switches called Hoverboards, which run on dedicated machines. Hoverboards and host switches run a user-space dataplane and share a common framework for constructing high-performance packet processors. These dataplanes bypass the host kernel network stack, and achieve high performance through a variety of techniques. Section 4 discusses the VM host dataplane architecture.

We employ a modified Open vSwitch [33] for the control portion of Andromeda's VM host switches. A user-space process called *vswitchd* receives OpenFlow programming from the OFE, and programs the datapath. The dataplane contains a flow cache, and sends packets that miss in the cache to vswitchd. vswitchd looks up the flow in its OpenFlow tables and inserts a cache entry.

We have modified the switch in a number of substantial ways. We added a C++ wrapper to the C-based vswitchd, to include a configuration mechanism, debugging hooks, and remote health checks. A management plane process called the *host agent* supports VM lifecycle events, such as creation, deletion, and migration. For example, when a VM is created, the host agent connects it to the switch by configuring a virtual port in Open vSwitch for each VM network interface. The host agent also updates VM to virtual port mapping in the FM.

Extension modules add functionality not readily expressed in OpenFlow. Such extensions include connection tracking firewall, billing, sticky load balancing, security token validation, and WAN bandwidth enforcement [26]. The extension framework consists of upcall handlers run during flow lookup. For example, *Pre-lookup handlers* manage flow cache misses prior to OpenFlow lookup. One such handler validates security tokens, which are cryptographic ids inserted into packet headers to prevent spoofing in the fabric. Another type is *group lookup handlers*, which override the behavior of specific OpenFlow groups, e.g., to provide sticky load balancing.

3.3 Scalable Network Programming

A key question we faced in the design and evolution of Andromeda was how to maintain correct forwarding behavior for individual virtual networks that could in theory scale to millions of individual VMs. Traditional networks heavily leverage address aggregation and physical locality to scale the programming of forwarding behavior. Andromeda, in contrast, decouples virtual and physical addresses [23]. This provides many benefits, including flexible addressing for virtual networks, and the ability to transparently migrate VMs within the physical infrastructure (Section 3.4). However, this flexibility comes at a cost, especially with respect to scaling the control plane.

One of the following three models is typically used to program software-defined networks:

Preprogrammed Model: The control plane programs a full mesh of forwarding rules from each VM to every other VM in the network. This model provides consistent and predictable performance. However, control plane overhead scales quadratically with network size, and any change in virtual network topology requires a propagation of state to every node in the network.

On Demand Model: The first packet of a flow is sent to the controller, which programs the required forwarding rule. This approach scales better than the preprogrammed model. However, the first packet of a flow has significantly higher latency. Furthermore, this model is very sensitive to control plane outages, and worse, it exposes the control plane to accidental or malicious packet floods from VMs. Rate limiting can mitigate such floods, but doing so while preserving fairness and isolation across tenants is complex.

Gateway Model: VMs send all packets of a specific type (e.g., all packets destined for the Internet) to a gateway device, designed for high speed packet processing. This model provides predictable performance and control plane scalability, since changes in virtual network state need to be communicated to a small number of gateways. The downside is that the number of gateways needs to scale with the usage of the network. Worse, the gateways need to be provisioned for peak bandwidth usage and we have found that peak to average bandwidth demands can vary by up to a factor of 100, making it a challenge to provision gateway capacity efficiently.

3.3.1 Hoverboard Model

Andromeda originally used the preprogrammed model for VM-VM communication, but we found that it was difficult to scale to large networks. Additionally, the preprogrammed model did not support *agility* – the ability to rapidly provision infrastructure – which is a key requirement for on-demand batch computing.

To address these challenges, we introduced the *Hoverboard Model*, which combines the benefits of On-Demand and Gateway models. The Andromeda VM host stack sends all packets for which it does not have a route to Hoverboard gateways, which have forwarding information for all virtual networks. However, unlike the gateway model, the control plane dynamically detects flows that exceed a specified usage threshold and programs *offload flows*, which are direct host-to-host flows that bypass the Hoverboards. Figure 3 shows flows that use Hoverboards as a default router and flows for which the control plane has programmed a direct host to host route and *offloaded* them from the Hoverboard.

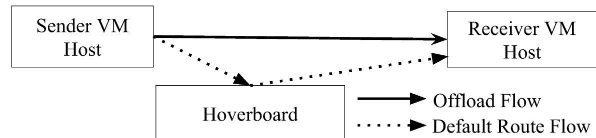


Figure 3: Hoverboard Packet Forwarding

The control plane detects these high bandwidth flows based on usage reports from the sending VM hosts. For robustness, we do not rely on usage reports from the Hoverboards themselves: the Hoverboards may not be able to send such reports if they are overloaded, and thus the control plane would be unable to install offload flows to reduce the load.

The Hoverboard model avoids the pitfalls with the other models. It is scalable and easy to provision: Our evaluation (see Section 5.3) shows that the distribution of flow bandwidth tends to be highly skewed, so a small number of offload flows installed by the control plane diverts the vast majority of the traffic in the cluster away from the Hoverboards. Additionally, unlike the On Demand Model, all packets are handled by a high performance datapath designed for low latency.

Currently we use the Hoverboard model only to make routing scale, but we plan to extend Hoverboards to support load balancing and other middlebox features. Stateful features, such as firewall connection tracking or sticky load balancing, are more challenging to support in the Hoverboard model. Challenges include state loss during Hoverboard upgrade or failure [21], transferring state when offloading, and ensuring that flows are "sticky" to the Hoverboard that has the correct state.

3.4 Transparent VM Live Migration

We opted for a high-performance software-based architecture instead of a hardware-only solution like SR-IOV because software enables flexible, high-velocity feature deployment (Section 4.1). VM Live Migration would be difficult to deploy transparently with SR-IOV as the guest would need to cope with different physical NIC resources on the migration target host.

Live migration [5, 11] makes it possible to move a running VM to a different host to facilitate maintenance, upgrades, and placement optimization. Migrations are

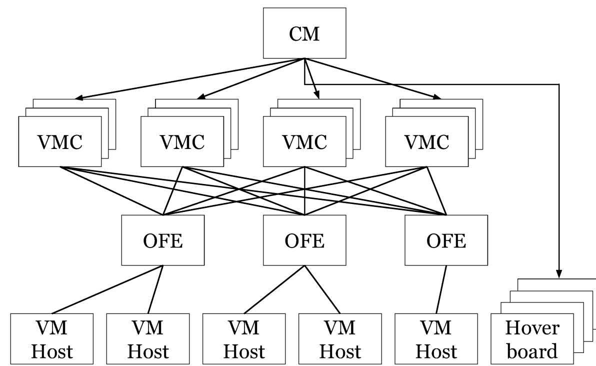


Figure 4: Control Plane Replication and Partitioning

virtually transparent to the VM: the VM continues to see the same virtual Ethernet device, and Andromeda ensures that network connections are not interrupted. The VM is paused during the migration blackout phase, which has a median duration of 65ms and 99th percentile of 388ms. After blackout, the VM resumes execution on the destination host.

Prior work [11, 30] focuses on migrations within a single layer-2 domain. In contrast, Andromeda supports global virtual networks and migrations across clusters. A key challenge is avoiding packet loss during migration even though global routing tables cannot be updated instantly. During blackout, Andromeda enables *hairpin* flows on the migration source host. The migration source will hairpin any ingress packets intended for the migrating VM by forwarding the packets to the migration destination. After blackout ends, other VM hosts and Hoverboards are updated to send packets destined for the migrated VM to the migration destination host directly. Finally, the hairpin flows on the migration source are removed. VROOM [35] uses a similar approach to live migrate virtual routers.

3.5 Reliability

The Andromeda control plane is designed to be highly available. To tolerate machine failures, each VMC partition consists of a Chubby-elected [9] master and two standbys. Figure 4 shows an Andromeda instance for a cluster with four replicated VMC partitions. We found the following principles important in designing a reliable, global network control plane:

Scoped Control Planes: Andromeda programs networks that can be global in scope, so the cluster control plane must receive updates for VMs in all other clusters. We must ensure that a bad update or overload in one region cannot spill over to the control planes for other regions. To address this challenge, we split the control plane into a *regionally aware control plane (RACP)* and a *globally aware control plane (GACP)*. The RACP programs all intra-region network connectivity, with its configuration limited to VMs in the local region. The GACP manages inter-region connectivity, receiving FM updates for both local and remote regions. This approach ensures that intra-region networking in each region is a separate failure domain.

Network Isolation: Churn within one customer's network should not impact network programming latency for other networks. To that end, VMCs maintain separate queues for processing FM updates for each network.

Fail static: Every layer of the control plane is designed to fail static. For example, hosts continue to forward packets using the last-known-good programming state, even if VMCs are unavailable. Hosts checkpoint their state to allow them to preserve the fail static behavior across host process restarts, and we routinely test fail-static behavior by simulating outages in a test environment.

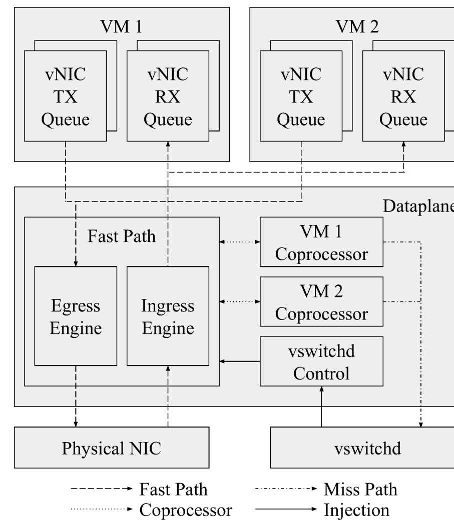


Figure 5: Host Dataplane Overview

4 VM Host Dataplane

Figure 5 illustrates the Andromeda VM host dataplane. The dataplane is a userspace process that performs all on-host VM packet processing, combining both virtual NIC and virtual switch functionality. There are two primary dataplane packet processing paths: the *Fast Path* and the *Coprocessor Path*. The Fast Path performs high-performance packet processing work such as encapsulation and routing via a flow table. The Coprocessor Path performs packet work that is either CPU-intensive or does not have strict latency requirements, such as WAN packet encryption.

Each VM is managed by a userspace Virtual Machine Manager (VMM). There is one VMM per VM. The VMM sends RPCs to the Andromeda dataplane for operations such as mapping guest VM memory, configuring virtual NIC interrupts and offloads, and attaching virtual NIC queues.

The Fast Path maintains a cache of forwarding state and associated packet processing actions. When a packet

misses in the Fast Path cache, it is sent to on-host vswitchd, which maintains the full forwarding state programmed by VMC. Vswitchd sends flow cache update instructions and *reinjects* packets into the Fast Path.

A key goal of the Fast Path is to provide high-throughput, low-latency VM networking. For performance, the Fast Path busy polls on a dedicated logical CPU. The other logical CPU on the physical core runs low-CPU control plane work, such as RPC processing, leaving most of the physical core for Fast Path use. The Fast Path can process over three million small packets per second with a single CPU, corresponding to a per-packet CPU budget of 300ns. The Fast Path can be scaled to multiple CPUs using multi-queue NICs.

4.1 Principles and Practices

Our overall dataplane design philosophy is flexible, high-performance software coupled with hardware offloads. A high performance software dataplane can provide performance indistinguishable from the underlying hardware. We provision sufficient Fast Path CPU to achieve throughput targets, leveraging hardware offloads on a per-platform basis to minimize the Fast Path CPU required. Currently, we offload encryption; checksums; and memory copies using Intel QuickData DMA Engines [2]. We are investigating more substantial hardware offloads.

A software dataplane allows a uniform featureset and performance profile for customers running on heterogeneous hardware with different NICs and hardware offloads. This uniformity enables transparent live migration across heterogenous hardware. Whether a network feature has none, some, or all functionality in software or hardware becomes a per-platform detail. On-host software is extensible, supports rapid release velocity (see Section 6.2), and scales to large amounts of state. A full SR-IOV hardware approach requires dedicated middleboxes to handle new features or to scale beyond hardware table limits. Such middleboxes increase latency, cost, failure rates, and operational overhead.

To achieve high performance in software, the Fast Path design *minimizes Fast Path features*. Each feature we add to the Fast Path has a cost and consumes per-packet CPU budget. Only performance-critical low-latency work belongs on the Fast Path. Work that is CPU-intensive or does not have strict latency requirements, such as work specific to inter-cluster or Internet traffic, belongs on the Coprocessor Path. Our design also *minimizes per-flow work*. All VM packets go through the Fast Path routing flow table. We can optimize by using flow key fields to pre-compute per-flow work. For example, Andromeda computes an efficient per-flow firewall classifier during flow insertion, rather than requiring an expensive full firewall ruleset match for every packet.

The Fast Path uses high-performance best practices: avoid locks and costly synchronization, optimize memory locality, use hugepages, avoid thread handoffs, end-to-end batching, and avoid system calls. For example, the Fast Path only uses system calls for Coprocessor thread wakeups and virtual interrupts. The Fastpath uses lock-free Single Producer / Single Consumer (SPSC) packet rings and channels for communication with control and Coprocessor threads.

4.2 Fast Path Design

The Fast Path performs packet processing actions required for performance-critical VM flows, such as intra-cluster VM-VM. The Fast Path consists of separate ingress and egress engines for packet processing and other periodic work such as polling for commands from control threads. Engines consist of a set of reusable, connected packet processing push or pull *elements* inspired by Click [29]. Elements typically perform a single task and operate on a batch of packets (up to 128 packets on ingress and 32 on egress). Batching results in a 2.4x increase in peak packets per second. VM and Host NIC queues are the sources and sinks of element chains. To avoid thread handoffs and system calls, the Fast Path directly accesses both VM virtual NIC and host physical NIC queues via shared memory, bypassing the VMM and host OS.

Figures 6 and 7 outline the elements and queues for Andromeda Fast Path engines. The pull-pusher is the C++ call point into the element chain in both Fast Path engines. In the egress engine, the pull-pusher pulls packets from the VM transmit queue chain, and pushes the packets to the NIC transmit queue chain. In the ingress engine, the pull-pusher pulls packets from NIC receive queue chain and pushes the packets to the VM chain. Engine element chains must support fan-in and fan-out, as there may be multiple queues and VMs. Hash Demux elements use a packet 5-tuple hash to fan-out a batch of packets across a set of push elements. Schedulers pull batches of packets from a set of pull elements to provide fan-in. To scale to many VMs and queues per host, the VM Round Robin Scheduler element in the egress engine checks if a VM's queues are idle before calling the long chain of C++ element methods to actually pull packets from the VM. Routing is the core of the Fast Path, and is implemented by the ingress and egress flow table pipeline elements discussed in Section 4.3.

Several elements assist with debugging and monitoring. *Tcpdump* elements allow online packet dumps. *Stats exporter* records internal engine and packet metrics for performance tuning. *Packet tracer* sends metadata to an off-host service for network analysis and debug. *Latency sampler* records metadata for off-host analysis of network RTT, throughput, and other performance information.

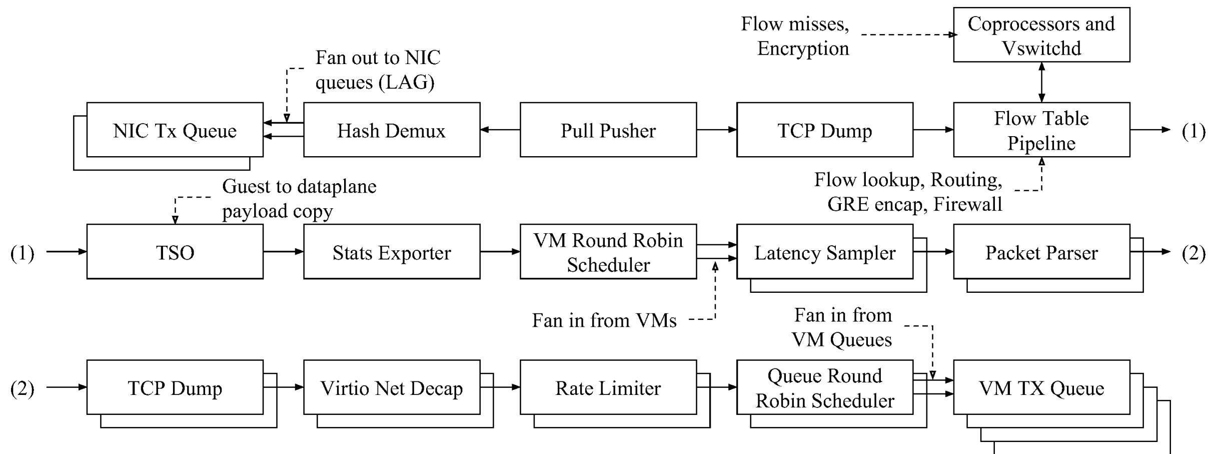


Figure 6: Andromeda dataplane egress engine. Note: Arrows indicate the direction of the push or pull C++ function call chain. Packets flow from the lower right to the upper left.

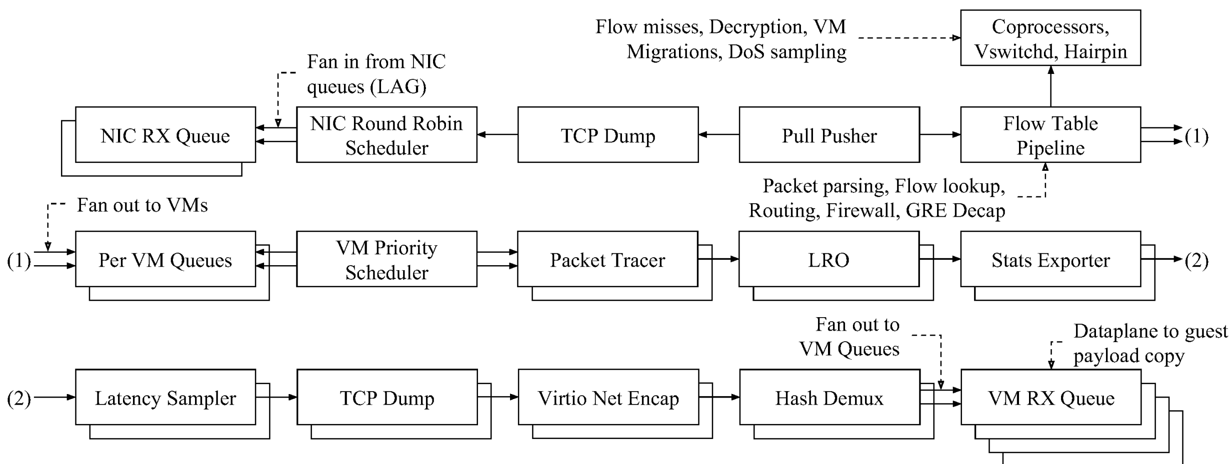


Figure 7: Andromeda dataplane ingress engine. Note: Arrows indicate the direction of the push or pull C++ function call chain. Packets flow from the upper left to the lower right.

4.3 Fast Path Flow Table

All VM packets pass through the engine flow table (FT) for both routing and per-flow packet operations such as encapsulation. The FT is a cache of the vswitchd flow tables. We *minimize per-flow work* through the FT in multiple ways. The FT uses only a single hash table lookup to map the flow key to a dense integer flow index. Subsequent tables, such as the action table and firewall policy table, are flat arrays indexed by flow index. Flow actions perform common packet operations such as encapsulation and setting the Ethernet header. Actions also store the destination virtual switch port and the set of Fast Path and Coprocessor packet stages for the flow, if any. Commonly, VM-VM intra-cluster flows have no packet stages enabled, and complete FT execution after applying the action set to the packet.

To avoid costly synchronization, the FT does not use locks, and is never modified by the engines. To update the FT, a control thread updates a shadow FT and then updates the engine via SPSC channels to point to the new FT [36]. Each engine maintains its own FT flow statistics, which vswitchd periodically reads and aggregates.

When a packet is sent through the FT, the FT computes the flow key for the packet, looks up the key in the flow index table, then applies the specified flow actions and any enabled Fast Path packet stages, and finally updates statistics. If Coprocessor stages are enabled for the flow, the packet is sent to the appropriate Coprocessor thread.

FT keys are either 3-tuple or 5-tuple. Ingress FT keys support encapsulation, and include both the inner and outer packet 3-tuple. Egress FT keys are unencapsulated. Flow index table lookup is attempted first with a 3-tuple flow key. If no match is found, a 5-tuple flow key is computed and lookup is retried. If no match is found again the packet is sent to the Flow Miss Coprocessor, which sends the packet to vswitchd. 3-tuple keys are used wherever possible, and are the common case for VM-VM traffic. Example uses of 5-tuple keys include VM connections to load balanced VIPs, as VIP backend selection is performed on a per-connection basis.

4.3.1 Middlebox Functionality

Andromeda provides middlebox functions such as firewall, load balancing, and NAT on-host [7, 27, 32]. This approach achieves higher performance and reduced provisioning complexity compared to traditional dedicated appliance middleboxes. A key challenge is how to ensure that these features do not degrade Fast Path performance. To accomplish this goal, we perform the expensive middlebox work in the on-host control plane on a flow miss. The control plane inserts the flow into the FT with any middlebox packet stage work pre-computed per-flow.

An example Fast Path feature is the always-on connection tracking firewall [4]. Traditional firewalls require a *firewall rule* lookup and a *connection-tracking table lookup* per packet, both of which are expensive. To *minimize per-flow work*, vswitchd analyzes the rules on a flow miss in order to minimize the amount of work that the Fast Path must do. If the IP addresses and protocol in the flow are always allowed in both directions, then no firewall work is needed in the Fast Path. Otherwise, vswitchd enables the firewall stage and computes a *flow firewall policy*, which indicates which port ranges are allowed for the flow IPs. The Fast Path matches packets against these port ranges, which is much faster than evaluating the full firewall policy. Furthermore, the firewall stage skips connection tracking if a packet connection 5-tuple is permitted by the firewall rules in both directions, which is a common case for VM-VM and server flows.

4.4 Coprocessor Path

The Coprocessor Path implements features that are CPU-intensive or do not have strict latency requirements. Coprocessors play a key role in *minimizing Fast Path features*, and decouple feature growth from Fast Path performance. Developing Coprocessor features is easier because Coprocessors do not have to follow the strict Fast Path best practices. For example, Coprocessor stages may acquire locks, allocate memory, and perform system calls.

Coprocessor stages include encryption, DoS, abuse detection, and WAN traffic shaping. The encryption stage provides transparent encryption of VM to VM traffic across clusters. The DoS and abuse detection stage enforces ACL policies for VM to internet traffic. The WAN traffic shaping stage enforces bandwidth sharing policies [26]. Coprocessor stages are executed in a per-VM floating Coprocessor thread, whose CPU time is attributed to the container of the corresponding VM. This design provides fairness and isolation between VMs, which is critical for more CPU-intensive packet work. A single coprocessor thread performing Internet ACL/shaping can send 11.8Gb/s in one netperf TCP stream, whereas one coprocessor thread performing WAN encryption without hardware offloads can send 4.6Gb/s. This inherent difference in per-packet processing overhead highlights the need for attributing and isolating packet processing work.

Fast Path FT lookup determines the Coprocessor stages enabled for a packet. If Coprocessor stages are enabled, the Fast Path sends the packet to the appropriate Coprocessor thread via an SPSC packet ring, waking the thread if necessary. The Coprocessor thread applies the Coprocessor stages enabled for the packet, and then returns the packet to the Fast Path via a packet ring.

5 Evaluation

This section evaluates the resource consumption, performance, and scalability of Andromeda.

5.1 On-Host Resource Consumption

Andromeda consumes a few percent of the CPU and memory on-host. One physical CPU core is reserved for the Andromeda dataplane. One logical CPU of the core executes the busy polling Fast Path. The other mostly idle logical CPU executes infrequent background work, leaving most of the core's shared resources available to the Fast Path logical CPU. In the future, we plan to increase the dataplane CPU reservation to two physical cores on newer hosts with faster physical NICs and more CPU cores in order to improve VM network throughput.

The Andromeda dataplane has a 1GB max memory usage target. To support non-disruptive upgrades and a separate dataplane lifecycle management daemon, the total dataplane memory container limit is 2.5GB. The combined vswitchd and host agent memory limit is 1.5GB.

5.2 Dataplane Performance

Dataplane performance has improved significantly throughout the evolution of Andromeda.

Pre-Andromeda was implemented entirely in the VMM and used UDP sockets for packet I/O. The dataplane supported only a single queue virtual NIC with zero offloads, such as LRO and TSO.

Andromeda 1.0 included an optimized VMM packet pipeline and a modified kernel Open vSwitch (OVS). We added virtual NIC multi-queue and egress offloads.

Andromeda 1.5 added ingress virtual NIC offloads and further optimized the packet pipeline by coalescing redundant lookups in the VMM with the kernel OVS flow table lookup. Host kernel scheduling and C-State management were also optimized, improving latency.

Andromeda 2.0 consolidated prior VMM and host kernel packet processing into a new OS-bypass busy-polling userspace dataplane. The VMM continued to handle VM virtual NIC ring access and interrupts, but all packet processing was performed by the new dataplane. The VMM exchanged packets with the dataplane via SPSC shared memory rings. The dataplane maps in all VM memory, and directly copies packet to/from VM memory.

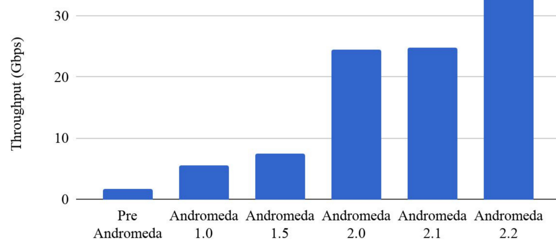


Figure 8: Multi-stream TCP throughput.

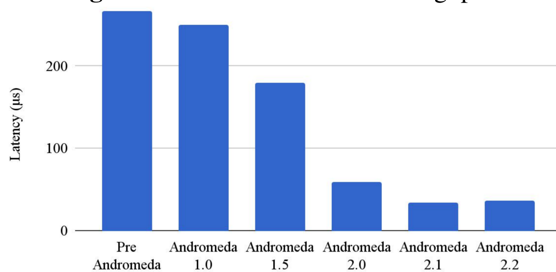


Figure 9: TCP round trip latency.

Andromeda 2.1 directly accesses VM virtual NIC rings in the dataplane, bypassing the VMM. Performance-critical packets are processed end-to-end by the busy-polling dataplane CPU without thread handoffs or context switches. Eliminating the VMM from the packet datapath removed four thread wakeups per network round trip which significantly improved latency and CPU efficiency.

Andromeda 2.2 uses Intel QuickData DMA Engines [2] to offload larger packet copies, improving throughput. DMA engines use an IOMMU for safety and are directly accessed by the dataplane via OS bypass. Tracking async packet copies and maintaining order caused a slight latency increase over Andromeda 2.1 for small packets.

Throughout this evolution, we tracked many performance metrics. Figure 8 plots the same-cluster throughput achievable by two VMs using 200 TCP streams. As the stack evolved, we improved throughput by 19x and improved same-cluster TCP round trip latency by 7x (Figure 9). Figure 10 plots the virtual network CPU efficiency in cycles per byte during a multi-stream benchmark. Our measurement includes CPU usage by the sender and receiver for both the VM and host network dataplane. The host dataplane covers all host network processing, including the VMM, host kernel, and new Andromeda OS bypass dataplane. Overall, Andromeda reduced cycles per byte by a factor of 16. For Andromeda 2.0 and later, we use the host resource limits described in Section 5.1 and execute the Fast Path on a single reserved logical CPU.

We measured these results on unthrottled VMs connected to the same Top of Rack switch. Benchmark hosts have Intel Sandy Bridge CPUs and 40Gb/s physical NICs except for Pre-Andromeda, which used bonded 2x10Gb/s NICs. The sender and receiver VMs run a Linux 3.18

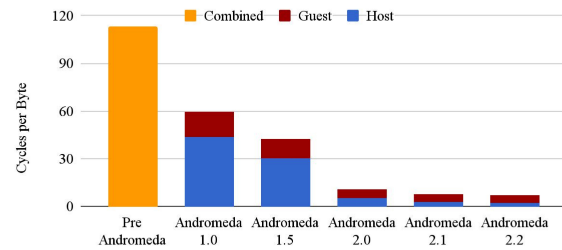


Figure 10: CPU Efficiency. Note: Guest versus host breakdown is unavailable for Pre-Andromeda

guest kernel, and are configured with 8 VCPUs. The only hardware offloads used are checksum offloads, and in Andromeda 2.2, memory copy offloads.

5.3 Control Plane Agility and Scale

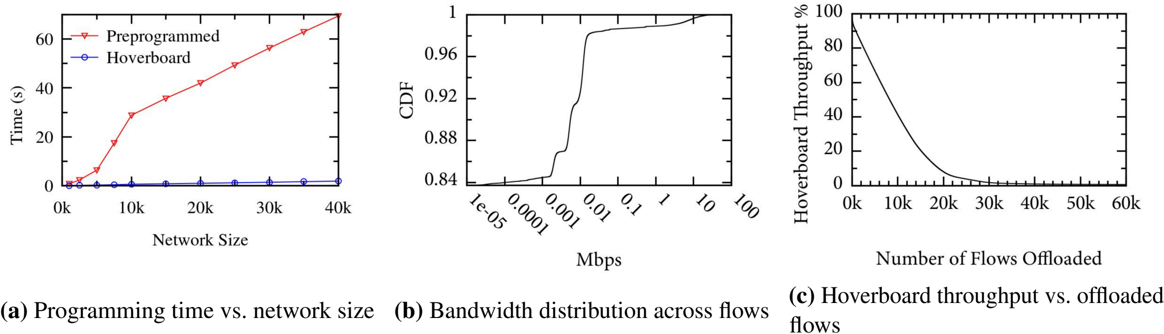
Andromeda 1.0 used the preprogrammed model for all VM-VM flows, and initially supported networks up to 2k VMs. Under the preprogrammed model, even a small change in a virtual network, such as adding a VM, requires updating the routing tables for all other VMs in the network.

We subsequently developed the Hoverboard model, which made the control plane substantially more scalable and agile. With Hoverboards, a new VM has network connectivity as soon as its host and the Hoverboards are programmed. Median programming latency – the time required for the VM controller to process an FM request (e.g., to add a VM to a network) and program the appropriate flow rules via OpenFlow – improved from 551ms to 184ms. The 99th percentile latency dropped even more substantially, from 3.7s to 576ms. Furthermore, the control plane now scales gracefully to virtual networks with 100k VMs.

Figure 11a shows VMC flow programming time for networks with varying numbers of VMs. The control plane has 30 VMC partitions with 8 Broadwell logical CPUs per partition, and 60 OFEs with 4 CPUs each. VMs are scheduled on a fixed pool of 10k physical hosts.

When using the Hoverboard model, the programming time and number of flows grows linearly in network size. However, when we use the preprogrammed model, the programming time and number of flows is $O(n \times h)$, where n is the number of VMs and h is the number of hosts with at least one VM in the network. Multiple VMs on the same host can share a forwarding table; without this optimization, the number of flows would be $O(n^2)$. Programming time grows quadratically in n until $n \approx h$, and linearly thereafter. Quadratic growth is representative of typical multi-regional deployments, for which $n \ll h$.

For the 40k VM network, VMCs program a total of 1.5M flows in 1.9 seconds with the Hoverboard model, using a peak of 513MB of RAM per partition. Under the preprogrammed model, VMCs program 487M flows



(a) Programming time vs. network size **(b)** Bandwidth distribution across flows **(c)** Hoverboard throughput vs. offloaded flows

Figure 11: Hoverboard scaling analysis

in 74 seconds, and peak RAM use per partition is 10GB. Beyond 40k VMs, preprogramming led to stability issues in the OFE and vswitchd due to the large number of flows.

The effectiveness of the Hoverboard model is predicated on the assumption that a small number of offloaded flows installed by the control plane can capture most of the traffic, with a long tail of small flows going through the Hoverboards. Indeed, network flows in our clusters follow a power law distribution, consistent with prior observations, e.g., [19]. Figure 11b shows a CDF of peak throughput across all VM pairs in a production cluster, measured over a 30-minute window. 84% of VM pairs never communicate: in the Hoverboard model, these flows are never programmed, and their host overhead is zero. 98% of flows have peak throughput less than 20kbps, so with an offload threshold of 20kbps, Hoverboards improve control plane scalability by 50x.

Figure 11c shows the peak traffic through the Hoverboard gateways as we program more direct host to host flows. The figure shows that by shifting a total of about 50k flows (less than 0.1% of the total flows possible) to end hosts, the peak throughput through the Hoverboard gateways drops to less than 1% of the cluster bandwidth.

6 Experiences

This section describes our experiences building Andromeda, challenges we faced as the system grew and evolved, and how we addressed those challenges.

6.1 Resource Management

In a Cloud environment, it is essential to provide isolation among tenants while making effective use of resources. Here we discuss CPU and memory; for a description of how we manage network bandwidth, see [26].

6.1.1 CPU Isolation

Andromeda 1.0 shipped with a kernel datapath that provided limited isolation. Ingress traffic was processed on kernel softirq threads shared by all VMs on-host. These softirq threads could run on any host CPU and did not have CPU attribution. The current Andromeda userspace datapath performs Fast Path processing on its own dedicated CPU. Packets requiring CPU-intensive processing are sent to a per-VM coprocessor thread which is CPU attributed to the VM container

As the Fast Path is shared by all VMs on-host, Andromeda provides isolation within the Fast Path. For egress, the Fast Path polls VMs round robin and each VM is rate limited to a VM virtual NIC line rate. Initially, we provided no Fast Path isolation between VMs for ingress. This resulted in noisy neighbor problems when the packet rate of one VM was higher than the Fast Path could pull packets off the physical NIC. The NIC queue would back up and drop packets, harming other VMs on-host.

To improve isolation, we split the Fast Path ingress engine into two parts. The front half polls the NIC, performs flow table lookup, and places packets into per-VM queues. The more CPU intensive back half pulls packets from the per-VM queues, copies the packets to the VM, and may raise a virtual interrupt. In Figure 7, the back half begins after the Per VM Queues stage. Isolation is provided within the back half by the VM Priority Scheduler. The scheduler selects the per-VM queue whose recent back half processing has consumed the least Fast Path CPU. Unfortunately, noisy neighbor issues may still arise if the less CPU intensive front half is overloaded. In the future we will explore use of per-VM physical NIC queues to provide isolation in the front half as well.

6.1.2 Guest VM Memory

The Andromeda dataplane maps in all of guest VM memory for all VMs on-host, as any guest memory address may be used for packet I/O. However, dataplane access to VM memory creates attribution and robustness challenges. Guest VM memory is backed by a host tmpfs file and is demand-allocated. Dataplane access to an unbacked VM memory page causes a page allocation, which is typically charged to the process triggering the allocation. To ensure that the dataplane does not exceed its memory limits (Section 5.1), we modified our host Linux kernel to always charge VM memory page allocations to the VM memory container rather than the process triggering the allocation.

The dataplane memory container is also charged for kernel page table memory. Over time, the dataplane can access hundreds of GB of VM memory. If page table memory usage exceeds a target limit, Andromeda asks the kernel to free page table entries. This is done by mmaping the VM memory file over its existing mapping, which atomically replaces the existing mapping. Remapping a guest VM memory region is done in multiple mmap calls over small chunks rather than one system call. This avoids tail latency due to kernel memory lock contention.

A VM can crash while the dataplane is accessing VM memory. A compromised VMM could also truncate the VM memory backing file. To handle these cases, all dataplane VM memory access occurs via custom assembly memcpy functions. On VM memory access failure, the dataplane receives a signal. The dataplane signal handler modifies the signal saved registers so that execution resumes at a custom memcpy fixup handler, which returns failure to the memcpy caller. On memcpy failure, the dataplane disconnects the offending VM's queues. We must minimize the Fast Path cost relative to normal memcpy in the common case of VM memory access success. Our approach requires only a single extra branch to test the memcpy return value. This is similar to how the Linux kernel handles failure during memory copies to userspace.

6.1.3 Memory Provisioning

Our Cloud environment serves a diverse set of applications which place varying memory demands on the network stack due to routing tables, firewall connection tracking, load balancing, etc. Therefore, a key challenge is how to provision memory with minimal waste.

We initially expected network virtualization features to span the Top of Rack switch (ToR), switch fabric, and host machine software. The fact that we already managed ToR and switch fabrics with OpenFlow contributed to our decision to use OpenFlow for Andromeda. However, we found that scaling a feature to many virtual networks is much easier on an end host where we can provision additional memory and CPU per network.

Currently we statically provision dataplane memory on each host (Section 5.1), regardless of the number of VMs running on the host or how those VMs use networking features. We are exploring attribution of dataplane networking memory usage to VM containers so that network features may dynamically scale memory usage. This approach reduces dataplane memory overprovisioning and allows the cluster manager to take network memory usage for a VM into account during VM placement.

6.2 Velocity

The Cloud ecosystem is evolving rapidly, so a key challenge was to build a platform with high feature velocity. Our strategy for achieving velocity has evolved over time.

6.2.1 Andromeda 1.0: Kernel Datapath

Andromeda 1.0 shipped with the Open vSwitch kernel datapath. Kernel upgrades were much slower than our control plane release cycle. The OpenFlow APIs provide a flexible flow programming model that allowed us to deploy a number of features with only control plane changes. OpenFlow was instrumental in getting the project off the ground and delivering some of the early features, such as load balancing and NAT, via control plane changes alone.

However, we also faced a number of difficulties. For example, OpenFlow was not designed to support stateful features such as connection tracking firewalls and load balancing. We initially tried to express firewall rules in OpenFlow. However, expressing firewall semantics in terms of generic primitives was complex, and the implementation was difficult to optimize and scale. We ultimately added an extension framework in OVS to support these features. Unlike OpenState [8] and VFP [13], which integrate stateful primitives into the flow lookup model, our extensions use a separate configuration push mechanism.

6.2.2 Andromeda 2.0: Userspace Datapath

Qualifying and deploying a new kernel to a large fleet of machines is intrinsically a slow process. Andromeda 2.0 replaced the kernel dataplane with a userspace OS bypass dataplane, which enabled weekly dataplane upgrades. At that point, the advantage of having a programming abstraction as generic as OpenFlow diminished.

A rapid dataplane release cycle requires non-disruptive updates and robustness to rare but inevitable dataplane bugs. To provide non-disruptive updates, we use an upgrade protocol consisting of a brownout and blackout phase. During brownout, the new dataplane binary starts and transfers state from the old dataplane. The old dataplane continues serving during brownout. After the initial state transfer completes, the old dataplane stops serving and blackout begins. The old dataplane then transfers delta state to the new dataplane, and the new dataplane begins serving. Blackout ends for a VM when the VMM connects to the new dataplane and attaches its VM virtual NIC. The median blackout time is currently 270ms. We plan to reduce blackout duration by passing VMM connection file descriptors as part of state transfer to avoid VMM reconnect time.

The userspace dataplane improves robustness to dataplane bugs. The VM and host network can be hardened so that a userspace dataplane crash only results in a temporary virtual network outage for the on-host VMs. On-host control services continuously monitor the health of the dataplane, restarting the dataplane if health checks fail. Host kernel networking uses separate queues in the NIC, so we can roll back dataplane releases even if the dataplane is down. In contrast, a failure in the Linux kernel network stack typically takes down the entire host.

6.3 Scaling and Agility

Our initial pre-Andromeda virtual network used a global control plane, which pre-programmed routes for VM-VM traffic, but also supported on-demand lookups to reduce perceived programming latency. The on-demand model was not robust under load. When the control plane fell behind, all VM hosts would request on-demand programming, further increasing the load on the control plane, leading to an outage.

Andromeda initially used OpenFlow and the pre-programmed model exclusively. As we scaled to large networks, we ran into OpenFlow limitations. For example, supporting a million-entry IP forwarding table across a large number of hosts requires the control plane to transmit a compact representation of the routes to the dataplane. Expressing such a large number of routes in OpenFlow adds unacceptable overhead. As another example, the Reverse Path Forwarding check in OpenFlow required us to duplicate data that was already present in the forwarding table, and we built a special-purpose extension to avoid this overhead. We also scaled the control plane by partitioning VM Controllers and parallelizing flow generation. However, the pre-programmed model's quadratic scaling (see Section 5.3) continued to create provisioning challenges, particularly for on-demand customer workloads.

Later, we introduced the Hoverboard model, which lead to more stable control plane overhead and enabled much faster provisioning of networks (see Section 5.3). Hoverboards have allowed us to scale to support large virtual networks. However, certain workloads create challenges. For example, batch workloads in which many VM pairs begin communicating simultaneously at high bandwidth consume substantial Hoverboard capacity until the control plane can react to offload the flows. We have made a number of improvements to address that, such as detecting and offloading high-bandwidth flows faster.

7 Related Work

The Andromeda control plane builds upon a wide body of software defined networking research [10, 15, 25], OpenFlow [3, 28] and Open vSwitch [33]. The data plane design overlaps with concepts described in Click [29], SoftNIC [17], and DPDK [1].

NVP [24] is a SDN-based network virtualization stack, like Andromeda. NVP also uses Open vSwitch, along with the OVS kernel datapath, similar to Andromeda 1.0. The NVP control plane uses the preprogrammed model (Section 3.3), so a network with n VMs will have $O(n^2)$ flows. NVP scales by using partitioning, and by dividing the control plane into virtual and physical layers. Andromeda also uses partitioning, but principally solves the scaling issue by using the Hoverboard model.

VFP [13] is the SDN-based virtualization host stack for Microsoft Azure, using a layered match-action table model with stateful rules. All VFP features support offloading to an exact-match fastpath implemented in the host kernel or an SR-IOV NIC. Andromeda uses a hierarchy of flexible software-based userspace packet processing paths. Relative to VFP, Coprocessors enable rapid iteration on features that are CPU-intensive or do not have strict latency targets, and allow these features to be built with high performance outside of the Fast Path. Andromeda does not rely on offloading entire flows to hardware: We demonstrate that a flexible software pipeline can achieve performance competitive with hardware. Our Fast Path supports 3-tuple flow lookups and minimizes use of stateful features such as firewall connection tracking. While the VFP paper does not focus on the control plane, we present our experiences and approach to building a highly scalable, agile, and reliable network control plane.

8 Conclusions

This paper presents the design principles and deployment experience with Andromeda, Google Cloud Platform's network virtualization stack. We show that an OS bypass software datapath provides performance competitive with hardware, achieving 32.8Gb/s using a single core. To achieve isolation and decouple feature growth from fastpath performance, we execute CPU-intensive packet work on per-VM Coprocessor threads. The Andromeda Control plane is designed for agility, availability, isolation, feature velocity, and scalability. We introduce Hoverboards, which makes it possible to program connectivity for tens of thousands of VMs in seconds. Finally, we describe our experiences deploying Andromeda, and we explain how non-disruptive upgrades and VM live migration were critical to navigating major architectural shifts in production with essentially no customer impact.

In the future, we will offload more substantial portions of the Fast Path to hardware. We will continue to improve scalability, as lightweight virtual network endpoints such as containers will result in much larger and more dynamic virtual networks.

Acknowledgements

We would like to thank our reviewers, shepherd Michael Kaminsky, Jeff Mogul, Rama Govindaraju, Aspi Siganporia, and Parthasarathy Ranganathan for paper feedback and guidance. We also thank the following individuals, who were instrumental in the success of the project: Aspi Siganporia, Alok Kumar, Andres Lagar-Cavilla, Bill Sommerfeld, Carlo Contavalli, Frank Swiderski, Jerry Chu, Mike Bennett, Mike Ryan, Pan Shi, Phillip Wells, Phong Chuong, Prashant Chandra, Rajiv Ranjan, Rüdiger Sonderfeld, Rudo Thomas, Shay Raz, Siva Sunkavalli, and Yong Ni.

References

[1] Intel data plane development kit. http://www.intel.com/go/dpdk.

[2] Intel I/O acceleration technology. https://www.intel.com/content/www/us/en/wireless-network/accel-technology.html.

[3] Openflow specification. https://www.opennetworking.org/software-defined-standards/specifications/.

[4] Using networks and firewalls. https://cloud.google.com/compute/docs/networking.

[5] M. Baker-Harvey. Google compute engine uses live migration technology to service infrastructure without application downtime. https://cloudplatform.googleblog.com/2015/03/Google-Compute-Engine-uses-Live-Migration-technology-to-service-infrastructure-without-application-downtime.html.

[6] 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 *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[7] S. M. Bellovin. Distributed firewalls. *IEEE Communications Magazine*, 32:50–57, 1999.

[8] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM*, 44(2):44–51, Apr. 2014.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[10] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 1–12, New York, NY, USA, 2007. ACM.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[12] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 523–535, Berkeley, CA, USA, 2016. USENIX Association.

[13] D. Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, 2017. USENIX Association.

[14] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[15] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, Oct. 2005.

[16] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97, Feb. 2015.

[17] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[18] M. R. Hines, U. Deshpande, and K. Gopalan. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.*, 43(3):14–26, July 2009.

[19] R. Jain. Characteristics of destination address locality in computer networks: A comparison of caching schemes. *Computer Networks and ISDN Systems*, 18:243–254, 1989.

[20] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[21] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for NFV: Simplifying middlebox modifications using statealyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 239–253, Santa Clara, CA, 2016. USENIX Association.

[22] C. Kim, M. Caesar, and J. Rexford. Floodless in Seattle: A scalable ethernet architecture for large enterprises. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 3–14, New York, NY, USA, 2008. ACM.

[23] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, 2014. USENIX Association.

[24] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, 2014. USENIX Association.

[25] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Z. Google, R. Ramanathan, Y. I. NEC, H. I. NEC, T. H. NEC, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 351–364, Berkeley, CA, USA, 2010.

[26] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 1–14, New York, NY, USA, 2015. ACM.

[27] J. Lee, J. Tourrilhes, P. Sharma, and S. Banerjee. No more middlebox: Integrate processing into network. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 459–460, New York, NY, USA, 2010. ACM.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[29] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 217–231, New York, NY, USA, 1999. ACM.

[30] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference*, USENIX '05, pages 391–394, Berkeley, CA, USA, 2005. USENIX Association.

[31] 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 *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.

[32] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[33] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, 2015. USENIX Association.

[34] L. Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[35] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: Live router migration as a network-management primitive. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 231–242, New York, NY, USA, 2008. ACM.

[36] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 432–445, New York, NY, USA, 2017. ACM.

LHD: Improving Cache Hit Rate by Maximizing Hit Density

Nathan Beckmann
Carnegie Mellon University
beckmann@cs.cmu.edu

Haoxian Chen
University of Pennsylvania
hxchen@seas.upenn.edu

Asaf Cidon
Stanford University/Barracuda Networks
asaf@cidon.com

Abstract

Cloud application performance is heavily reliant on the hit rate of datacenter key-value caches. Key-value caches typically use least recently used (LRU) as their eviction policy, but LRU's hit rate is far from optimal under real workloads. Prior research has proposed many eviction policies that improve on LRU, but these policies make restrictive assumptions that hurt their hit rate, and they can be difficult to implement efficiently.

We introduce least hit density (LHD), a novel eviction policy for key-value caches. LHD predicts each object's expected hits-per-space-consumed (*hit density*), filtering objects that contribute little to the cache's hit rate. Unlike prior eviction policies, LHD does not rely on heuristics, but rather rigorously models objects' behavior using conditional probability to adapt its behavior in real time.

To make LHD practical, we design and implement RankCache, an efficient key-value cache based on memcached. We evaluate RankCache and LHD on commercial memcached and enterprise storage traces, where LHD consistently achieves better hit rates than prior policies. LHD requires much less space than prior policies to match their hit rate, on average 8× less than LRU and 2–3× less than recently proposed policies. Moreover, RankCache requires no synchronization in the common case, improving request throughput at 16 threads by 8× over LRU and by 2× over CLOCK.

1 Introduction

The hit rate of distributed, in-memory key-value caches is a key determinant of the end-to-end performance of cloud applications. Web application servers typically send requests to the cache cluster over the network, with latencies of about 100 µs, before querying a much slower database, with latencies of about 10 ms. Small increases in cache hit rate have an outsize impact on application performance. For example, increasing hit rate by just 1% from 98% to 99% halves the number of requests to the database. With the latency numbers used above, this decreases the mean service time from 210 µs to 110 µs (nearly 2×) and, importantly for cloud applications, halves the tail of long-latency requests [21].

To increase cache hit rate, cloud providers typically scale the number of servers and thus total cache capacity [37]. For example, Facebook dedicates tens of thousands of continuously running servers to caching. However, adding servers is not tenable in the long run, since hit rate increases logarithmically as a function of cache capacity [3, 13, 20]. Prohibitively large amounts of memory are needed to significantly impact hit rates.

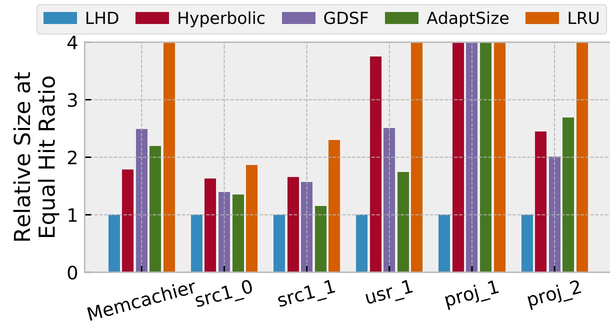


Figure 1: Relative cache size needed to match LHD's hit rate on different traces. LHD requires roughly one-fourth of LRU's capacity, and roughly half of that of prior eviction policies.

This paper argues that improving the eviction policy is much more effective, and that there is significant room to improve cache hit rates. Popular key-value caches (e.g., memcached, Redis) use least recently used (LRU) or variants of LRU as their eviction policy. However, LRU is far from optimal for key-value cache workloads because: *(i)* LRU's performance suffers when the workload has variable object sizes, and *(ii)* common access patterns expose pathologies in LRU, leading to poor hit rate.

These shortcomings of LRU are well documented, and prior work has proposed many eviction policies that improve on LRU [4, 14, 16, 25, 35, 38, 40]. However, these policies are not widely adopted because they typically require extensive parameter tuning, which makes their performance unreliable, and globally synchronized state, which hurts their request throughput. Indeed, to achieve acceptable throughput, some systems use eviction policies such as CLOCK or FIFO that sacrifice hit rate to reduce synchronization [22, 33, 34].

More fundamentally, prior policies make assumptions that do not hold for many workloads, hurting their hit rate. For example, most policies prefer recently used objects, all else equal. This is reasonable—such objects are often valuable—, but workloads often violate this as-

sumption. Prior policies handle the resulting pathologies by adding new mechanisms. For example, ARC [35] adds a second LRU list for newly admitted objects, and AdaptSize [9] adds a probabilistic filter for large objects.

We take a different approach. Rather than augmenting or recombining traditional heuristics, we seek a new mechanism that just "does the right thing". The key motivating question for this paper is: *What would we want to know about objects to make good caching decisions, independent of workload?*

Our answer is a metric we call *hit density*, which measures how much an object is expected to contribute to the cache's hit rate. We infer each object's hit density from what we know about it (e.g., its age or size) and then evict the object with least hit density (LHD). Finally, we present an efficient and straightforward implementation of LHD on memcached called RankCache.

1.1 Contributions

We introduce ***hit density***, an intuitive, workload-agnostic metric for ranking objects during eviction. We arrive at hit density from first principles, without any assumptions about how workloads tend to reference objects.

Least hit density (LHD) is an eviction policy based on hit density. LHD monitors objects online and uses conditional probability to predict their likely behavior. LHD draws on many different object features (e.g., age, frequency, application id, and size), and easily supports others. Dynamic ranking enables LHD to adapt its eviction strategy to different application workloads over time without any hand tuning. For example, on a certain workload, LHD may initially approximate LRU, then switch to most recently used (MRU), least frequently used (LFU), or a combination thereof.

RankCache is a key-value cache based on memcached that efficiently implements LHD (and other policies). RankCache supports arbitrary *ranking functions*, making policies like LHD practical. RankCache approximates a true global ranking while requiring no synchronization in the common case, and adds little implementation complexity over existing LRU caches. RankCache thus avoids the unattractive tradeoff in prior systems between hit rate and request throughput, showing it is possible to achieve the best of both worlds.

1.2 Summary of Results

We evaluate LHD on a weeklong commercial memcached trace from Memcachier [36] and storage traces from Microsoft Research [48]. LHD significantly improves hit rate prior policies—e.g., reducing misses by half vs. LRU and one-quarter vs. recent policies—and also avoids pathologies such as performance cliffs that afflict prior policies. Fig. 1 shows the cache size (i.e., number of caching servers) required to achieve the same hit rate as LHD at 256 MB on Memcachier and 64 GB on Microsoft traces. LHD requires much less space than prior eviction policies, saving the cost of thousands of servers in a modern datacenter. On average, LHD needs 8× less space than LRU, 2.4× less than GDSF [4], 2.5× less than Hyperbolic [11], and 2.9× less than AdaptSize [9]. Finally, at 16 threads, RankCache achieves 16× higher throughput than list-based LRU and, at 90% hit rate, 2× higher throughput than CLOCK.

2 Background and Motivation

We identify two main opportunities to improve hit rate beyond existing eviction policies. First, prior policies make implicit assumptions about workload behavior that hurt their hit rate when they do not hold. Second, prior policies rely on implementation primitives that unnecessarily limit their design. We avoid these pitfalls by going back to first principles to design LHD, and then build RankCache to realize it practically.

2.1 Implicit assumptions in eviction policies

Eviction policies show up in many contexts, e.g., OS page management, database buffer management, web proxies, and processors. LRU is widely used because it is intuitive, simple to implement, performs reasonably well, and has some worst-case guarantees [12, 47].

However, LRU also has common pathologies that hurt its performance. LRU uses only *recency*, or how long it has been since an object was last referenced, to decide which object to evict. In other words, LRU assumes that recently used objects are always more valuable. But common access patterns like scans (e.g., AB. . . Z AB. . . Z . . .) violate this assumption. As a result, LRU caches are often polluted by infrequently accessed objects that stream through the cache without reuse.

Prior eviction policies improve on LRU in many different ways. Nearly all policies augment recency with additional mechanisms that fix its worst pathologies. For example, ARC [35] uses two LRU lists to distinguish newly admitted objects and limit pollution from infrequently accessed objects. Similarly, AdaptSize [9] adds a probabilistic filter in front of an LRU list to limit pollution from large objects. Several recent policies split accesses across multiple LRU lists to eliminate performance cliffs [6, 18, 51] or to allocate space across objects of different sizes [10, 17, 18, 37, 41, 43, 49].

All of these policies use LRU lists as a core mechanism, and hence retain recency as built-in assumption. Moreover, their added mechanisms can introduce new assumptions and pathologies. For example, ARC assumes that frequently accessed objects are more valuable by placing them in a separate LRU list from newly admitted objects and preferring to evict newly admitted objects. This is often an improvement on LRU, but can behave pathologically.

Other policies abandon lists and rank objects using a heuristic function. GDSF [4] is a representative example. When an object is referenced, GDSF assigns its rank using its frequency (reference count) and global value L:

$$\text{GDSF Rank} = \frac{\text{Frequency}}{\text{Size}} + L \quad (1)$$

On a miss, GDSF evicts the cached object with the lowest rank and then updates L to this victim's rank. As a result, L increases over time so that recently used objects have higher rank. GDSF thus orders objects according to some combination of recency, frequency, and size. While it is intuitive that each of these factors should play some role, it is not obvious why GDSF combines them in this particular formula. Workloads vary widely (Sec. 3.5), so no factor will be most effective in general. Eq. 1 makes implicit assumptions about how important each factor will be, and these assumptions will not hold across all workloads. Indeed, subsequent work [16, 27] added weighting parameters to Eq. 1 to tune GDSF for different workloads.

Hence, while prior eviction policies have significantly improved hit rates, they still make implicit assumptions that lead to sub-optimal decisions. Of course, all online policies must make some workload assumptions (e.g., adversarial workloads could change their behavior arbitrarily [47]), but these should be minimized. We believe the solution is *not* to add yet more mechanisms, as doing so quickly becomes unwieldy and requires yet more assumptions to choose among mechanisms. Instead, our goal is to find a new mechanism that leads to good eviction decisions across a wide range of workloads.

2.2 Implementation of eviction policies

Key-value caches, such as memcached [23] and Redis [1], are deployed on clusters of commodity servers, typically based on DRAM for low latency access. Since DRAM caches have a much lower latency than the backend database, the main determinant of end-to-end request latency is cache hit rate [19, 37].

Request throughput: However, key-value caches must also maintain high request throughput, and the eviction policy can significantly impact throughput. Table 1 summarizes the eviction policies used by several popular and recently proposed key-value caches.

Most key-value caches use LRU because it is simple and efficient, requiring $O(1)$ operations for admission, update, and eviction. Since naïve LRU lists require global synchronization, most key-value caches in fact use approximations of LRU, like CLOCK and FIFO, that eliminate synchronization except during evictions [22, 33, 34]. Policies that use more complex ranking (e.g., GDSF) pay a price in throughput to maintain an ordered ranking (e.g., $O(\log N)$ operations for a min-heap) and to synchronize other global state (e.g., L in Eq. 1).

Key-Value Cache	Allocation	Eviction Policy
memcached [23]	Slab	LRU
Redis [1]	jemalloc	LRU
Memshare [19]	Log	LRU
Hyperbolic [11]	jemalloc	GD
Cliffhanger [18]	Slab	LRU
GD-Wheel [32]	Slab	GD
MICA [34]	Log	≈LRU
MemC3 [22]	Slab	≈LRU

Table 1: Allocation and eviction strategies of key-value caches. GD-Wheel and Hyperbolic's policy is based on GreedyDual [53]. We discuss a variant of this policy (GDSF) in Sec. 2.1.

For this reason, most prior policies restrict themselves to well-understood primitives, like LRU lists, that have standard, high-performance implementations. Unfortunately, these implementation primitives restrict the design of eviction policies, preventing policies from retaining the most valuable objects. List-based policies are limited to deciding how the lists are connected and and which objects to admit to which list. Similarly, to maintain data structure invariants, policies that use min-heaps (e.g., GDSF) can change ranks only when an object is referenced, limiting their dynamism.

We ignore such implementation restrictions when designing LHD (Sec. 3), and consider how to implement the resulting policy efficiently in later sections (Secs. 4 & 5).

Memory management: With objects of highly variable size, another challenge is memory fragmentation. Key-value caches use several memory allocation techniques (Table 1). This paper focuses on the most common one, slab allocation. In slab allocation, memory is divided into fixed 1 MB *slabs*. Each slab can store objects of a particular size range. For example, a slab can store objects between 0–64 B, 65–128 B, or 129–256 B, etc. Each object size range is called a *slab class*.

The advantages of slab allocation are its performance and bounded fragmentation. New objects always replace another object of the same slab class, requiring only a single eviction to make space. Since objects are always inserted into their appropriate slab classes, there is no external fragmentation, and internal fragmentation is bounded. The disadvantage is that the eviction policy is implemented on each slab class separately, which can hurt overall hit rate when, e.g., the workload shifts from larger to smaller objects.

Other key-value caches take different approaches. However, non-copying allocators [1] suffer from fragmentation [42], and log-structured memory [19, 34, 42] requires a garbage collector that increases memory bandwidth and CPU consumption [19]. RankCache uses slab-based allocation due to its performance and bounded fragmentation, but this is not fundamental, and LHD could be implemented on other memory allocators.

3 Replacement by Least Hit Density (LHD)

We propose a new replacement policy, LHD, that dynamically predicts each object's *expected hits per space consumed*, or *hit density*, and evicts the object with the lowest. By filtering out objects that contribute little to the cache's hit rate, LHD gradually increases the average hit rate. Critically, LHD avoids ad hoc heuristics, and instead ranks objects by rigorously modeling their behavior using conditional probability. This section presents LHD and shows its potential in an idealized setting. The following sections will present RankCache, a high-performance implementation of LHD.

3.1 Predicting an object's hit density

Our key insight is that policies must account for both *(i)* the probability an object will hit in its lifetime; and *(ii)* the resources it will consume. LHD uses the following function to rank objects:

$$\text{Hit density} = \frac{\text{Hit probability}}{\text{Object size} \times \text{Expected time in cache}} \tag{2}$$

Eq. 2 measures an object's contribution to the cache's hit rate (in units of hits per byte-access). We first provide an example that illustrates how this metric adapts to real-world applications, and then show how we derived it.

3.2 LHD on an example application

To demonstrate LHD's advantages, consider an example application that scans repeatedly over a few objects, and accesses many other objects with Zipf-like popularity distribution. This could be, for example, the common media for a web page (scanning) plus user-specific content (Zipf). Suppose the cache can fit the common media and some of the most popular user objects. In this case, each scanned object is accessed frequently (once per page load for all users), whereas each Zipf-like object is accessed much less frequently (only for the same user). The cache should ideally therefore *keep the scanned objects and evict the Zipf-like objects when necessary*.

Fig. 2a illustrates this application's access pattern, namely the distribution of time (measured in accesses) between references to the same object. Scanned objects produce a characteristic peak around a single reference time, as all are accessed together at once. Zipf-like objects yield a long tail of reference times. Note that in this example 70% of references are to the Zipf-like objects and 30% to scanned objects, but the long tail of popularity in Zipf-like objects leads to a low reference probability in Fig. 2a.

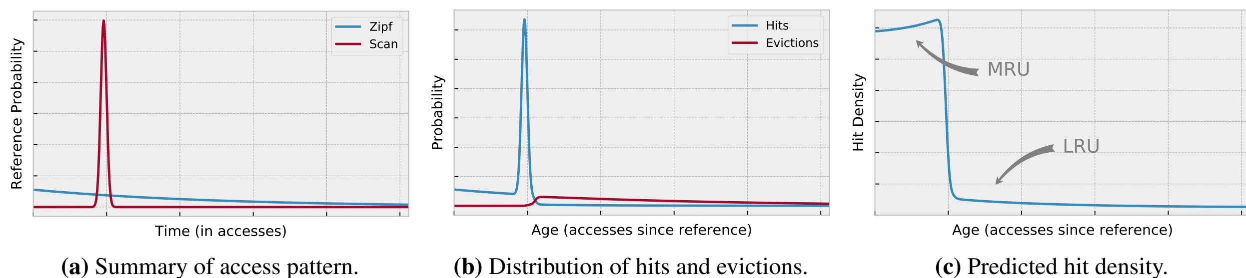
Fig. 2b illustrates LHD's behavior on this example application, showing the distribution of hits and evictions vs. an object's *age*. Age is the number of accesses since an object was last referenced. For example, if an object enters the cache at access T, hits at accesses $T+4$ and $T+6$, and is evicted at access $T+12$, then it has two hits at age 4 and 2 and is evicted at age 6 (each reference resets age to zero). Fig. 2b shows that LHD keeps the scanned objects and popular Zipf references, as desired.

LHD does not know whether an object is a scanned object or a Zipf-like object until ages pass the scanning peak. It must conservatively protect all objects until this age, and all references at ages less than the peak therefore result in hits. LHD begins to evict objects immediately after the peak, since it is only at this point it knows that any remaining objects must be Zipf-like objects, and it can safely evict them.

Finally, Fig. 2c shows how LHD achieves these outcomes. It plots the predicted hit density for objects of different ages. The hit density is high up until the scanning peak, because LHD predicts that objects are potentially one of the scanned objects, and might hit quickly. It drops after the scanning peak because it learns they are Zipf objects and therefore unlikely to hit quickly.

Discussion: Given that LHD evicts the object with the lowest predicted hit density, what is its emergent behavior on this example? The object ages with the lowest predicted hit density are those that have aged past the scanning peak. These are guaranteed to be Zipf-like objects, and their hit density decreases with age, since their implied popularity decreases the longer they have not been referenced. LHD thus evicts older objects; i.e., LRU.

However, if no objects older than the scanning peak are available, LHD will prefer to evict the *youngest* objects, since these have the lowest hit density. This is



(a) Summary of access pattern. **(b)** Distribution of hits and evictions. **(c)** Predicted hit density.

Figure 2: How LHD performs on an application that scans over 30% of objects and Zipf over the remaining 70%.

the most recently used (MRU) eviction policy, or *anti-LRU*. MRU is the correct policy to adopt in this example because *(i)* without more information, LHD cannot distinguish between scanning and Zipf-like objects in this age range, and *(ii)* MRU guarantees that some fraction of the scanning objects will survive long enough to hit. Because scanning objects are by far the most frequently accessed objects (Fig. 2a), keeping as many scanned objects as possible maximizes the cache's hit rate, even if that means evicting some popular Zipf-like objects.

Overall, then, LHD prefers to evict objects older than the scanning peak and evicts LRU among these objects, and otherwise evicts MRU among younger objects. This policy caches as many of the scanning objects as possible, and is the best strictly age-based policy for this application. LHD adopts this policy automatically based on the cache's observed behavior, without any pre-tuning required. By adoping MRU for young objects, LHD avoids the potential performance cliff that recency suffers on scanning patterns. We see this behavior on several traces (Sec. 3.5), where LHD significantly outperforms prior policies, nearly all of which assume recency.

3.3 Analysis and derivation

To see how we derived hit density, consider the cache in Fig. 3. Cache space is shown vertically, and time increases from left to right. (Throughout this paper, time is measured in accesses, not wall-clock time.) The figure shows how cache space is used over time: each block represents an object, with each reference or eviction starting a new block. Each block thus represents a single *object lifetime*, i.e., the idle time an object spends in the cache between hits or eviction. Additionally, each block is colored green or red, indicating whether it ends in a hit or eviction, respectively.

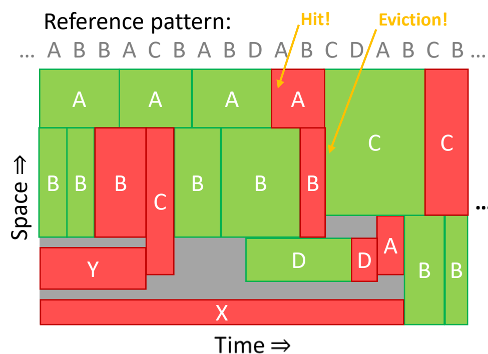


Figure 3: Illustration of a cache over time. Each block depicts a single object's lifetime. Lifetimes that end in hits are shown in green, evictions in red. Block size illustrates resources consumed by an object; hit density is inversely proportional to block size.

Fig. 3 illustrates the challenge replacement policies face: they want to maximize hits given limited resources. In other words, they want to fit as many green blocks into the figure as possible. Each object takes up resources proportional to *both* its size (block height) and the time it spends in the cache (block width). Hence, the replacement policy wants to keep *small objects that hit quickly*.

This illustration leads directly to hit density. Integrating uniformly across the entire figure, each green block contributes 1 hit spread across its entire block. That is, resources in the green blocks contribute hits at a rate of: $1\ \text{hit}/(\text{size} \times \text{lifetime})$. Likewise, lifetimes that end in eviction (or space lost to fragmentation) contribute zero hits. Thus, if there are N hits and M evictions, and if object i has size S_i bytes and spends L_i accesses in the cache, then the cache's overall hit density is:

$$\frac{\sum_{\text{Lifetimes}} \overbrace{1 + 1 + ... + 1}^{\text{Hits}} \quad + \quad \overbrace{0 + 0 + ... + 0}^{\text{Evictions}}}{\sum_{\text{Lifetimes}} \underbrace{S_1 \times L_1 + ... + S_N \times L_N}_{\text{Hit resources}} + \underbrace{S_1 \times L_1 + ... + S_M \times L_M}_{\text{Eviction resources}}}$$

The cache's overall hit density is directly proportional to its hit rate, so maximizing hit density also maximizes the hit rate. Furthermore, it follows from basic arithmetic that replacing an object with one of higher density will increase the cache's overall hit density.[1]

LHD's challenge is to predict an object's hit density, without knowing whether it will result in a hit or eviction, nor how long it will spend in the cache.

Modeling object behavior: To rank objects, LHD must compute their hit probability and the expected time they will spend in the cache. (We assume that an object's size is known and does not change.) LHD infers these quantities in real-time using probability distributions. Specifically, LHD uses distributions of hit and eviction age.

The simplest way to infer hit density is from an object's age. Let the random variables H and L give hit and lifetime age; that is, $\mathbb{P}[H = a]$ is the probability that an object hits at age a, and $\mathbb{P}[L = a]$ is the probability that an object is hit *or* evicted at age a. Now consider an object of age a. Since the object has reached age a, we know it cannot hit or be evicted at any age earlier than a. Its hit probability conditioned on age a is:

$$\text{Hit probability} = \mathbb{P}[\text{hit}|\text{age } a] = \frac{\mathbb{P}[H > a]}{\mathbb{P}[L > a]} \qquad (3)$$

Similarly, its expected remaining lifetime[2] is:

$$\text{Lifetime} = \mathbb{E}[L - a|\text{age } a] = \frac{\sum_{x=1}^{\infty} x \cdot \mathbb{P}[L = a+x]}{\mathbb{P}[L > a]} \qquad (4)$$

Altogether, the object's hit density at age a is:

$$\text{Hit density}_{\text{age}}(a) = \frac{\sum_{x=1}^{\infty} \mathbb{P}[H = a + x]}{\text{Size} \cdot (\sum_{x=1}^{\infty} x \cdot \mathbb{P}[L = a+x])} \qquad (5)$$

[1] Specifically, if $a/b > c/d$, then $(a + c)/(b + d) > c/d$.
[2] We consider the *remaining* lifetime to avoid the sunk-cost fallacy.

3.4 Using classification to improve predictions

One nice property of LHD is that it intuitively and rigorously incorporates additional information about objects. Since LHD is based on conditional probability, we can simply condition the hit and eviction age distributions on the additional information. For example, to incorporate reference frequency, we count how many times each object has been referenced and gather separate hit and eviction age distributions for each reference count. That is, if an object that has been referenced twice is evicted, LHD updates only the eviction age distribution *of objects that have been referenced twice*, and leaves the other distributions unchanged. LHD then predicts an object's hit density using the appropriate distribution during ranking.

To generalize, we say that an object belongs to an equivalence class c; e.g., c could be all objects that have been referenced twice. LHD predict this object's hit density as:

$$\text{Hit density}(a, c) = \frac{\sum_{x=1}^{\infty} \mathbb{P}[H = a + x|c]}{\text{Size} \cdot \left(\sum_{x=1}^{\infty} x \cdot \mathbb{P}[L = a+x|c]\right)} \quad (6)$$

where $\mathbb{P}[H = a|c]$ and $\mathbb{P}[L = a|c]$ are the conditional hit and lifetime age distributions for class c.

3.5 Idealized evaluation

To demonstrate LHD's potential, we simulate an idealized implementation of LHD that globally ranks objects. Our figure of merit is the cache's miss ratio, i.e., the fraction of requests resulting in misses. To see how miss ratio affects larger system tradeoffs, we consider the cache size needed to achieve equal miss ratios.

Methodology: Unfortunately, we are unaware of a public trace of large-scale key-value caches. Instead, we evaluate two sets of traces: *(i)* a weeklong, commercial trace provided by Memcachier [36] containing requests from hundreds of applications, and *(ii)* block traces from Microsoft Research [48]. Neither trace is ideal, but together we believe they represent a wide range of relevant behaviors. Memcachier provides caching-as-a-service and serves objects from a few bytes to 1 MB (median: 100 B); this variability is a common feature of key-value caches [5, 22]. However, many of its customers massively overprovision resources, forcing us to consider scaled-down cache sizes to replicate miss ratios seen in larger deployments [37]. Fortunately, scaled-down caches are known to be good models of behavior at larger sizes [6, 30, 51]. Meanwhile, the Microsoft Research traces let us study larger objects (median: 32 KB) and cache sizes. However, its object sizes are much less variable, and block trace workloads may differ from key-value workloads.

We evaluate 512 M requests from each trace, ignoring the first 128 M to warm up the cache. For the shorter traces, we replay the trace if it terminates to equalize trace length across results. All included traces are much longer than LHD's reconfiguration interval (see Sec. 5).

Since it is too expensive to compute Eq. 2 for every object on each eviction, evictions instead sample 64 random objects, as described in Sec. 4.1. LHD monitors hit and eviction distributions online and, to escape local optima, devotes a small amount of space (1%) to "explorer" objects that are not evicted until a very large age.

What is the best LHD configuration?: LHD uses an object's age to predict its hit density. We also consider two additional object features to improve LHD's predictions: an object's last hit age and its app id. LHD_{APP} classifies objects by hashing their app id into one of N classes (mapping several apps into each class limits overheads). We only use LHD_{APP} on the Memcachier trace, since the block traces lack app ids. $\text{LHD}_{\text{LAST HIT}}$ classifies objects by the age of their last hit, analogous to LRU-K [38], broken into N classes spaced at powers of 2 up to the maximum age. (E.g., with max age = 64 K and $N = 4$, classes are given by last hit age in $0 < 16\,\text{K} < 32\,\text{K} < 64\,\text{K} < \infty$).

We swept configurations over the Memcachier and Microsoft traces and found that both app and last-hit classification reduce misses. Furthermore, these improvements come with relatively few classes, after which classification yields diminishing returns. Based on these results, we configure LHD to classify by last hit age (16 classes) and application id (16 classes). We refer to this configuration as LHD+ for the remainder of the paper.

How does LHD+ compare with other policies?: Fig. 4 shows the miss ratio across many cache sizes for LHD+, LRU, and three prior policies: GDSF [4, 16], AdaptSize [9], and Hyperbolic [11]. GDSF and Hyperbolic use different ranking functions based on object recency, frequency, and size (e.g., Eq. 1). AdaptSize probabilistically admits objects to an LRU cache to avoid polluting the cache with large objects (Sec. 6). LHD+ achieves the best miss ratio across all cache sizes, outperforms LRU by a large margin, and outperforms Hyperbolic, GDSF, and AdaptSize, which perform differently across different traces. No prior policy is consistently close to LHD+'s hit ratio.

Moreover, Fig. 4 shows that LHD+ needs less space than these other policies to achieve the same miss ratio, sometimes substantially less. For example, on Memcachier, a 512 MB LHD+ cache matches the hit rate of a 768 MB Hyperbolic cache, a 1 GB GDSF, or a 1 GB AdaptSize cache, and LRU does not match the performance even with 2 GB. In other words, LRU requires more than 4× as many servers to match LHD+'s hit rate.

Averaged across all sizes, LHD+ incurs 45% fewer misses than LRU, 27% fewer than Hyperbolic and GDSF and 23% fewer than AdaptSize. Moreover, at the largest

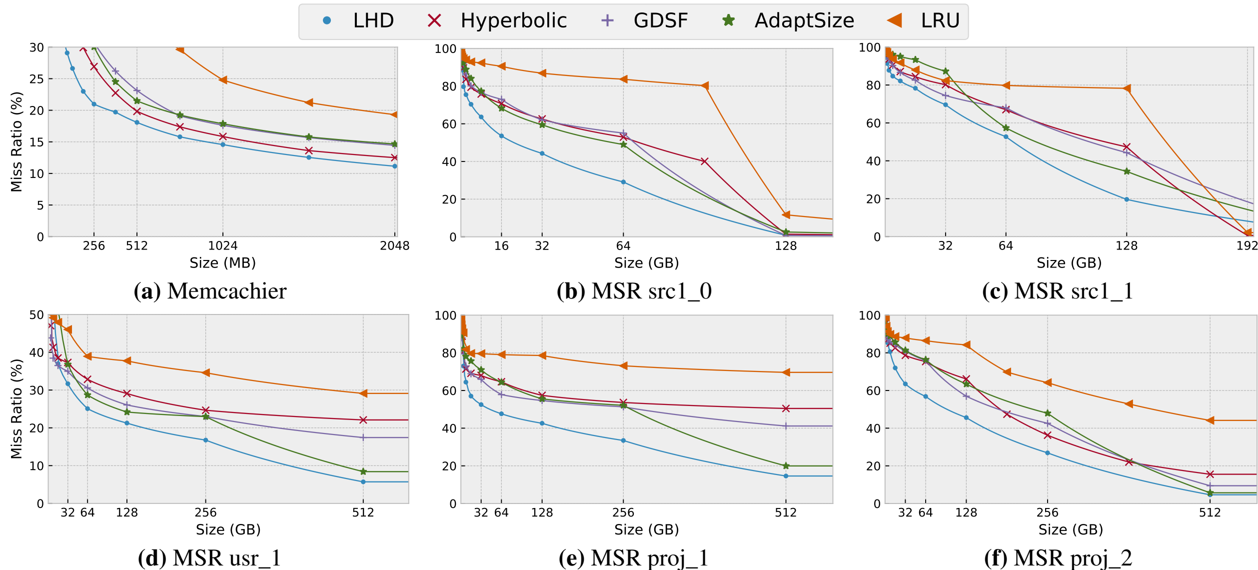


Figure 4: Miss ratio for LHD+ vs. prior policies over 512 M requests and cache sizes from 2 MB to 2 GB on Memcachier trace and from 128 MB to 512 GB on MSR traces. LHD+ consistently outperforms prior policies on all traces.

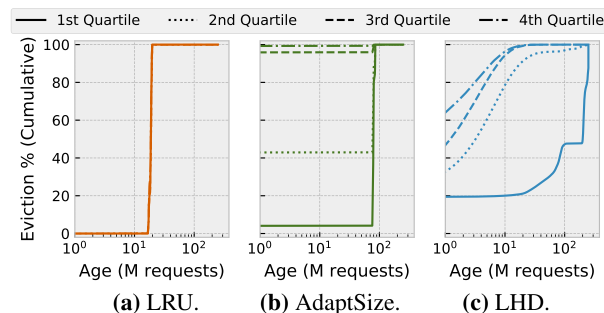


Figure 5: When policies evict objects, broken into quartiles by object size. LRU evicts all objects at roughly the same age, regardless of their size, wasting space on big objects. AdaptSize bypasses most large objects, losing some hits on these objects, while also ignoring object size after admission, still wasting space. LHD dynamically ranks objects to evict larger objects sooner, allocating space across all objects to maximize hits.

sizes, LHD+ incurs very few non-compulsory misses, showing it close to exhausting all possible hits.

Where do LHD+'s benefits come from?: LHD+'s dynamic ranking gives it the flexibility to evict the least valuable objects, without the restrictions or built-in assumptions of prior policies. To illustrate this, Fig. 5 compares when LRU, AdaptSize, and LHD evict objects on the Memcachier trace at 512 MB. Each line in the figure shows the cumulative distribution of eviction age for objects of different sizes; e.g., the solid line in each figure shows when the smallest quartile of objects are evicted.

LRU ignores object size and evicts all objects at roughly the same age. Because of this, LRU wastes space on large objects and must evict objects when they are relatively young (age≈30 M), hurting its hit ratio. AdaptSize improves on LRU by bypassing most large objects so that admitted objects survive longer (age≈75 M). This lets AdaptSize get more hits than LRU, at the cost of forgoing some hits to the bypassed objects. However, since AdaptSize evicts LRU after admission, it still wastes space on large, admitted objects.

LHD+ is not limited in this way. It can admit all objects and evict larger objects sooner. This earns LHD+ more hits on large objects than AdaptSize, since they are not bypassed, and lets small objects survive longer than AdaptSize (age≈200 M), getting even more hits.

Finally, although many applications are recency-friendly, several applications in the Memcachier trace as well as most of the Microsoft Research traces show that this is not true in general. As a result, policies that include recency (i.e., nearly all policies, including GDSF, Hyperbolic, and AdaptSize), suffer from pathologies like performance cliffs [6, 18]. For example, LRU, GDSF, and Hyperbolic suffer a cliff in src1_0 at 96 MB and proj_2 at 128 MB. LHD avoids these cliffs and achieves the highest performance of all policies (see Sec. 6).

4 RankCache Design

LHD improves hit rates, but implementability and request throughput also matter in practice. We design RankCache to efficiently support arbitrary ranking functions, including hit density (Eq. 5). The challenge is that, with arbitrary ranking functions, the rank-order of objects can change constantly. A naïve implementation would scan all cached objects to find the best victim for each eviction, but this is far too expensive. Alternatively, for some restricted ranking functions, prior work has used priority queues (i.e., min-heaps), but these queues require expensive global synchronization to keep the data

structure consistent [9].

RankCache solves these problems by *approximating* a global ranking, avoiding any synchronization in the common case. RankCache does not require synchronization even for evictions, unlike prior high-performance caching systems [22, 34], letting it achieve high request throughput with non-negligible miss rates.

4.1 Lifetime of an eviction in LHD

Ranks in LHD constantly change, and this dynamism is critical for LHD, since it is how LHD adapts its policy to the access pattern. However, it would be very expensive to compute Eq. 5 for all objects on every cache miss. Instead, two key techniques make LHD practical: *(i)* precomputation and *(ii)* sampling. Fig. 6 shows the steps of an eviction in RankCache, discussed below.

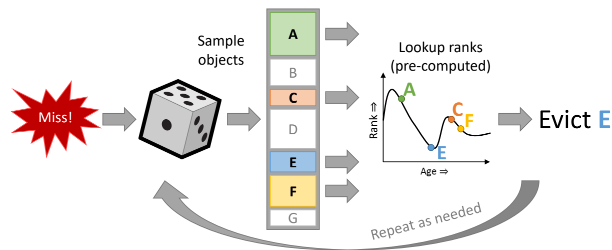


Figure 6: Steps for an eviction in RankCache. First, randomly sample objects, then lookup their precomputed rank and evict the object with the worst rank.

Selecting a victim: RankCache randomly samples cached objects and evicts the object with the worst rank (i.e., lowest hit density) in the sample. With a large enough sample, the evicted object will have eviction priority close to the global maximum, approximating a perfect ranking. Sampling is an old idea in processor caches [44, 46], has been previously proposed for web proxies [39], and is used in some key-value caches [1, 11, 19]. Sampling is effective because the quality of a random sample depends on its size, *not the size of the underlying population* (i.e., number of cached objects). Sampling therefore lets RankCache implement dynamic ranking functions in constant time with respect to the number of cached objects.

Sampling eliminates synchronization: Sampling makes cache management concurrent. Both linked lists and priority queues have to serialize GET and SET operations to maintain a consistent data structure. For example, in memcached, where LRU is implemented by a linked list, every cache hit promotes the hit object to the head of the list. On every eviction, the system first evicts the object from the tail of the list, and then inserts the new object at the head of the list. These operations serialize all GETs and SETs in memcached.

To avoid this problem, systems commonly sacrifice hit ratio: by default, memcached only promotes objects if they are older than one minute; other systems use CLOCK [22] or FIFO [33], which do not require global updates on a cache hit. However, *these policies still serialize all evictions*.

Sampling, on the other hand, allows each item to update its metadata (e.g., reference timestamp) independently on a cache hit, and evictions can happen concurrently as well except when two threads select the same victim. To handle these rare races, RankCache uses memcached's built-in versioning and optimistic concurrency: evicting threads sample and compare objects in parallel, then lock the victim and check if its version has changed since sampling. If it has, then the eviction process is restarted. Thus, although sampling takes more operations per eviction, it increases concurrency, letting RankCache achieve higher request throughput than CLOCK/FIFO under high load.

Few samples are needed: Fig. 7 shows the effect of sampling on miss ratio going from associativity (i.e., sample size) of one to 128. With only one sample, the cache randomly replaces objects, and all policies perform the same. As associativity increases, the policies quickly diverge. We include a sampling-based variant of LRU, where an object's rank equals its age. LRU, Hyperbolic, and LHD+ all quickly reach diminishing returns, around associativity of 32. At this point, true LRU and sampling-based LRU achieve identical hit ratios.

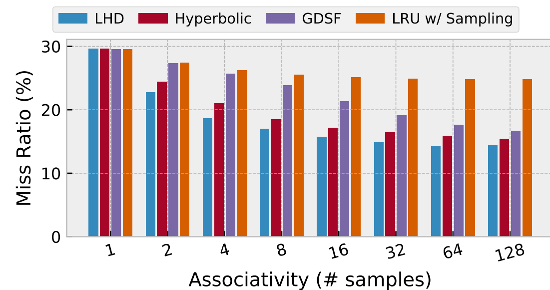


Figure 7: Miss ratios at different associativities.

Since sampling happens at each eviction, lower associativity is highly desirable from a throughput and latency perspective. Therefore, RankCache uses an associativity of 64.

We observe that GDSF is much more sensitive to associativity, since each replacement in GDSF updates global state (L, see Sec. 2.1). In fact, GDSF still has not converged at 128 samples. GDSF's sensitivity to associativity makes it unattractive for key-value caches, since it needs expensive data structures to accurately track its state (Fig. 10). Hyperbolic [11] uses a different ranking function without global state to avoid this problem.

Precomputation: RankCache precomputes object ranks so that, given an object, its rank can be quickly found by indexing a table. In the earlier example, RankCache would precompute Fig. 2c so that ranks can be looked up

directly from an object's age. With LHD, RankCache periodically (e.g., every one million accesses) recomputes its ranks to remain responsive to changes in application behavior. This approach is effective since application behavior is stable over short time periods, changing much more slowly than the ranks themselves fluctuate. Moreover, Eq. 5 can be computed efficiently in linear time [8], and RankCache configures the maximum age to keep overheads low (Sec. 5).

4.2 Approximating global rankings with slabs

RankCache uses slab allocation to manage memory because it ensures that our system achieves predictable $O(1)$ insertion and eviction performance, it does not require a garbage collector, and it has no external fragmentation. However, in slab allocation, each slab class evicts objects independently. Therefore, another design challenge is to approximate a global ranking when each slab allocation implements its own eviction policy.

Similar to memcached, when new objects enter the cache, RankCache evicts the lowest ranked object from the same slab class. RankCache approximates a global ranking of all objects by periodically rebalancing slabs among slab classes. It is well-known that LRU effectively evicts objects once they reach a characteristic age that depends on the cache size and access pattern [15]. This fact has been used to balance slabs across slab classes to approximate global LRU by equalizing eviction age across slab classes [37]. RankCache generalizes this insight, such that caches essentially evict objects once they reach a *characteristic rank*, rather than age, that depends on the cache size and access pattern.

Algorithm: In order to measure the average eviction rank, RankCache records the cumulative rank of evicted objects and the number of evictions. It then periodically moves a slab from the slab class that has the highest average victim rank to that with the lowest victim rank.

However, we found that some slab classes rarely evict objects. Without up-to-date information about their average victim rank, RankCache was unable to rebalance slabs away from them to other slab classes. We solved this problem by performing one "fake eviction" (i.e., sampling and ranking) for each slab class during rebalancing. By also averaging victim ranks across several decisions, this mechanism gives RankCache enough information to effectively approximate a global ranking.

RankCache decides whether it needs to rebalance slabs every 500 K accesses. We find that this is sufficient to converge to the global ranking on our traces, and more frequent rebalancing is undesirable because it has a cost: when a 1 MB slab is moved between slab classes, 1 MB of objects are evicted from the original slab class.

Evaluation: Fig. 8 shows the effect of rebalancing slabs in RankCache. It graphs the distribution of victim rank

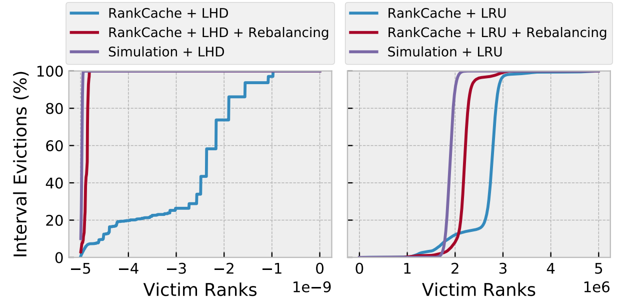


Figure 8: Distribution of victim rank for slab allocation policies with and without rebalancing vs. true global policy. LHD+ is on the left, LRU on the right.

for several different implementations, with each slab class shown in a different color. The right-hand figure shows RankCache with sampling-based LRU, and the left shows RankCache with LHD+. An idealized, global policy has victim rank tightly distributed around a single peak—this demonstrates the accuracy of our characteristic eviction rank model. Without rebalancing, each slab evicts objects around a different victim rank, and is far from the global policy. With rebalancing, the victim ranks are much more tightly distributed, and we find this is sufficient to approximate the global policy.

5 RankCache Implementation

We implemented RankCache, including its LHD ranking function, on top of memcached [23]. RankCache is backwards compatible with the memcached protocol and is a fairly lightweight change to memcached v1.4.33.

The key insight behind RankCache's efficient implementation is that, by design, RankCache is an approximate scheme (Sec. 4). We can therefore tolerate loosely synchronized events and approximate aging information. Moreover, RankCache does not modify memcached's memory allocator, so it leverages existing functionality for events that require careful synchronization (e.g., moving slabs).

Aging: RankCache tracks time through the total number of accesses to the cache. Ages are *coarsened* in large increments of *COARSENESS* accesses, up to a *MAX_AGE*. *COARSENESS* and *MAX_AGE* are chosen to stay within a specified error tolerance (see appendix); in practice, coarsening introduces no detectable change in miss ratio or throughput for reasonable error tolerances (e.g., 1%).

Conceptually, there is a global timestamp, but for performance we implement distributed, fuzzy counters. Each server thread maintains a thread-local access count, and atomic-increments the global timestamp periodically when its local counter reaches *COARSENESS*.

RankCache must track the age of objects to compute their rank, which it does by adding a 4 B timestamp to the object metadata. During ranking, RankCache computes an object's coarsened age by subtracting the object

timestamp from the global timestamp.

Ranking: RankCache adds tables to store the ranks of different objects. It stores ranks up to *MAX_AGE* per class, each rank a 4 B floating-point value. With 256 classes (Sec. 3), this is 6.4 MB total overhead. Ranks require no synchronization, since they are read-only between reconfigurations, and have a single writer (see below). We tolerate races as they are infrequently updated.

Monitoring behavior: RankCache monitors the distribution of hit and eviction age by maintaining histograms of hits and evictions. RankCache increments the appropriate counter upon each access, depending on whether it was a hit or eviction and the object's coarsened age. To reduce synchronization, these are also implemented as distributed, fuzzy counters, and are collected by the updating thread (see below). Counters are 4 B values; with 256 classes, hit and eviction counters together require 12.6 MB per thread.

Sampling: Upon each eviction, RankCache samples objects from within the same slab class by randomly generating indices and then computing the offset into the appropriate slab. Because objects are stored at regular offsets within each slab, this is inexpensive.

Efficient evictions: For workloads with non-negligible miss ratios, evictions are the rate-limiting step in RankCache. To make evictions efficient, RankCache uses two optimizations. First, rather than adding an object to a slab class's free list and then immediately claiming it, RankCache directly allocates the object within the same thread after it has been freed. This avoids unnecessary synchronization.

Second, RankCache places object metadata in a separate, contiguous memory region, called the *tags*. Tags are stored in the same order as objects in the slab class, making it easy to find an object from its metadata. Since slabs themselves are stored non-contiguously in memory, each object keeps a back pointer into the tags to find its metadata. Tags significantly improve spatial locality during evictions. Since sampling is random by design, without separate tags, RankCache suffers 64 (associativity) cache misses per eviction. Compact tags allow RankCache to sample 64 candidates with just 4 cache misses, a 16× improvement in locality.

Background tasks: Both updating ranks and rebalancing slabs are off the critical path of requests. They run as low-priority background threads and complete in a few milliseconds. Periodically (default: every 1 M accesses), RankCache aggregates histograms from each thread and recomputes ranks. First, RankCache averages histograms with prior values, using an exponential decay factor (default: 0.9). Then it computes LHD for each class in linear time, requiring two passes over the ages using an algorithm similar to [8]. Also periodically (every 500 K accesses), RankCache rebalances one slab from the slab with the highest eviction rank to the one with the lowest, as described in Sec. 4.2.

Across several orders of magnitude, the reconfiguration interval and exponential decay factor have minimal impact on hit rate. On the Memcachier trace, LHD+'s non-compulsory miss rate changes by 1% going from reconfiguring every 10 K to 10 M accesses, and the exponential decay factor shows even smaller impact when it is set between 0.1 and 0.99.

5.1 RankCache matches simulation

Going left-to-right, Fig. 9 compares the miss ratio over 512 M accesses on Memcachier at 1 GB for *(i)* stock memcached using true LRU within each slab class; RankCache using sampling-based LRU as its ranking function *(ii)* with and *(iii)* without rebalancing; RankCache using LHD+ *(iv)* with and *(v)* without rebalancing; and *(vi)* an idealized simulation of LHD+ with global ranking.

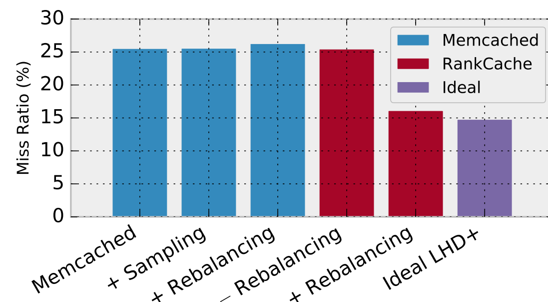


Figure 9: RankCache vs. unmodified memcached and idealized simulation. Rebalancing is necessary to improve miss ratio, and effectively approximates a global ranking.

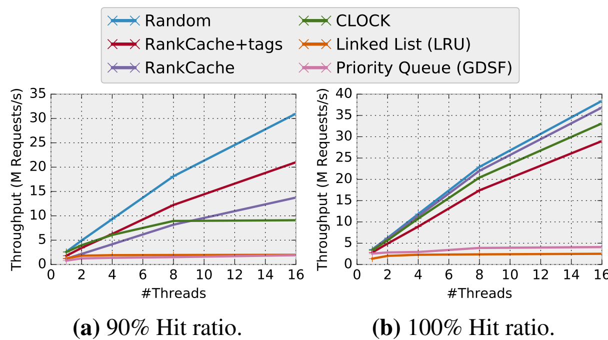
As the figure shows, RankCache with slab rebalancing closely matches the miss ratio of the idealized simulation, but without slab rebalancing it barely outperforms LRU. This is because LHD+ operating independently on each slab cannot effectively take into account object size, and hence on an LRU-friendly pattern performs similarly to LRU. The small degradation in hit ratio vs. idealized simulation is due to forced, random evictions during slab rebalancing.

5.2 RankCache with LHD+ achieves both high hit ratio and high performance

Methodology: To evaluate RankCache's performance, we stress request serving within RankCache itself by conducting experiments within a single server and bypassing the network. Each server thread pulls requests off a thread-local request list. We force all objects to have the same size to maximally stress synchronization in each policy. Prior work has explored techniques to optimize the network in key-value stores [22, 33, 34]; these topics are not our contribution.

We compare RankCache against list-based LRU, GDSF using a priority queue (min-heap), and CLOCK. These cover the main implementation primitives used in key-value caches (Sec. 2). We also compare against random evictions to show peak request throughput when the eviction policy does no work and maintains no state. (Random pays for its throughput by suffering many misses.)

Scalability: Fig. 10 plots the aggregate request throughput vs. number of server threads on a randomly generated trace with Zipfian object popularities. We present throughput at 90% and 100% hit ratio; the former represents a realistic deployment, the latter peak performance.



(a) 90% Hit ratio. **(b)** 100% Hit ratio.

Figure 10: RankCache's request throughput vs. server threads. RankCache's performance approaches that of random, and outperforms CLOCK with non-negligible miss ratio.

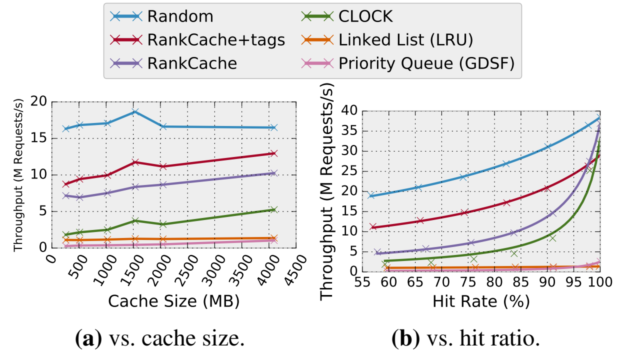
RankCache scales nearly as well as random because sampling avoids nearly all synchronization, whereas LRU and GDSF barely scale because they serialize all operations. Similarly, CLOCK performs well at 100% hit ratio, but serializes evictions and underperforms RankCache with 10% miss ratio. Finally, using separate tags in RankCache lowers throughput with a 100% hit ratio, but improves performance even with a 10% miss ratio.

Trading off throughput and hit ratio: Fig. 11a plots request throughput vs. cache size for these policies on the Memcachier trace. RankCache achieves the highest request throughput of all policies except random, and tags increase throughput at every cache size. RankCache increases throughput because *(i)* it eliminates nearly all synchronization and *(ii)* LHD+ achieves higher hit ratio than other policies, avoiding time-consuming evictions.

Fig. 11b helps explain these results by plotting request throughput vs. hit ratio for the different systems. These numbers are gathered by sweeping cache size for each policy on a uniform random trace, equalizing hit ratio across policies at each cache size. Experimental results are shown as points, and we fit a curve to each dataset by assuming that:

$$\text{Total service time} = \text{\# GETs} \times \text{GET time} + \text{\# SETs} \times \text{SET time}$$

As Fig. 11b shows, this simple model is a good fit, and thus GET and SET time are independent of cache size.



(a) vs. cache size. **(b)** vs. hit ratio.

Figure 11: Request throughput on Memcachier trace at 16 server threads. RankCache with LHD achieves the highest request throughput of all implementations, because it reduces synchronization and achieves a higher hit ratio than other policies. Tags are beneficial except at very high hit ratios.

Fig. 11b shows how important hit ratio is, as small improvements in hit ratio yield large gains in request throughput. This effect is especially apparent on CLOCK because it synchronizes on evictions, but not on hits. Unfortunately, CLOCK achieves the lowest hit ratio of all policies, and its throughput suffers as a result. In constrast, LHD+ pushes performance higher by improving hit ratio, and RankCache removes synchronization to achieve the best scaling of all implementations.

Response latency: Fig. 12 shows the average response time of GETs and SETs with different policies running at 1 and 16 server threads, obtained using the same procedure as Fig. 11b. The 16-thread results show that, in a parallel setting, RankCache achieves the lowest per-operation latency of all policies (excluding random), and in particular using separate tags greatly reduces eviction time. While list- or heap-based policies are faster in a sequential setting, RankCache's lack of synchronization dominates with concurrent requests. Because CLOCK synchronizes on evictions, its evictions are slow at 16 threads, explaining its sensitivity to hit ratio in Fig. 11b. RankCache reduces GET time by 5× vs. list and prio-queue, and SET time by 5× over CLOCK.

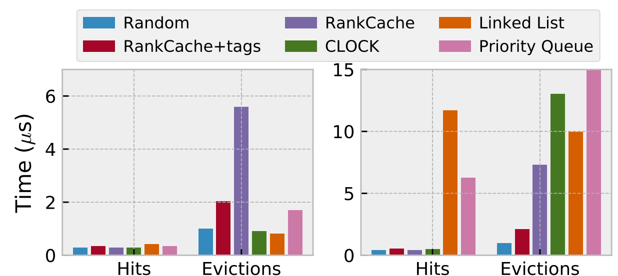


Figure 12: Request processing time for hits and evictions at a single thread (left) and 16 threads (right).

In a real-world deployment, RankCache's combination of high hit ratio and low response latency would yield greatly reduced mean and tail latencies and thus

to significantly improved end-to-end response latency.

6 Related Work

Prior work in probabilistic eviction policies: EVA, a recent eviction policy for processor caches [7, 8], introduced the idea of using conditional probability to balance hits vs. resources consumed. There are several significant differences between LHD and EVA that allow LHD to perform well on key-value workloads.

First, LHD and EVA use different ranking functions. EVA ranks objects by their net contribution measured in hits, not by hit density. This matters, because EVA's ranking function does not converge in key-value cache workloads and performs markedly worse than LHD. Second, unlike processor caches, LHD has to deal with variable object sizes. Object size is one of the most important characteristics in a key-value eviction policy. RankCache must also rebalance memory across slab classes to implement a global ranking. Third, LHD classifies objects more aggressively than is possible with the implementation constraints of hardware policies, and classifies by last hit age instead of frequency, which significantly improves hit ratio.

Key-value caches: Several systems have tried to improve upon memcached's poor hit ratio under objects of varying sizes. Cliffhanger [18] uses shadow queues to incrementally assign memory to slab classes that would gain the highest hit ratio benefit. Similarly, Dynacache [17], Moirai [49], Mimir [43] and Blaze [10] determine the appropriate resource allocation for objects of different sizes by keeping track of LRU's stack distances. Twitter [41] and Facebook [37] periodically move memory from slabs with a high hit ratio to those with a low hit ratio. Other systems have taken a different approach to memory allocation than memcached. Memshare [19] and MICA [34] utilize log-structured memory allocation. In the case of all the systems mentioned above, the memory allocation is intertwined with their eviction policy (LRU).

Similar to RankCache, Hyperbolic caching [11] also uses sampling to implement dynamic ranking functions. However, as we have demonstrated, Hyperbolic suffers from higher miss ratios, since it is a recency-based policy that is susceptible to performance cliffs, and Hyperbolic did not explore concurrent implementations of sampling as we have done in RankCache.

Replacement policies: Prior work improves upon LRU by incorporating more information about objects to make better decisions. For example, many policies favor objects that have been referenced frequently in the past, since intuitively these are likely to be referenced again soon. Prominent examples include LRU-K [38], SLRU [29], 2Q [28], LRFU [31], LIRS [26], and ARC [35]. There is also extensive prior work on replacement policies for objects of varying sizes. LRU-MIN [2], HYBRID [52], GreedyDual-Size (GDS) [14], GreedyDual-Size-Frequency (GDSF) [4, 16], LNC-R-W3 [45], AdaptSize [9], and Hyperbolic [11] all take into account the size of the object.

AdaptSize [9] emphasizes object admission vs. eviction, but this distinction is only important for list-based policies, so long as objects are small relative to the cache's size. Ranking functions (e.g., GDSF and LHD) can evict low-value objects immediately, so it makes little difference if they are admitted or not (Fig. 5).

Several recent policies explicitly avoid cliffs seen in LRU and other policies [6, 11, 18]. Cliffs arise when policies' built-in assumptions are violated and the policy behaves pathologically, so that hit ratios do not improve until all objects fit in the cache. LHD also avoids cliffs, but does so by avoiding pathological behavior in the first place. Cliff-avoiding policies achieve hit ratios along the cliff's convex hull, and no better [6]; LHD matches or exceeds this performance on our traces.

Tuning eviction policies: Many prior policies require application-specific tuning. For example, SLRU divides the cache into S partitions. However, the optimal choice of S, as well as how much memory to allocate to each partition, varies widely depending on the application [24, 50]. Most other policies use weights that must be tuned to the access pattern (e.g., [2, 11, 27, 38, 45, 52]). For example, GD* adds an exponential parameter to Eq. 1 to capture burstiness [27], and LNC-R-W3 has separate weights for frequency and size [45]. In contrast to LHD, these policies are highly sensitive to their parameters. (We have implemented LNC-R-W3, but found it performs worse than LRU without extensive tuning at each size, and so do not present its results.)

7 Conclusions

This paper demonstrates that there is a large opportunity to improve cache performance through non-heuristic approach to eviction policies. Key-value caches are an essential layer for cloud applications. Scaling the capacity of LRU-based caches is an unsustainable approach to scale their performance. We have presented a practical and principled approach to tackle this problem, which allows applications to achieve their performance goals at significantly lower cost.

Acknowledgements

We thank our anonymous reviewers, and especially our shepherd, Jon Howell, for their insightful comments. We also thank Amit Levy and David Terei for supplying the Memcachier traces, and Daniel Berger for his feedback and help with implementing AdaptSize. This work was funded by a Google Faculty Research Award and supported by the Parallel Data Lab at CMU.

References

[1] Redis. http://redis.io/. 7/24/2015.

[2] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching proxies: Limitations and potentials. Technical report, Blacksburg, VA, USA, 1995.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the Fourth International Conference on on Parallel and Distributed Information Systems*, DIS '96, pages 92–107, Washington, DC, USA, 1996. IEEE Computer Society.

[4] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review*, 2000.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[6] N. Beckmann and D. Sanchez. Talus: A simple way to remove cliffs in cache performance. In *HPCA-21*, 2015.

[7] N. Beckmann and D. Sanchez. Modeling cache performance beyond LRU. *HPCA-22*, 2016.

[8] N. Beckmann and D. Sanchez. Maximizing cache performance under uncertainty. *HPCA-23*, 2017.

[9] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 483–498, Boston, MA, 2017. USENIX Association.

[10] H. Bjornsson, G. Chockler, T. Saemundsson, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 59. ACM, 2013.

[11] A. Blankstein, S. Sen, and M. J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, Santa Clara, CA, 2017. USENIX Association.

[12] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995.

[13] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134. IEEE, 1999.

[14] P. Cao and S. Irani. Cost-aware www proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, USITS'97, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[15] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 2002.

[16] L. Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[17] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association.

[18] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, Mar. 2016. USENIX Association.

[19] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, 2017. USENIX Association.

[20] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW client-based traces. Technical report, Boston, MA, USA, 1995.

[21] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2), 2013.

[22] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 371–384, Berkeley, CA, USA, 2013. USENIX Association.

[23] B. Fitzpatrick. Distributed caching with Memcached. *Linux journal*, 2004(124):5, 2004.

[24] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, 2013. ACM.

[25] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction. In *ISCA-37*, 2010.

[26] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002.

[27] S. Jin and A. Bestavros. GreedyDualâĹŮ web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24(2):174–183, 2001.

[28] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[29] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, Mar. 1994.

[30] R. E. Kessler, M. D. Hill, and D. A. Wood. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Transactions on Computers*, 1994.

[31] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. *SIGMETRICS Perform. Eval. Rev.*, 27(1):134–143, May 1999.

[32] C. Li and A. L. Cox. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, page 5. ACM, 2015.

[33] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 476–488, New York, NY, USA, 2015. ACM.

[34] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association.

[35] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[36] Memcachier. www.memcachier.com.

[37] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[38] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, pages 297–306, New York, NY, USA, 1993. ACM.

[39] K. Psounis and B. Prabhakar. A randomized web-cache replacement scheme. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1407–1415. IEEE, 2001.

[40] M. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA-34*, 2007.

[41] M. Rajashekhar and Y. Yue. Twemcache. blog.twitter.com/2012/caching-with-twemcache.

[42] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *FAST*, pages 1–16, 2014.

[43] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[44] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *MICRO-43*, 2010.

[45] P. Scheuermann, J. Shim, and R. Vingralek. A case for delay-conscious caching of web documents. *Computer Networks and ISDN Systems*, 29(8):997–1005, 1997.

[46] A. Seznec. A case for two-way skewed-associative caches. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 169–178. ACM, 1993.

[47] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, Feb. 1985.

[48] SNIA. MSR Cambridge Traces. http://iotta.snia.org/traces/388, 2008.

[49] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 174–181. ACM, 2015.

[50] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, Santa Clara, CA, Feb. 2015. USENIX Association.

[51] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, 2017. USENIX Association.

[52] R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. In *Selected Papers from the Sixth International Conference on World Wide Web*, pages 977–986, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[53] N. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.

A Age coarsening with bounded error

RankCache chooses how much to coarsen ages and how many ages to track in order to stay within a user-specified error tolerance. RankCache is very conservative, so that in practice much more age coarsening and fewer ages can be used with no perceptible loss in hit rate.

Choosing a maximum age: The effect of age coarsening is to divide ages into equivalence classes in chunks of *COARSENESS*, so that the maximum true age that can be tracked is *COARSENESS* $\times$ *MAX_AGE*. Any events above this maximum true age cannot be tracked. Hence, if the access pattern is a scan at a larger reuse distance than this, the cache will be unable to find these objects, even with an optimal ranking metric.

If the cache fits N objects and the scan contains M objects, then the maximum hit rate on the trace is N/M. To keep the error tolerance below ϵ, we must track ages up to $M \geq N/\epsilon$, hence:

$$\textit{MAX_AGE} \geq \frac{N}{\textit{COARSENESS} \times \epsilon} \tag{7}$$

Choosing age coarsening: *COARSENESS* hurts performance by forcing RankCache to be conservative and keep objects around longer than necessary, until RankCache is certain that they can be safely evicted. The effect of large *COARSENESS* is to reduce effective cache capacity, since more space is spent on objects that will be eventually evicted. In the worst case, all evicted objects spend an additional *COARSENESS* accesses in the cache, reducing the space available for hits proportionally.

Coarsening thus "pushes RankCache down the hit rate curve". The lost hit rate is maximized when the hit rate curve has maximum slope. Since optimal eviction policies have concave hit rate curves [6], the loss from coarsening is maximized when the hit rate curve is a straight line. Once again, this is the hit rate curve of a scanning pattern with uniform object size.

Without loss of generality, assume objects have size $= 1$. The cache size equals the sum of the expected resources spent on hits and evictions [8],

$$N = \mathbb{E}[H] + \mathbb{E}[E]$$

In the worst case, coarsening increases space spent on evictions by

$$\mathbb{E}[E'] = \mathbb{E}[E] + \textit{COARSENING},$$

so space for hits is reduced

$$\mathbb{E}[H'] = \mathbb{E}[H] - \textit{COARSENING}$$

With a scan over M objects, the effect of coarsening is thus to reduce cache hit rate by

$$\text{Hit rate loss} = \frac{\textit{COARSENING}}{M}$$

This loss is maximized when M is small, but M cannot be too small since $M \leq N$ leads to zero misses.

To bound this error below ϵ, RankCache coarsens ages such that

$$\textit{COARSENING} \leq N \times \epsilon \tag{8}$$

Substituting into Eq. 7 yields

$$\textit{MAX_AGE} \geq \frac{1}{\epsilon^2} \tag{9}$$

Implementation: Age coarsening thus depends only on the error tolerance and number of cached objects. RankCache monitors the number of cached objects and, every 100 intervals, updates *COARSENING* and *MAX_AGE*. We find that hit rate is insensitive to these parameters, so long as they are within the right order of magnitude.

Performance analysis of cloud applications

Dan Ardelean
Google

Amer Diwan
Google

Chandra Erdman
Google

Abstract

Many popular cloud applications are large-scale distributed systems with each request involving tens to thousands of RPCs and large code bases. Because of their scale, performance optimizations without actionable supporting data are likely to be ineffective: they will add complexity to an already complex system often without chance of a benefit. This paper describes the challenges in collecting actionable data for Gmail, a service with more than 1 billion active accounts.

Using production data from Gmail we show that both the load and the nature of the load changes continuously. This makes Gmail performance difficult to model with a synthetic test and difficult to analyze in production. We describe two techniques for collecting actionable data from a production system. First, coordinated bursty tracing allows us to capture bursts of events across all layers of our stack simultaneously. Second, vertical context injection enables us combine high-level events with low-level events in a holistic trace without requiring us to explicitly propagate this information across our software stack.

1 Introduction

Large cloud applications, such as Facebook and Gmail, serve over a billion active users [4, 5]. Performance of these applications is critical: their latency affects user satisfaction and engagement with the application [2, 26] and their resource usage determines the cost of running the application. This paper shows that understanding and improving their resource usage and latency is difficult because their continuously varying load continually changes the performance characteristics of these applications.

Prior work has shown that as the user base changes, the load on cloud applications (often measured as queries-per-second or QPS) also changes [28]. We show, using data from Gmail, that the biggest challenges in analyzing performance come not from changing QPS but in changing load *mixture*. Specifically, we show that the load on a cloud application is a continually varying mixture of diverse loads, some generated by users and some generated by the system itself (e.g., essential maintenance tasks). Even if we consider only load generated by users, there is significant variation in load generated by different users; e.g., some user mailboxes are four orders of magnitude larger than others and operations on larger mailboxes are fundamentally more expensive than those on smaller mailboxes.

This time-varying mixture of load on our system has two implications for performance analysis. First, to determine the effect of a code change on performance, we must collect and analyze data from many points in time and thus many different mixtures of load; any single point in time gives us data only for one mixture of load. Second, to reproduce a performance problem we may need to reproduce the combination of load that led to the performance problem in the first place. While sometimes we can do this in a synthetic test, often times we need to collect and analyze data from a system serving real users.

Doing experiments in a system serving real users is challenging for two reasons. First, since we do not control the load that real users generate, we need to do each experiment in a system serving a large (tens of millions users) random sample of users to get statistically significant results. Second, since experiments in a system serving real users is inherently risky (a mistake can negatively impact users) we use statistics whenever possible to predict the likely outcome of an experiment before actually undertaking the experiment. We show that this is not without its pitfalls: sometimes the distribution of the data (and thus the appropriate statistical method) is not obvious.

To collect rich performance data from a system serving real users we have developed two techniques.

First, *coordinated bursty tracing* collects coordinated

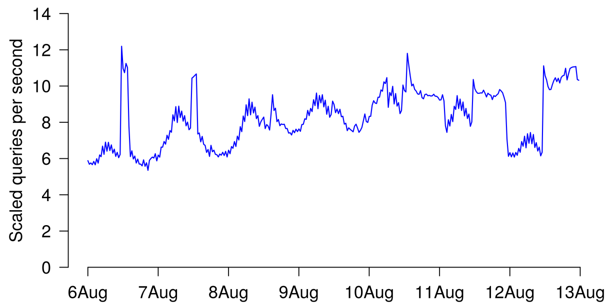


Figure 1: Continuously changing queries per second

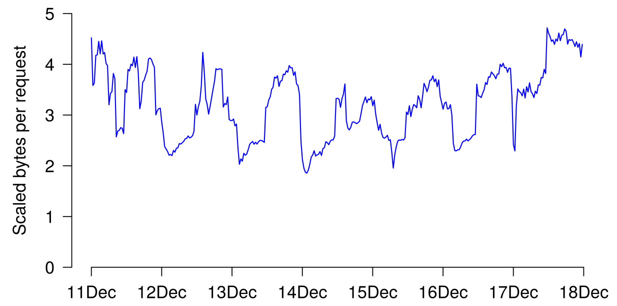


Figure 2: Continuously changing response size

bursts of traces across all software layers without requiring explicit coordination. Unlike traditional sampling or bursty approaches which rely on explicitly maintained counters [19, 6] or propagation of sampling decisions [27, 22], coordinated bursty tracing uses time to coordinate the start and end of bursts. Since all layers collect their bursts at the same time (clock drift has not been a problem in practice), we can reason across the entire stack of our application rather than just a single layer. By collecting many bursts we get a random sampling of the mix of operations which enables us to derive valid conclusions from our performance investigations.

Second, since interactions between software layers are responsible for many performance problems, we need to be able to connect trace events at one layer with events at another. *Vertical context injection* solves this problem by making a stylized sequence of innocuous system calls at each high-level event of interest. These system calls insert the system call events into the kernel trace which we can analyze to produce a trace that interleaves both high and low-level events. Unlike prior work (e.g., [27] or [15]) our approach does not require explicit propagation of a trace context through the layers of the software stack.

To illustrate the above points, this paper presents data from Gmail, a popular email application from Google. However, the authors have used the techniques for many other applications at Google, particularly Google Drive; the lessons in this paper apply equally well to those other applications.

2 Our challenge: constantly varying load

The primary challenge in performance analysis of cloud applications stems from their constantly varying load. Figure 1 shows *scaled* queries per second (QPS) across thousands of processes serving tens of millions of users over the course of a week for one deployment of Gmail. By "scaled" we mean that we have multiplied the actual numbers by a constant to protect Google's proprietary information; since we have multiplied each point by the

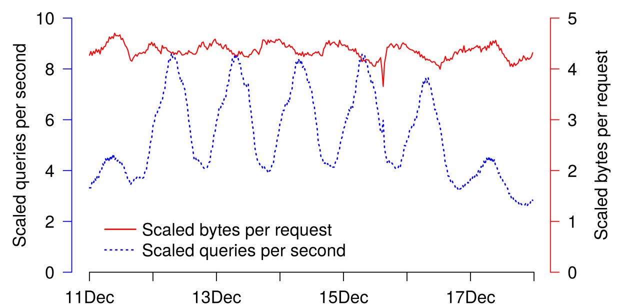


Figure 3: Continuously changing user behavior

same constant and each graph is zero based, it allows relative comparisons between points or curves on the same graph.[1] The time axes for all graphs in this paper are in US Pacific time and start on a Sunday unless the graph is for a one-off event in which case we pick the time axis most suitable for the event. We see that load on our system changes continuously: from day to day and from hour to hour by more than a factor of two.

While one expects fluctuations in QPS (e.g., there are more active users during the day than at night), one does not expect the *mix* of requests to fluctuate significantly. Figure 2 shows one characteristic of requests, the response size per request, over the course of a week. Figure 2 shows that response size per request changes over the course of the week and from hour to hour (by more than a factor of two) which indicates that the actual mix of requests to our system (and not just their count) changes continuously.

The remainder of this section explores the sources of variation in the mix of requests.

2.1 Variations in rate and mix of user visible requests (UVR)

Figure 3 gives the response size per request (upper curve) and QPS (lower curve) for "user visible requests" (UVR for short) only. By UVR we mean requests that users explicitly generate (e.g., by clicking on a message), background requests that the user's client generates (e.g., background sync by the IMAP client), and message delivery.

From Figure 3 we see that, unlike Figure 1, the QPS curve for UVR exhibits an obvious diurnal cycle; during early morning (when both North America and Europe are active) we experience the highest QPS, and the QPS gradually ramps up and down as expected. Additionally we see higher QPS on weekdays compared to weekends.

We also see that the bytes per response in Figure 3 is flatter than in Figure 2: for UVR, we see that the highest observed response size per request is approximately 20% higher than the lowest observed response size per request. In contrast, when we consider all requests (and not just UVR), the highest response size is more than 100% higher than the lowest (Figure 2).

While a 20% variation is much smaller than a 100% variation, it is still surprising: especially given that the data is aggregated over tens of millions of users, we expect that there would not be much variation in average response size over time. We have uncovered two causes for this variation.

First, the mix of mail delivery (which has a small response size) and user activity (which usually has a larger response size) varies over the course of the day which in turn affects the average response size per request. This occurs because many bulk mail senders send bursts of email at particular times of the day and those times do not necessarily correlate with activity of interactive users.

Second, the mix of user interactive requests and sync requests varies over the course of the day. For example, Figure 4 shows the scaled ratio of UVR requests from a web client to UVR requests from an IMAP client (e.g., from an iOS email application or from Microsoft Outlook). IMAP requests are mostly synchronization requests (i.e., give me all messages in this folder) while web client requests are a mix of interactive requests (i.e., user clicks on a message) and prefetch requests. Thus, IMAP requests tend to have a larger response size compared to interactive requests. We see that this ratio too exhibits a diurnal cycle indicating that over the course of the day the relative usage of different email clients changes (by more than a factor of three), and thus the response size per request also changes. This comes about due to varying email client preferences; for example one user can use a dedicated Gmail application while another can use a generic IMAP-based email application on a mobile device).

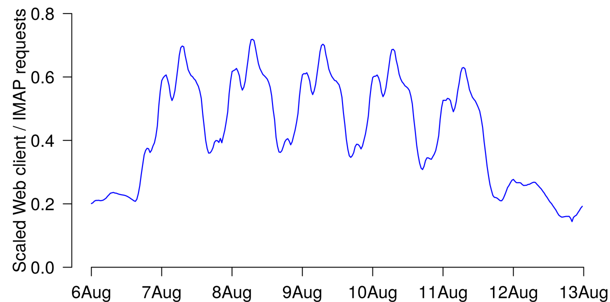


Figure 4: Scaled fraction of web client requests to IMAP requests

In summary, even if we consider only UVR, both the queries per second and mix of requests changes hour to hour and day to day.

2.2 Variations in rate and mix of essential non-UVR work

In addition to UVR requests which directly provide service to our users, our system performs many essential tasks:

- Continuous validation of data. Because of the scale of our system, any kind of failure that *could* happen *does* actually happen. Besides software failures, we regularly encounter hardware errors (e.g., due to an undetected problem with a memory chip). To prevent these failures from causing widespread corruption of our data, we continuously check invariants of the user data. For example, if our index indicates that N messages contain a particular term, we check the messages to ensure that those and only those messages contain the term. As another example, we continuously compare replicas of each user's data to detect any divergence between the replicas.

- Software updates. Our system has many interacting components and each of them has its own update cycle. Coordinating the software updates of all components is not only impractical but undesirable: it is invaluable for each team to be able to update its components when necessary without coordination with other teams. Rather than using dynamic software updating [18] which attempts to keep each process up while updating it, we use a simpler approach: we push software updates by restarting a few processes at a time; our system automatically

[1] All graphs in this paper use such scaling but different graphs may use different scaling to aid readability and thus absolute values are not comparable across graphs.

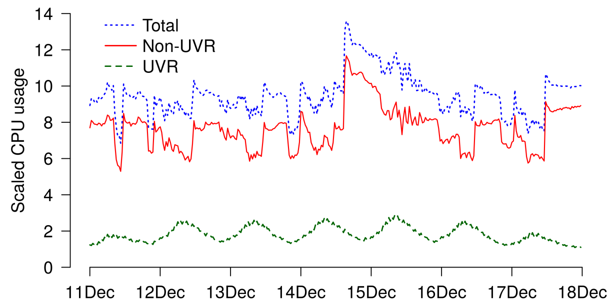


Figure 5: Scaled CPU usage of UVR and non-UVR work

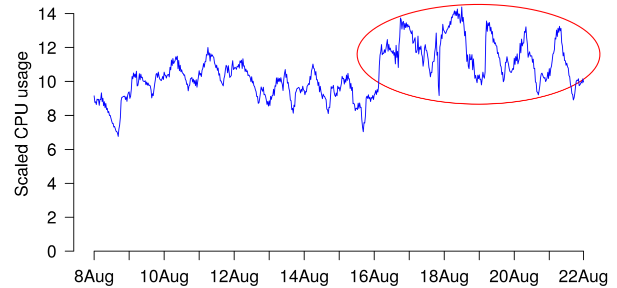


Figure 6: CPU usage goes up globally (circled) after a natural event

moves users away from servers that are being updated to other servers and thus our users get uninterrupted service.

- Repairs. Hardware and software bugs can and do happen. For example, we once had a bug in the message tokenizer that incorrectly tokenized certain email addresses in the message. This bug affected the index and thus users could not search for messages using the affected email addresses. To fix this problem, we had to reindex all affected messages using a corrected tokenizer. Given the scale of our system, such fixes can take weeks or longer to complete and while the fixes are in progress they induce their own load on the system.

- Data management. Gmail uses Bigtable as its underlying storage [11, 10]. Bigtable maintains stacks of key-value pairs and periodically compacts each stack to optimize reads and to remove overwritten values. This process is essential for the resource usage and performance of our system. These compactions occur throughout the day and are roughly proportional to the rate of updates to our system.

As with UVR work, the mix of non-UVR work also changes continuously. When possible we try to schedule non-UVR work when the system is under a low UVR load: this way we not only minimize impact on user-perceived performance but are also able to use resources reserved for UVR that would otherwise go unused (e.g., during periods of low user activity).

Figure 5 shows the scaled CPU usage of UVR work (lowest line), non-UVR work (middle line) and total scaled CPU consumed by the email backend (top line). From this we see that UVR *directly* consumes only about 20% of our CPU; indirectly UVR consumes more CPU because it also induces data management work. Thus, focusing performance analysis on just the UVR or just the non-UVR work alone is inadequate for understanding the performance of our system.

2.3 Variations due to one-off events

In addition to UVR and essential non-UVR work, our system also experiences *one-off events*. We cannot plan for one-off events: they may come from hardware or software outages or from work that must be done right away (e.g., a problem may require us to move the users served by one datacenter into other datacenters).

Figure 6 shows a 20% increase, on average, in CPU usage (circled) of our system after lightning struck Google's Belgian datacenter *four* times and potentially caused corruption in some disks [3]. Gmail dropped all the data in the affected datacenter because it could have been corrupted and automatically reconstructed data from a known uncorrupted source in other datacenters; all of this without the users experiencing any outage or issues with their email. Consequently, during the recovery period after this event, we experienced increased CPU usage in datacenters that were not *directly* affected by the event; this is because they were now serving more users than before the event and because of the work required to reconstruct another copy of the user's data.

One-off events can also interact with optimizations. For example, we wanted to evaluate a new policy for using hedged requests [13] when reading from disk. Our week-long experiment clearly showed that this change was beneficial: it reduced the number of disk operations without degrading latency.

Unfortunately, we found that this optimization interacted poorly with a datacenter maintenance event which temporarily took down a percentage of the disks for a software upgrade. Figure 7 shows the scaled 99th percentile latency of a critical email request with and without our optimization during a maintenance event. We see that during the events, our latency nearly tripled, which of course was not acceptable.

The latency degradation was caused by a bug in our heuristic for deciding whether to redirect a read to another copy of the data: our heuristic was taking too long to blacklist the downed disks and thus rather than redi-

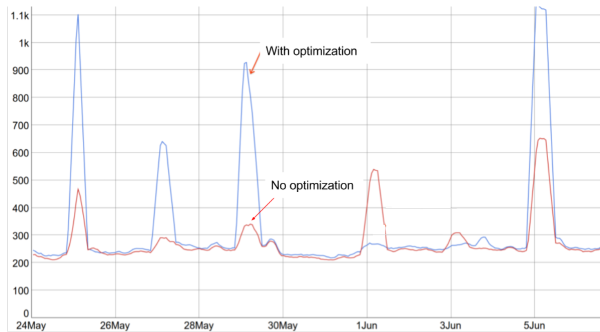


Figure 7: Scaled latency during a datacenter maintenance event

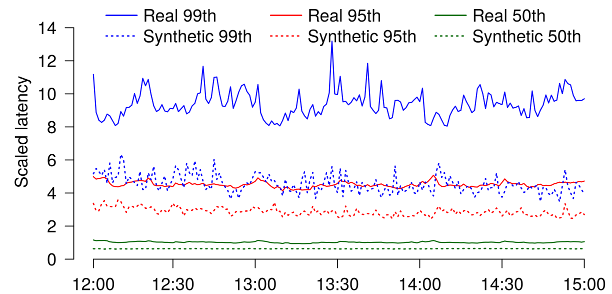


Figure 8: Latency for synthetic users versus real users at the 50th, 95th, and 99th percentiles

recting the read to another copy, it was waiting for the downed disks until the request would time out. Fixing this bug enabled our optimization to save resources without degrading latency even during maintenance events.

In summary, the resource usage during one-off events is often different from resource usage in the stable state. Since one-off events are relatively rare (affect each datacenter every few months) they are easy to miss in our experiments.

2.4 Variation in load makes it difficult to use a synthetic environment

The superimposition of UVR load, non-UVR load, and one-off events results in a complex distribution of latency and resource usage, often with a long-tail. When confronted with such long-tail performance puzzles, we first try to reproduce them with synthetic users: if we can do this successfully, it simplifies the investigation: with synthetic users we can readily add more instrumentation, change the logic of our code, or rerun operations as needed. With real users we are obviously limited because we do not want our experiments to compromise our users' data in any way and because experimenting with real users is much slower than with synthetic users: before we can release a code change to real users it needs to undergo code reviews, testing, and a gradual rollout process.

Gmail has a test environment that brings up the entire software stack and exercises it with synthetic mailboxes and load. We have modeled synthetic users as closely as we can on real users: the mailbox sizes of synthetic users have the same distribution as the mailbox sizes of real users and the requests from synthetic users is based on a model from real users (e.g., same distribution of interarrival times, same mixture of different types of requests, and similar token frequency).

Figure 8 compares the latency of a common email operation for real users to the latency for synthetic users. We collected all data from the same datacenter (different datacenters may use different hardware and thus are not always comparable). The dotted lines in Figure 8 give the latency from the synthetic load and the solid lines display the latency from the real load at various percentiles.

Despite our careful modeling, latency distribution in our test environment is different (and better) than latency distribution of real users. As discussed in Section 2.1, the continuously varying load is difficult to model in any synthetic environment and large distributed systems incorporate many optimizations based on empirical observations of the system and its users [24]; it is therefore not surprising that our test environment yields different results from real users. Even then, we find our synthetic environment to be an invaluable tool for debugging many egregious performance inefficiencies. While we cannot directly use the absolute data (e.g., latencies) we get from our synthetic environment, the relationships are often valid (e.g., if an optimization improves the latency of an operation with synthetic user it often does so for real users but the magnitude of the improvement may be different).

For most subtle changes though, we must run experiments in a live system serving real users. An alternate, less risky, option is to mirror live traffic to a test environment that incorporates the changes we wish to evaluate. While we have tried mirroring in the past, it is difficult to get right: if we wait for the mirrored operation to finish (or at least get queued to ensure ordering), we degrade the latency of user-facing operations; if we mirror operations asynchronously, our mirror diverges from our production system over time.

2.5 Effect of continuously-varying load

The continuously-varying load mixtures affect both the resource usage and latency of our application. For example, Figure 9 shows two curves: the dotted curve gives the scaled QPS to the Gmail backend and the solid line gives the 99th percentile latency of a particular operation

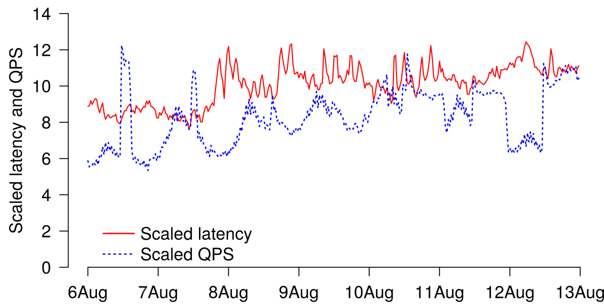
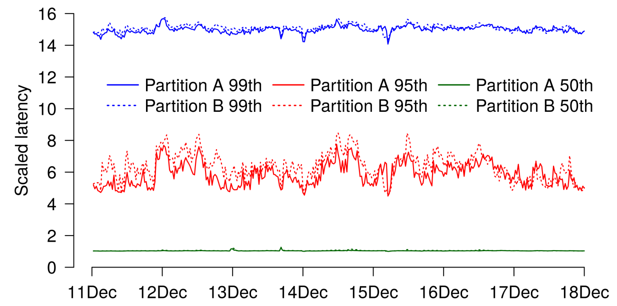
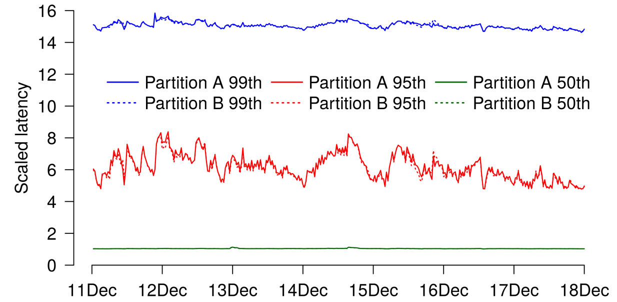


Figure 9: Changing load and changing latency

to the Gmail backend across tens of millions of users: this operation produces the list of message threads in a label (such as inbox). Since we are measuring the latency of a particular operation and our sample is large, we do not expect much variation in latency over time. Surprisingly, we see that latency varies by more than 50% over time as the load on the system changes. Furthermore the relationship between load and latency is not obvious or intuitive: e.g., the highest 99th percentile latency does not occur at the point that QPS is at its highest and sometimes load and latency appear negatively correlated.



(a) 100K users



(b) X0M users

Figure 10: Latency comparison: (a) 100K users, (b) X0M users

3 Our approach

Prior work has shown that even understanding the performance of systems with stable load (e.g., [23]) is difficult; thus, understanding the performance of a system with widely varying load (Section 2) will be even harder. Indeed we have found that even simple performance questions, such as "Does this change improve or degrade latency?" are difficult to answer. In this section we describe the methodologies and tools that we have developed to help us in our performance analysis.

Since we cannot easily reproduce performance phenomena in a synthetic environment, we conduct most of our experiments with live users; Section 3.1 explains how we do this. Section 3.2 describes how we can sometimes successfully use statistics to predict the outcome of experiments and the pitfalls we encounter in doing so. Section 3.3 describes the two contexts in which we need to debug the cause of slow or resource intensive operations in a system under constantly varying load.

3.1 Running experiments in a live serving system

As discussed in Section 2.4, we often need to do our performance analysis on systems serving live users. This section describes and explains our approach.

To conduct controlled experiments with real users we partition our users (randomly) and each experiment uses one partition as the test and the other as the control. The partitions use disjoined sets of processes: i.e., in the non-failure case, a user is served completely by the processes in their partition. Large partitions enable us to employ the law of large numbers to factor out differences between two samples of users. We always pick both the test and control in the same datacenter to minimize the difference in hardware between test and control and we have test and control pairs in all datacenters.

Figure 10(a) shows scaled 50th, 95th and 99th percentile latency for two partitions, each serving 100K active users. Each partition has the same number of processes and each process serves the same number of randomly selected users. We expect the two partitions to have identical latency distributions because neither is running an experiment but we see that this is not the case. For example, the 95th percentiles of the two partitions differ by up to 50% and often differ by at least 15%. Thus, 100K randomly selected users is clearly not enough to overwhelm the variation between users and operations.

Figure 10(b) shows scaled 50th, 95th, and 99th percentile latency for two partitions, each serving tens of millions users. With the exception of a few points where the 95th percentiles diverge, we see that these partitions

are largely indistinguishable from each other; the differences between the percentiles rarely exceed 1%.

Given that many fields routinely derive statistics from fewer than 100K observations, why do our samples differ by up to 50% at some points in time? The diversity of our user population is responsible for this: Gmail users exhibit a vast spread of activity and mailbox sizes with a long-tail. The 99th percentile mailbox size is four orders of magnitude larger than the median. The resource usage of many operations (such as synchronization operations and search) depend on the size of the mailbox. Thus, if one sample of 100K users ends up with even one more large user compared to another sample, it can make the first sample appear measurably worse. With larger samples we get a smaller standard deviation (by definition) in the distribution of mailbox sizes and thus it is less likely that two samples will differ significantly by chance. Intuitively, we need a greater imbalance (in absolute terms) in mailbox sizes as samples increase in size before we observe a measurable difference between the samples.

3.2 Using statistics to determine the impact of a change

Since a change may affect system performance differently under different load, we:

- Collect performance data from a sample of production. A large sample (tens of millions of randomly chosen users) makes it likely that we see all of the UVR behaviors.
- Collect performance data over an entire week and discount data from holiday seasons which traditionally have low load. By collecting data over an entire week we see a range of likely mixtures of UVR and non-UVR loads.
- Collect data from "induced" one-off events. For example, we can render one cluster as "non-serving" which forces other clusters to take on the additional load.
- Compare our test to a control at the same time and collect additional metrics to ensure that there was no other factor (e.g., a one-off event or the residual effects of a previous experiment) that renders the comparison invalid.

Once we have the data, we use statistical methods to analyze it. Unfortunately, the choice of the statistical method is not always obvious: most methods make assumptions about the data that they analyze (e.g., the distribution or independence of the data). Suppose, for example, that we want to test for statistically significant differences between latency in two partitions of randomly selected users (as illustrated in Figure 10). Because the Kolmogorov-Smirnov (K-S) test is non-parametric (without distributional assumptions) it seems like a good candidate to use. However, due to the scale of the application, we store only aggregate latency at various percentiles rather than latency of each operation. Applying the K-S test to the percentiles (treating the repeated measurements over time as independent observations) would violate the independence assumption and inflate the power of the test. Any time we violate the assumptions of a statistical method, the method may produce misleading results (e.g., [9]).

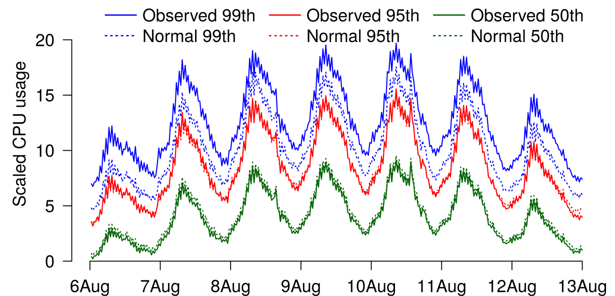


Figure 11: CPU of processes running the same binary has a near-normal distribution

This section gives two real examples: one where statistical modeling yields a valid outcome and one where it surprisingly does not.

3.2.1 Example 1: Data is near normal

A critical binary in our email system runs with N processes, each with C CPU cores serving U users. We wanted to deploy processes for $2U$ users but without using $2C$ CPU (our computers did not have $2C$ CPU). Since larger processes allow for more pooling of resources, we knew that the larger process would not need $2C$ CPU; but then how much CPU would it need?

Looking at the distribution of the CPU usage of the N processes, we observed that (per the central limit theorem) they exhibited a near normal distribution. Thus, at least in theory, we could calculate the properties of the larger process ($2N$ total processes) using the standard method for adding normal distributions. Concretely, when adding a normal distribution to itself, the resulting mean is two times the mean of the original distribution and the standard deviation is $\sqrt{2}$ of the standard deviation of the original distribution. Using this method we predicted the resources needed by the larger process and deployed it using the prediction.

The solid lines in Figure 11 show the observed 50th, 95th, and 99th percentiles of scaled CPU usage of the larger process using a sample of 3,000 processes. The

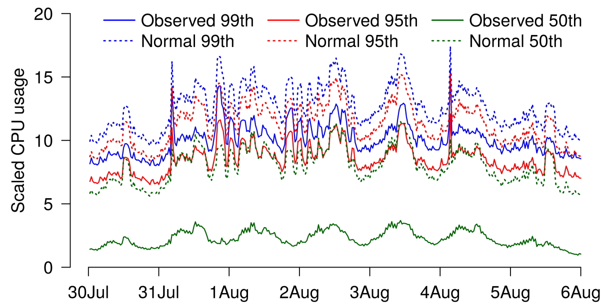


Figure 12: Time-shifted resource usage

dashed lines display the percentiles predicted by the statistical model. We see that the dashed and solid lines are close to each other (usually within 2%) at the 50th and 95th percentiles but the observed 99th percentile is always larger than the normal 99th percentile, usually by 10 to 20%. With the exception of the tail, our statistical model effectively predicted the resource needs of the larger process.

3.2.2 Example 2: Data is near normal but not independent

Our email application is made up of many different services which communicate with each other using RPC calls. Similar to the pooling in the previous section, we wanted to explore if allowing pairs of communicating processes (one from the higher-level binary and one for the lower-level binary) to share CPU would be beneficial.

The dashed ("Normal") lines in Figure 12 show the 50th, 95th, and 99th percentiles of scaled CPU usage that we expected when we shared resources between pairs of communicating processes. The solid ("Observed") lines show the actual percentiles of scaled CPU usage that we observed for the paired processes. To our surprise, the actual CPU usage is lower than the modeled CPU usage by up to 25% at the 95th and 99th percentiles and by 50% to 65% at the median. Clearly, our statistical model is incorrect.

On further investigation we determined that the CPU usage of the two communicating processes is not independent of each other: the two processes actually exhibit "time-shifted" CPU usage: when the higher-level process makes an RPC call to the lower-level process, it waits for the response and thus consumes no CPU (for that request); when the lower-level process returns a response it stops using CPU (for that request) and the higher-level process then becomes active processing the response. Thus, because these two distributions are not independent we cannot add them together.

3.3 Temporal and operation context

Section 3.2 shows how we approach establishing the impact of a change to the live system. Before we can determine what changes we need for a cloud application to become more responsive or cost effective, we need to understand why a request to a cloud application is slow or expensive. To achieve that understanding, we need to consider two contexts.

First, because cloud applications experience continuously varying load and load mixture (Section 2) we need to consider the *temporal context* of the request. By "temporal context" we mean all (possibly unrelated) concurrent activity on computers involved in serving our request throughout the time required to complete the request. Unrelated activity may degrade the performance of our request by, for example, causing cache evictions. While in our multi-tenant environment (i.e., many different applications share each computer) the operation context could include activity from processes uninvolved in serving our request, in practice we have found that this rarely happens: this is because each process runs in a container that (for the most part) isolates it from other processes running on the same computer. Thus, we are only interested in temporal context from processes that are actually involved in serving our request.

Second, a single request to a cloud application can involve many processes possibly running on different computers [21]. Furthermore, each process runs a binary with many software layers, often developed by independent teams. Thus a single user request involves remote-procedure calls between processes, functional calls between user-level software layers, and system calls between user-level software and the kernel. A request may be slow (expensive) because (i) a particular call is unreasonably slow (expensive), (ii) arguments to a particular call causes it to be slow (expensive), or (iii) the individual calls are fast but there is an unreasonably large number of them which adds up to a slow (expensive) request. Knowing the full tree of calls that make up an operation enables us to tease apart the above three cases. We call this the *operation context* because it includes the calls involved in a request along with application-level annotations on each call.

We now illustrate the value of the two contexts with a real example. While attempting to reduce the CPU reservation for the backend of our email service, we noticed that even though its average (over 1 minute) utilization was under 50%, we could not reduce its CPU reservation even slightly (e.g., by 5%) without degrading latency. The *temporal context* showed that RPCs to our service came in bursts that caused queuing even though the average utilization was low. The *full operation context* told us what operations were causing the RPCs in the bursts.

By optimizing those operations (essentially using batching to make fewer RPCs per operation) we were able to save 5% CPU for our service without degrading latency.

3.3.1 Coordinated bursty tracing

The obvious approach for using sampling yet getting the temporal context is to use bursty tracing [6, 19]). Rather than collecting a single event at each sample, bursty tracing collects a burst of events. Prior work triggers a burst based on a count (e.g., Arnold *et al.* trigger a burst on the nth invocation of a method). This approach works well when we are only interested in collecting a trace at a single layer, it does not work across processes or across layers. For example, imagine we collect a burst of requests $o_1, \cdots, o_n$ arriving at a server O and each of these operations need to make an RPC to a service $\mathscr{S}$. For scalability, $\mathscr{S}$ itself is comprised of multiple servers $s_1, \cdots, s_m$ which run on many computers (i.e., it is made up of many processes each running on a different computer) and the different o_i may each go to a different server in the $s_1, \cdots, s_m$ set. Thus, a burst of requests does not give us temporal context on any of the $s_1, \cdots, s_m$ servers. As another example, the nth invocation of a method will not correspond to the nth invocation of a system call and thus the bursts we get at the method level and at the kernel level will not be coordinated.

We can address the above issues with bursty tracing by having each layer (including the kernel) continuously record all events in a circular buffer. When an event of interest occurs, we save the contents of the buffers on all processes that participated in the event. In practice we have found this approach to be problematic because it needs to identify all processes involved in an event and save traces from all of these processes before the process overwrites the events. Recall that the different processes often run on different computers and possibly in different datacenters.

To solve the above limitations of bursty tracing, we use *coordinated bursty tracing* which uses wall-clock time to determine the start and end of a burst. By "coordinated" we mean that all computers and all layers collect the burst at the same time and thus we can *simultaneously* collect traces from all the layers that participated in servicing a user request. Since a burst is for a contiguous interval of time, we get the temporal context. Since we are collecting bursts across computers, we can stitch together the bursts into an operation context as long as we enabled coordinated bursty tracing on all of the involved computers.

We specify our bursts using a *burst-config* which is a 64-bit unsigned integer. The burst-config dictates both the duration and the period of the burst. Most commonly it is of the form:

$(1)^m(0)^n$ (in binary)

i.e., it is m 1s followed by n 0s (in base 2). Each process and each layer performs tracing whenever:

burst-config & WallTimeMillis() == burst-config

Intuitively, each burst lasts for 2^nms and there is one burst every 2^{n+m}ms. For example if we want to trace for 4ms every 32ms we would use the burst-config of 11100 (in base 2). Unlike common mechanisms for bursty tracing, this mechanism does not require the application to maintain state (e.g., count within a burst [6, 19]) or for the different processes to communicate with each other to coordinate their bursts; instead, it can control the bursts using a cheap conditional using only *local* information (i.e., wall-clock time).

Collecting many bursts spread out over a period of time ensures that we get the complete picture: i.e., by selecting an appropriate burst-config we can get bursts spread out over time and thus over many different loads and load mixes.

There are five challenges in using coordinated bursty tracing.

First, bursty tracing assumes that clocks across different computers are aligned. Fortunately because of true time [12] clock misalignment is not a problem for us in practice.

Second, we need to identify and enable coordinated bursty tracing on *all* processes involved in our request; otherwise we will get an incomplete operation context. Because we partition our users (Section 3.1) and processes of a partition only communicate with other processes of the same partition (except in the case of failure handling) we can readily identify a set of processes for which we need to enable coordinated bursty tracing to get both the operation and the temporal context of all requests by users in the partition.

Third, since coordinated bursty tracing is time based (and not count based as in prior work on bursty tracing), a burst may start or end in the middle of a request; thus we can get incomplete data for such requests. To alleviate this, we always pick a burst period that is at least 10 times the period of the request that we are interested in. This way, while some requests will be cut off, we will get full data for the majority of requests within each burst.

Fourth, any given burst may or may not actually contain the event of interest (e.g., a slow request); we need to search through our bursts to determine which ones contain interesting events. Sometimes this is as simple as looking for requests of a particular type that take longer than some threshold. Other times, the criteria is more subtle: e.g., we may be interested in requests that make at least one RPC to service $\mathscr{S}$ and $\mathscr{S}$ returns a particular size of a response. Thus, we have built tools that use

temporal logic to search over the bursts to find interesting operations [25].

Fifth, coordinated bursty tracing, and really any tracing, can itself perturb system behavior. If severe enough, the perturbation can mislead our performance analysis. The perturbation of coordinated bursty tracing depends on the traces that we are collecting; since traces for different layers have vastly different costs, we cannot quantify the perturbation of coordinated bursty tracing in a vacuum. Instead, we always confirm any findings with corroborating evidence or further experiments. In practice we have not yet encountered a situation where the perturbation due to bursty tracing was large enough to mislead us.

We use coordinated bursty tracing whenever we need to combine traces from different layers to solve a performance mystery. As a real example, cache traces at our storage layer showed that the cache was getting repeated requests for the same block in a short (milliseconds) time interval. While cache lookups are cheap, our cache stores blocks in compressed form and thus each read needs to pay the price of uncompressing the block. For efficiency reasons, the cache layer uses transient file identifiers and an offset as the key; thus logs of cache access contain only these identifiers and not the file system path of the file. Without knowing the path of the file, we did not know what data our application was repeatedly reading from the cache (the file path encodes the type of data in the file). The knowledge of the file path was in a higher level trace in the same process and the knowledge of the operation that was resulting in the reads was in another process. On enabling coordinated bursty tracing we found that most of the repeated reads were of the index block: a single index block provides mapping for many data blocks but we were repeatedly reading the index block for each data block. The problem was not obvious from the code structure: we would have to reason over a deeply nested loop structure spread out over multiple modules to find the problem via code inspection alone. The fix improved both the long-tail latency and CPU usage of our system.

3.3.2 Vertical context injection

Section 3.3.1 explores how we can collect a coordinated burst across layers. Since a burst contains traces from different layers, we need to associate events in one trace with events in another trace. More concretely, we would like to project all of the traces from a given machine so we get a holistic trace that combines knowledge from all the traces.

Simply interleaving traces based on timestamps or writing to the same log file is not enough to get a holistic trace. For example by interleaving RPC events with kernel events we will know that certain system calls, context switches, and interrupts occurred while an RPC was in progress. However, we will not know if those system calls were on behalf of the RPC or on behalf of some unrelated concurrent work.

The obvious approach to combining high- and low-level traces is to propagate the operation context through all the layers and tag all trace events with it. This approach is difficult because it requires us to propagate the context through many layers, some of which are not even within our direct control (e.g., libraries that we use or the kernel). We could use dynamic instrumentation tools, such as DTrace [1], to instrument all code including external libraries. This approach is unfortunately unsuitable because propagating the operation context through layers can require non-trivial logic; thus even if we could do this using DTrace, the code with the instrumentation would be significantly different from the code that we have reviewed and tested. Using such untested and unreviewed code for serving real user requests could potentially compromise our user's data and thus we (as a policy) never do this.

Our approach instead relies on the insight that any layer of the software stack can directly cause kernel-level events by making system calls. By making a stylized sequence of innocuous system calls, any layer can actually *inject* information into the kernel traces. [2]

For example, let's suppose we want to inject the RPC-level event, "start of an RPC," into a kernel trace. We could do this as follows:

```
syscall(getpid, kStartRpc);
syscall(getpid, RpcId);
```

The argument to the first *getpid* (*kStartRpc*) is a constant for "start of RPC." *getpid* ignores all arguments passed to it, but the kernel traces still record the value of those arguments. The argument to the second *getpid* identifies the RPC that is starting. Back-to-back *getpid* calls (with the first one having an argument of kStartRpc) are unlikely to appear naturally and thus the above sequence can reliably inject RPC start events into the kernel trace.

When we find the above sequence in the kernel trace we know that an RPC started but more importantly we know the thread on which the RPC started (the kernel traces contain the context switch events which enable us to tell which thread is running on which CPU). We can now tag all system calls on the thread with our RPC until either (i) the kernel preempts the thread, in which case the thread resumes working on our RPC when the kernel schedules it again, or (ii) the thread switches to working on a different CPU (which we again detect with a pattern of system calls).

[2] We discovered this idea in collaboration with *Dick Sites* at Google.

We use vertical context injection for other high-level events also. For example, by also injecting "just acquired a lock after waiting for a long time" into the kernel trace, we can uncover exactly what RPC was holding the lock that we were waiting on; it will be the one that invokes *sched_wakeup* to wake up the blocked *futex* call.

Implementing the above approach for injecting RPC and lock-contention information into our kernel traces took less than 100 lines of code. The analysis of the kernel traces to construct the holistic picture is of course more complex but this code is offline; it does not add any work or complexity to our live production system. In contrast, an approach that propagated context across the layers would have been far more intrusive and complex.

We use vertical context injection as a last resort: its strength is that it provides detailed information that frequently enables us to get to the bottom of whatever performance mystery we are chasing; its weakness is that these detailed traces are large and understanding them is complex.

As a concrete example, we noticed that our servers had low CPU utilization: under 50%. When we reduced the CPU per server (to increase utilization and save CPU) the latency of our service became worse. This indicated that there were micro bursts where the CPU utilization was higher than 50%; reducing the CPU per server was negatively affecting our micro bursts and thus degrading latency. We used vertical context injection along with coordinated bursty tracing to collect bursts of kernel traces. On analyzing the traces we found that reading user properties for each user request was responsible for the bursts; for each user request we need a number of properties and each property is small: e.g., one property gives the number of bytes used by the user. Rather than reading all properties at once, our system was making a separate RPC for each property which resulted in bursty behavior and short (a few milliseconds) CPU bursts. Reading all properties in a single RPC improved both the CPU and latency of our service. Vertical context injection enabled us to determine that the work in the bursts was related to servicing the RPCs involved in reading the properties.

4 Related work

We now review related work in the areas of analyzing the performance of cloud applications and performance tools.

Magpie [7, 8] collects data for all requests (thus the temporal context) and stitches together information across multiple computers to provide the operation context. Because Magpie collects data for all requests it does not scale to billions of user requests per day; Dapper and Canopy (discussed below) address this issue by using sampling.

Dapper [27] propagates trace identifiers to produce an RPC tree for an operation. We use Dapper extensively in our work because Dapper tells us exactly what RPCs execute as part of an operation and the performance data pertinent to the RPCs. By default Dapper performs random sampling: e.g., it may sample 1 in *N* traces. While this gives us the operation context, it does not give us the temporal context. Thus, we use Dapper in conjunction with coordinated bursty tracing and vertical context injection.

Canopy [22] effectively extends Dapper by providing APIs where any layer can add new events to a Dapper-like trace and users can use DSL to extend traces. Canopy's solution to combining data from multiple layers is to require each layer to use a Canopy API; while this is effective for high-level software layers, it may be unsuitable for low-level libraries and systems (e.g., OS kernel) because it creates a software dependency from critical systems software on high-level APIs.

Xtrace [15] uses auxiliary information (X-Trace Metadata) to tie together events that belong to the same operation. Thus, one could use Xtrace to tie together high-level RPC events and kernel-level events; however, unlike vertical context injection, Xtrace requires each layer to explicitly propagate the metadata across messages.

Jalaparti *et al.* [21] discuss the variations in latency that they observe in Bing along with mitigation strategies that involve trading off accuracy and resources with latency. Unlike our work, Jalaparti *et al.* do not explore the effects of time-varying load on performance.

Vertical profiling [17] recognizes the value of the operation context and subsequent work extends vertical profiling to combine traces from multiple layers using trace alignment [16]. Unlike our work, this work only considers the operation context within a single Java binary.

5 Discussion

This paper shows that for Gmail the load varies continually over time; these variations are not just due to changes in QPS; the actual nature of the load changes over time (for example, due to the the mix of IMAP traffic and web traffic). These variations create challenges for evaluating the performance impact of code changes. Concretely, (i) we cannot compare before-after performance data at different points in time and instead we run the test and control experiments simultaneously to ensure that they experience comparable loads; (ii) it is difficult to model this changing load in a synthetic environment and thus we conduct our experiments in a system serving live users; and (iii) we need large sample sizes (millions of users) to get statistically significant results.

We can take each source of variation described in this paper and devise a fix for it. For example, by isolating

IMAP and web traffic in disjoint services we circumvent the variation due to different mixes of IMAP and web traffic over the course of time. Unfortunately, such an approach does not work in general: there are many other causes of load variation besides the ones that we have described and fixes for each of the many causes becomes untenable quickly. The remainder of this section discusses some other sources of variation and in doing so generalizes the contributions of this paper.

The continuously changing software [14] causes performance to vary. A given operation to a Cloud application may involve hundreds of RPCs in tens of services; for example a median request to Bing involves 15 stages processed across 10s to 1000s of servers [21]. Different teams maintain and deploy these stages independently. To ensure the safety of the deployed code, we cannot deploy new versions of software atomically even if the number of instances is small enough to allow it (e.g., [20]). Instead we deploy it in a staged fashion: e.g., first we may deploy a software only to members of the team, then to random users at Google, then to small sets of randomly picked external users, and finally to all users. As a result, at any given time many different versions of a software may be running. Two identical operations at different points in time will often touch different versions of some of the services involved in serving the operation. Since different versions of a server may induce different loads (e.g., by making different RPCs to downstream servers or by inducing different retry behavior on the upstream servers), continuous deployment of software also continuously changes the load on our system.

Real-world events can easily change the mix or amount of load on our systems and thus cause performance to vary. For example, Google applications that are popular in academia (e.g., Drive or Classroom) see a surge of activity at back-to-school time. Some world events may be more subtle: e.g., a holiday in a place with limited or expensive internet connectivity may change the mix of operations to our system because clients in such areas often access our systems in a different mode (e.g., using offline mode or syncing mode) than clients in areas with good connectivity.

Datacenter management software can change the number of servers that are servicing user requests, move servers across physical machines, and turn up or turn down VMs; in doing so they directly vary the load on services.

In summary, there are many and widespread reasons beyond the ones explored in this paper that result in continuously varying load on cloud applications. Thus, we believe that the approaches in this paper for doing experiments in the presence of these variations and the tools that we have developed are widely applicable.

6 Conclusions

Performance analysis of cloud applications is important; with many cloud applications serving more than a billion active users, even a 1% saving in resources translates into significant cost savings. Furthermore, being able to maintain acceptable long-tail latency at such large scale requires constant investment in performance monitoring and analysis.

We show that performance analysis of cloud application is challenging and the challenges stem from constantly varying load patterns. Specifically, we show that superimposition of requests from users, different email clients, essential background work, and one-off events results in a continuously changing load that is hard to model and analyze. This paper describes how we meet this challenge. Specifically, our approach has the following components:

- We conduct performance analysis in a live application setting with each experiment setting involving millions of real users. Smaller samples are not enough to capture the diversity of our users and their usage patterns.
- We do longitudinal studies over a week or more to capture the varying load mixes. Additionally, we primarily compare data (between the test and control setups) at the same point in time so that they are serving a similar mix of load.
- We use statistics to analyze data and to predict the outcome of risky experiments and we corroborate the results with other data.
- We use coordinated bursty tracing to capture bursts of traces across computers and layers so we get both the temporal and the operation context needed for debugging long-tail performance issues.
- We project higher-level events into kernel events so that we can get a vertical trace which allows us to attribute all events (low or high level) to the high-level operation (e.g., RPC or user operation).

We have used the above strategies for analyzing and optimizing various large applications at Google including Gmail (more than 1 billion users) and Drive (hundreds of millions users).

7 Acknowledgments

Thanks to Brian Cooper, Rick Hank, Dick Sites, Garrick Toubassi, John Wilkes, and Matei Zaharia for feedback on the paper and for countless discussions on performance.

References

[1] About dtrace. http://dtrace.org/blogs/about.

[2] Google research blog: Speed matters. http://googleresearch.blogspot.com/2009/06/speed-matters.html, June 2009.

[3] Google loses data as lightning strikes. BBC News, Aug. 2015. http://www.bbc.com/news/technology-33989384.

[4] Facebook stats. http://newsroom.fb.com/company-info/, 2016.

[5] Gmail now has more than 1B monthly active users. https://techcrunch.com/2016/02/01/gmail-now-has-more-than-1b-monthly-active-users/, Feb. 2016.

[6] ARNOLD, M., HIND, M., AND RYDER, B. G. Online feedback-directed optimization of java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2002), OOPSLA '02, ACM, pp. 111–129.

[7] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 18–18.

[8] BARHAM, P., ISAACS, R., MORTIER, R., AND NARAYANAN, D. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Berkeley, CA, USA, 2003), HOTOS'03, USENIX Association, pp. 15–15.

[9] BLACKBURN, S. M., DIWAN, A., HAUSWIRTH, M., SWEENEY, P. F., AMARAL, J. N., BRECHT, T., BULEJ, L., CLICK, C., EECKHOUT, L., FISCHMEISTER, S., FRAMPTON, D., HENDREN, L. J., HIND, M., HOSKING, A. L., JONES, R. E., KALIBERA, T., KEYNES, N., NYSTROM, N., AND ZELLER, A. The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations. *ACM Trans. Program. Lang. Syst. 38*, 4 (Oct. 2016), 15:1–15:20.

[10] Apache cassandra. http://cassandra.apache.org/.

[11] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[12] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[13] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56* (2013), 74–80.

[14] FEITELSON, D., FRACHTENBERG, E., AND BECK, K. Development and deployment at facebook. *IEEE Internet Computing 17*, 4 (July 2013), 8–17.

[15] FONSECA, R., PORTER, G., KATZ, R. H., AND SHENKER, S. X-trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)* (Cambridge, MA, 2007), USENIX Association.

[16] HAUSWIRTH, M., DIWAN, A., SWEENEY, P. F., AND MOZER, M. C. Automating vertical profiling. *SIGPLAN Not. 40*, 10 (Oct. 2005), 281–296.

[17] HAUSWIRTH, M., SWEENEY, P. F., DIWAN, A., AND HIND, M. Vertical profiling: Understanding the behavior of object-priented applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2004), OOPSLA '04, ACM, pp. 251–269.

[18] HICKS, M., AND NETTLES, S. Dynamic software updating. *ACM Trans. Program. Lang. Syst. 27*, 6 (Nov. 2005), 1049–1096.

[19] HIRZEL, M., AND CHILIMBI, T. Bursty tracing: A framework for low-overhead temporal profiling. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization* (2001), pp. 117–126.

[20] ISARD, M. Autopilot: Automatic data center management. Tech. rep., April 2007.

[21] JALAPARTI, V., BODIK, P., KANDULA, S., MENACHE, I., RYBALKIN, M., AND YAN, C. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 219–230.

[22] KALDOR, J., MACE, J., BEJDA, M., GAO, E., KUROPATWA, W., O'NEILL, J., ONG, K. W., SCHALLER, B., SHAN, P., VISCOMI, B., VENKATARAMAN, V., VEERARAGHAVAN, K., AND SONG, Y. J. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 34–50.

[23] LI, J., SHARMA, N. K., PORTS, D. R. K., AND GRIBBLE, S. D. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 9:1–9:14.

[24] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 385–398.

[25] RYCKBOSCH, F., AND DIWAN, A. Analyzing performance traces using temporal formulas. *Softw., Pract. Exper. 44*, 7 (2014), 777–792.

[26] SCHURMAN, E., AND BRUTLAG, J. Performance related changes and their user impact. http://conferences.oreilly.com/velocity/velocity2009/-public/schedule/detail/8523, June 2009. Additional Bytes, and HTTP Chunking in Web Search.

[27] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a large-scale distributed systems tracing infrastructure.

[28] VEERARAGHAVAN, K., MEZA, J., CHOU, D., KIM, W., MARGULIS, S., MICHELSON, S., NISHTALA, R., OBENSHAIN, D., PERELMAN, D., AND SONG, Y. J. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 635–651.

007: Democratically Finding The Cause of Packet Drops

Behnaz Arzani[1], Selim Ciraci[2], Luiz Chamon[3], Yibo Zhu[1], Hongqiang (Harry) Liu[1], Jitu Padhye[2], Boon Thau Loo[3], Geoff Outhred[2]
[1] *Microsoft Research,*[2] *Microsoft,*[3] *University of Pennsylvania*

Abstract – Network failures continue to plague datacenter operators as their symptoms may not have direct correlation with where or why they occur. We introduce 007, a lightweight, always-on diagnosis application that can find problematic links and also pinpoint problems *for each TCP connection*. 007 is completely contained within the end host. During its two month deployment in a tier-1 datacenter, it detected every problem found by previously deployed monitoring tools while also finding the sources of other problems previously undetected.

1 Introduction

007 has an ambitious goal: for every packet drop on a TCP flow in a datacenter, find the link that dropped the packet and do so with negligible overhead and no changes to the network infrastructure.

This goal may sound like an overkill—after all, TCP is supposed to be able to deal with a few packet losses. Moreover, packet losses might occur due to congestion instead of network equipment failures. Even network failures might be transient. Above all, there is a danger of drowning in a sea of data without generating any actionable intelligence.

These objections are valid, but so is the need to diagnose "failures" that can result in severe problems for applications. For example, in our datacenters, VM images are stored in a storage service. When a VM boots, the image is mounted over the network. Even a small network outage or a few lossy links can cause the VM to "panic" and reboot. In fact, 17% of our VM reboots are due to network issues and in over 70% of these none of our monitoring tools were able to find the links that caused the problem.

VM reboots affect customers and we need to understand their root cause. Any persistent pattern in such transient failures is a cause for concern and is potentially actionable. One example is silent packet drops [1]. These types of problems are nearly impossible to detect with traditional monitoring tools (e.g., SNMP). If a switch is experiencing these problems, we may want to reboot or replace it. These interventions are "costly" as they affect a large number of flows/VMs. Therefore, careful blame assignment is necessary. Naturally, this is only one example that would benefit from such a detection system.

There is a lot of prior work on network failure diagnosis, though one of the existing systems meet our ambitious goal. Pingmesh [1] sends periodic probes to detect failures and can leave "gaps" in coverage, as it must manage the overhead of probing. Also, since it uses out-of-band probes, it cannot detect failures that affect only in-band data. Roy et al. [2] monitor all paths to detect failures but require modifications to routers and special features in the switch (§10). Everflow [3] can be used to find the location of packet drops but it would require capturing all traffic and is not scalable. We asked our operators what would be the most useful solution for them. Responses included: "In a network of $\geq 10^6$ links its a reasonable assumption that there is a non-zero chance that a number (> 10) of these links are bad (due to device, port, or cable, etc.) and we cannot fix them simultaneously. Therefore, fixes need to be prioritized based on customer impact. However, currently we do not have a direct way to correlate customer impact with bad links". This shows that current systems do not satisfy operator needs as they do not provide application and connection level context.

To address these limitations, we propose 007, a simple, lightweight, always-on monitoring tool. 007 records the path of TCP connections (flows) suffering from one or more retransmissions and assigns proportional "blame" to each link on the path. It then provides a *ranking* of links that represents their relative drop rates. Using this ranking, it can find the most likely cause of drops in each TCP flow.

007 has several noteworthy properties. First, it does not require any changes to the existing networking infrastructure. Second, it does not require changes to the client software—the monitoring agent is an independent entity that sits on the side. Third, it detects in-band failures. Fourth, it continues to perform well in the presence of noise (e.g. lone packet drops). Finally, it's overhead is negligible.

While the high-level design of 007 appear simple, the practical challenges of making 007 work and the theoretical challenge of *proving* it works are non-trivial. For example, its path discovery is based on a traceroute-like approach. Due to the use of ECMP, traceroute packets have to be carefully crafted to ensure that they follow the same path as the TCP flow. Also, we must ensure that we do not overwhelm routers by sending too many traceroutes (traceroute responses are handled by control-plane CPUs of routers, which are quite puny). Thus, we

need to ensure that our sampling strikes the right balance between accuracy and the overhead on the switches. On the theoretical side, we are able to show that 007's simple blame assignment scheme is highly accurate even in the presence of noise.

We make the following contributions: (i) we design 007, a simple, lightweight, and yet accurate fault localization system for datacenter networks; (ii) we prove that 007 is accurate without imposing excessive burden on the switches; (iii) we prove that its blame assignment scheme correctly finds the failed links with high probability; and (iv) we show how to tackle numerous practical challenges involved in deploying 007 in a real datacenter.

Our results from a two month deployment of 007 in a datacenter show that it finds all problems found by other previously deployed monitoring tools while also finding the sources of problems for which information is not provided by these monitoring tools.

2 Motivation

007 aims to identify the cause of retransmissions with high probability. It is is driven by two practical requirements: (i) it should scale to datacenter size networks and (ii) it should be deployable in a running datacenter with as little change to the infrastructure as possible. Our current focus is mainly on analyzing infrastructure traffic, especially connections to services such as storage as these can have severe consequences (see §1, [4]). Nevertheless, the same mechanisms can be used in other contexts as well (see §9). We deliberately include congestion-induced retransmissions. If episodes of congestion, however short-lived, are common on a link, we want to be able to flag them. Of course, in practice, any such system needs to deal with a certain amount of noise, a concept we formalize later.

There are a number of ways to find the cause of packet drops. One can monitor switch counters. These are inherently unreliable [5] and monitoring thousands of switches at a fine time granularity is not scalable. One can use new hardware capabilities to gather more useful information [6]. Correlating this data with each retransmission *reliably* is difficult. Furthermore, time is needed until such hardware is production-ready and switches are upgraded. Complicating matters, operators may be unwilling to incur the expense and overhead of such changes [4]. One can use PingMesh [1] to send probe packets and monitor link status. Such systems suffer from a rate of probing trade-off: sending too many probes creates unacceptable overhead whereas reducing the probing rate leaves temporal and spatial gaps in coverage. More importantly, the probe traffic does not capture what the end user and TCP flows see. Instead, we choose to use data traffic itself as probe traffic. Using data traffic has the advantage that the system introduces little to no monitoring overhead.

As one might expect, almost all traffic in our datacenters is TCP traffic. One way to monitor TCP traffic is to use a system like Everflow. Everflow inserts a special tag in every packet and has the switches mirror tagged packets to special collection servers. Thus, if a tagged packet is dropped, we can determine the link on which it happened. Unfortunately, there is no way to know in advance which packet is going to be dropped, so we would have to tag and mirror every TCP packet. This is clearly infeasible. We could tag only a fraction of packets, but doing so would result in another sampling rate trade-off. Hence, we choose to rely on some form of network tomography [7, 8, 9]. We can take advantage of the fact that TCP is a connection-oriented, reliable delivery protocol so that any packet loss results in retransmissions that are easy to detect.

If we knew the path of all flows, we could set up an optimization to find which link dropped the packet. Such an optimization would minimize the number of "blamed" links while simultaneously explaining the cause of all drops. Indeed past approaches such as MAX COVERAGE and Tomo [10, 11] aim to approximate the solution of such an optimization (see §12 for an example). There are problems with this approach: (i) the optimization is NP-hard [12]. Solving it on a datacenter scale is infeasible. (ii) tracking the path of every flow in the datacenter is not scalable in our setting. We can use alternative solutions such as Everflow or the approach of [2] to track the path of SYN packets. However, both rely on making changes to the switches. The only way to find the path of a flow without any special infrastructure support is to employ something like a traceroute. Traceroute relies on getting ICMP TTL exceeded messages back from the switches. These messages are generated by the control-plane, i.e., the switch CPU. To avoid overloading the CPU, our administrators have capped the rate of ICMP responses to 100 per second. This severely limits the number of flows we can track.

Given these limitations, what can we do? We analyzed the drop patterns in two of our datacenters and found: typically when there are packet drops, multiple flows experience drops. We show this in Figure 1a for TCP flows in production datacenters. The figure shows the number of flows experiencing drops in the datacenter conditioned on the total number of packets dropped in that datacenter in 30 second intervals. The data spans one day. We see that the more packets are dropped in the datacenter,

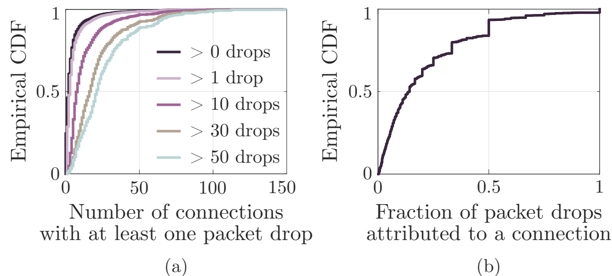


Figure 1: Observations from a production network: (a) CDF of the number of flows with at least one retransmission; (b) CDF of the fraction of drops belonging to each flow in each 30 second interval.

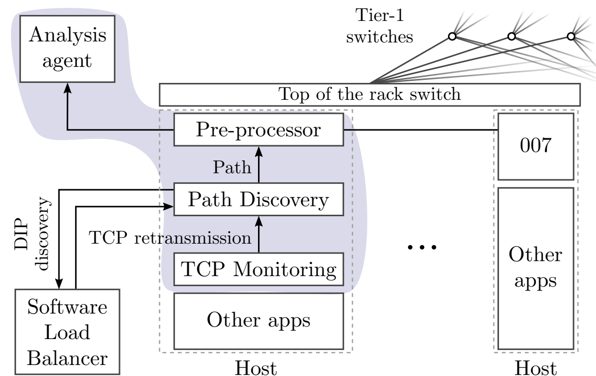


Figure 2: Overview of 007 architecture

the more flows experience drops and 95% of the time, at least 3 flows see drops when we condition on ≥ 10 total drops. We focus on the ≥ 10 case because lower values mostly capture noisy drops due to one-off packet drops by healthy links. In most cases drops are distributed across flows and no single flow sees more than 40% of the total packet drops. This is shown in Figure 1b (we have discarded all flows with 0 drops and cases where the total number of drops was less than 10). We see that in $\geq 80\%$ of cases, no single flow captures more than 34% of all drops.

Based on these observations and the high path diversity in datacenter networks [13], we show that if: (a) we only track the path of those flows that have retransmissions, (b) assign each link on the path of such a flow a vote of $1/h$, where h is the path length, and (c) sum up the votes during a given period, then the top-voted links are almost always the ones dropping packets (see §5)! Unlike the optimization, our scheme is able to provide a *ranking* of the links in terms of their drop rates, i.e. if link A has a higher vote than B, it is also dropping more packets (with high probability). This gives us a heat-map of our network which highlights the links with the most impact to a *given application/customer* (because we know which links impact a particular flows).

3 Design Overview

Figure 2 shows the overall architecture of 007. It is deployed alongside other applications on each end-host as a user-level process running in the host OS. 007 consists of three agents responsible for TCP monitoring, path discovery, and analysis.

The *TCP monitoring agent* detects retransmissions at each end-host. The Event Tracing For Windows (ETW) [14] framework[1] notifies the agent as soon as an active flow suffers a retransmission.

Upon a retransmission, the monitoring agent triggers the *path discovery agent* (§4) which identifies the flow's path to the destination IP (DIP).

At the end-hosts, a voting scheme (§5) is used based on the paths of flows that had retransmissions. At regular intervals of 30s the votes are tallied by a centralized *analysis agent* to find the top-voted links. Although we use an aggregation interval of 30s, failures *do not* have to last for 30s.

007's implementation consists of 6000 lines of C++ code. Its memory usage never goes beyond 600 KB on any of our production hosts, its CPU utilization is minimal (1-3%), and its bandwidth utilization due to traceroute is minimal (maximum of 200 KBps per host). 007 is proven to be accurate (§5) in typical datacenter conditions (a full description of the assumed conditions can be found in §9).

4 The Path Discovery Agent

The path discovery agent uses traceroute packets to find the path of flows that suffer retransmissions. These packets are used solely to identify the path of a flow. They do not need to be dropped for 007 to operate. We first ensure that the number of traceroutes sent by the agent does not overload our switches (§4.1). Then, we briefly describe the key engineering issues and how we solve them (§4.2).

4.1 ICMP Rate Limiting

Generating ICMP packets in response to traceroute consumes switch CPU, which is a valuable resource. In our network, there is a cap of $T_{\max} = 100$ on the number of ICMP messages a switch can send per second. To ensure that the traceroute load does not exceed $T_{\max}$, we start by noticing that a small fraction of flows go through tier-3 switches (T_3). Indeed, after monitoring all TCP flows in our network for one hour, only 2.1% went through a T_3 switch. Thus we can ignore T_3 switches in our analysis. Given that our network is a Clos topology and assuming that hosts under a top of the rack switch (ToR) communicate with hosts under a different ToR uniformly at random (see §6 for when this is not the case):

[1] Similar functionality exists in Linux.

Theorem 1. *The rate of ICMP packets sent by any switch due to a traceroute is below $T_{\max}$ if the rate C_t at which hosts send traceroutes is upper bounded as*

$$C_t \leq \frac{T_{\max}}{n_0 H} \min\left[n_1, \frac{n_2(n_0 n_{\text{pod}} - 1)}{n_0(n_{\text{pod}} - 1)}\right], \qquad (1)$$

where n_0, n_1, and n_2, are the numbers of ToR, T_1, and T_2 switches respectively, n_{pod} is the number of pods, and H is the number of hosts under each ToR.

See §12 for proof. The upper bound of C_t in our datacenters is 10. As long as hosts do not have more than 10 flows *with retransmissions* per second, we can *guarantee* that the number of traceroutes sent by 007 will not go above $T_{\max}$. We use C_t as a threshold to limit the traceroute rate of each host. Note that there are two independent rate limits, one set at the host by 007 and the other set by the network operators on the switch ($T_{\max}$). Additionally, the agent triggers path discovery *for a given connection* no more than once every epoch to further limit the number of traceroutes. We will show in §5 that this number is sufficient to ensure high accuracy.

4.2 Engineering Challenges

Using the correct five-tuple. As in most datacenters, our network also uses ECMP. All packets of a given flow, defined by the five-tuple, follow the same path [15]. Thus, traceroute packets must have the same five-tuple as the flow we want to trace. To ensure this, we must account for load balancers.

TCP connections are initiated in our datacenter in a way similar to that described in [16]. The connection is first established to a virtual IP (VIP) and the SYN packet (containing the VIP as destination) goes to a software load balancer (SLB) which assigns that flow to a physical destination IP (DIP) and a service port associated with that VIP. The SLB then sends a configuration message to the virtual switch (vSwitch) in the hypervisor of the source machine that registers that DIP with that vSwitch. The destination of all subsequent packets in that flow have the DIP as their destination and do not go through the SLB. For the path of the traceroute packets to match that of the data packets, its header should contain the DIP and not the VIP. Thus, before tracing the path of a flow, the path discovery agent first queries the SLB for the VIP-to-DIP mapping for that flow. An alternative is to query the vSwitch. In the instances where the failure also results in connection termination the mapping may be removed from the vSwitch table. It is therefore more reliable to query the SLB. Note that there are cases where the TCP connection establishment itself may fail due to packet loss. Path discovery is not triggered for such connections. It is also not triggered when the query to the SLB fails to avoid tracerouting the internet.

Re-routing and packet drops. Traceroute itself may fail. This may happen if the link drop rate is high or due to a blackhole. This actually helps us, as it directly pinpoints the faulty link and our analysis engine (§5) is able to use such partial traceroutes.

A more insidious possibility is that routing may change by the time traceroute starts. We use BGP in our datacenter and a lossy link may cause one or more BGP sessions to fail, triggering rerouting. Then, the traceroute packets may take a different path than the original connection. However, RTTs in a datacenter are typically less than 1 or 2 ms, so TCP retransmits a dropped packet quickly. The ETW framework notifies the monitoring agent immediately, which invokes the path discovery agent. The only additional delay is the time required to query the SLB to obtain the VIP-to-DIP mapping, which is typically less than a millisecond. Thus, as long as paths are stable for a few milliseconds after a packet drop, the traceroute packets will follow the same path as the flow and the probability of error is low. Past work has shown this to be usually the case [17].

Our network also makes use of link aggregation (LAG) [18]. However, unless all the links in the aggregation group fail, the L3 path is not affected.

Router aliasing [19]. This problem is easily solved in a datacenter, as we know the topology, names, and IPs of all routers and interfaces. We can simply map the IPs from the traceroutes to the switch names.

To summarize, 007's path discovery implementation is as follows: Once the TCP monitoring agent notifies the path discovery agent that a flow has suffered a retransmission, the path discovery agent checks its cache of discovered path for that epoch and if need be, queries the SLB for the DIP. It then sends 15 appropriately crafted TCP packets with TTL values ranging from 0–15. In order to disambiguate the responses, the TTL value is also encoded in the IP ID field [20]. This allows for concurrent traceroutes to multiple destinations. The TCP packets deliberately carry a bad checksum so that they do not interfere with the ongoing connection.

5 The Analysis Agent

Here, we describe 007's analysis agent focusing on its voting-based scheme. We also present alternative NP-hard optimization solutions for comparison.

5.1 Voting-Based Scheme

007's analysis agent uses a simple voting scheme. If a flow sees a retransmission, 007 votes its links as *bad*. Each vote has a value that is tallied at the end of every epoch, providing a natural ranking of the links. We set the value of good votes to 0 (if a flow has no

retransmission, no traceroute is needed). Bad votes are assigned a value of $\frac{1}{h}$, where h is the number of hops on the path, since each link on the path is equally likely to be responsible for the drop.

The ranking obtained after compiling the votes allows us to identify the most likely cause of drops on each flow: links ranked higher have higher drop rates (Theorem 2). To further guard against high levels of noise, we can use our knowledge of the topology to adjust the links votes. Namely, we iteratively pick the most voted link $l_{\max}$ and estimate the portion of votes obtained by all other links due to failures on $l_{\max}$. This estimate is obtained for each link k by (i) assuming all flows having retransmissions and going through $l_{\max}$ had drops due to $l_{\max}$ and (ii) finding what fraction of these flows go through k by assuming ECMP distributes flows uniformly at random. Our evaluations showed that this results in a 5% reduction in false positives.

Algorithm 1 Finding the most problematic links in the network.

```
1: 𝓛 ← Set of all links
2: 𝓟 ← Set of all possible paths
3: v(l_i) ← Number of votes for l_i ∈ 𝓛
4: 𝓑 ← Set of most problematic links
5: l_max ← Link with maximum votes in ∀l_i ∈ 𝓛 ∩ 𝓑^c
6: while v(l_max) ≥ 0.01(∑_{l_i ∈ 𝓛} v(l_i)) do
7:     l_max ← argmax_{l_i ∈ 𝓛 ∩ 𝓑^c} v(l_i)
8:     𝓑 ← 𝓑 ∪ {l_max}
9:     for l_i ∈ 𝓛 ∩ 𝓑^c do
10:        if ∃ p_i ∈ 𝓟 s.t. l_i ∈ p_i & l_max ∈ p_i then
11:            Adjust the score of l_i
12:        end if
13:    end for
14: end while
15: return 𝓑
```

007 can also be used to detect failed links using Algorithm 1. The algorithm sorts the links based on their votes and uses a threshold to determine if there are problematic links. If so, it adjusts the votes of all other links and repeats until no link has votes above the threshold. In Algorithm 1, we use a threshold of 1% of the total votes cast based on a parameter sweep where we found that it provides a reasonable trade-off between precision and recall. Higher values reduce false positives but increase false negatives.

Here we have focused on detecting link failures. 007 can also be used to detect switch failures in a similar fashion by applying votes to switches instead of links. This is beyond the scope of this work.

5.2 Voting Scheme Analysis

Can 007 deliver on its promise of finding the most probable cause of packet drops on each flow? This is not trivial. In its voting scheme, failed connections contribute to increase the tally of both good and bad links. Moreover, in a large datacenter such as ours, occasional, lone, and sporadic drops can and *will* happen due to good links. These failures are akin to noise and can cause severe inaccuracies in any detection system [21], 007 included. We show that the likelihood of 007 making these errors is small. Given our topology (Clos):

Theorem 2. *For $n_{\text{pod}} \geq \frac{n_0}{n_1}+1$, 007 will find with probability $1-2e^{-\mathcal{O}(N)}$ the $k < \frac{n_2(n_0 n_{\text{pod}}-1)}{n_0(n_{\text{pod}}-1)}$ bad links that drop packets with probability p_b among good links that drop packets with probability p_g if*

$$p_g \leq (n_u \alpha)^{-1}\left[1-(1-p_b)^{n_l}\right],$$

where N is the total number of flows between hosts, n_l and n_u are lower and upper bounds, respectively, on the number of packets per connection, and

$$\alpha = \frac{n_0(4n_0-k)(n_{\text{pod}}-1)}{n_2(n_0 n_{\text{pod}}-1)-n_0(n_{\text{pod}}-1)k}. \qquad (2)$$

The proof is deferred to the appendices due to space constraints. Theorem 2 states that under mild conditions, links with higher drop rates are ranked higher by 007. Since a single flow is unlikely to go through more than one failed link in a network with thousands of links, it allows 007 to find the most likely cause of packet drops on each flow.

A corollary of Theorem 2 is that in the absence of noise ($p_g = 0$), 007 can find all bad links with high probability. In the presence of noise, 007 can still identify the bad links as long as the probability of dropping packets on non-failed links is low enough (the signal-to-noise ratio is large enough). This number is compatible with typical values found in practice. As an example, let n_l and n_u be the 10th and 90th percentiles respectively of the number of packets sent by TCP flows across all hosts in a 3 hour period. If $p_b \geq 0.05\%$, the drop rate on good links can be as high as 1.8×10^{-6}. Drop rates in a production datacenter are typically below 10^{-8} [22].

Another important consequence of Theorem 2 is that it establishes that the probability of errors in 007's results diminishes exponentially with N, so that even with the limits imposed by Theorem 1 we can accurately identify the failed links. The conditions in Theorem 2 are sufficient but not necessary. In fact, §6 shows how well 007 performs even when the conditions in Theorem 2 do not hold.

5.3 Optimization-Based Solutions

One of the advantages of 007's voting scheme is its simplicity. Given additional time and resources we may consider searching for the optimal sets of failed links by finding the most likely cause of drops given the available evidence. For instance, we can find the

least number of links that explain all failures as we know the flows that had packet drops and their path. This can be written as an optimization problem we call the *binary program*. Explicitly,

$$\begin{aligned} \text{minimize} \quad & \|\boldsymbol{p}\|_0 \\ \text{subject to} \quad & \boldsymbol{A}\boldsymbol{p} \geq \boldsymbol{s} \\ & \boldsymbol{p} \in \{0,1\}^L \end{aligned} \tag{3}$$

where $\boldsymbol{A}$ is a $C \times L$ routing matrix; $\boldsymbol{s}$ is a $C \times 1$ vector that collects the status of each flow during an epoch (each element of $\boldsymbol{s}$ is 1 if the connection experienced at least one retransmission and 0 otherwise); L is the number of links; C is the number of connections in an epoch; and $\|\boldsymbol{p}\|_0$ denotes the number of nonzero entries of the vector $\boldsymbol{p}$. Indeed, if the solution of (3) is $\boldsymbol{p}^\star$, then the i-th element of $\boldsymbol{p}^\star$ indicates whether the binary program estimates that link i failed.

Problem (3) is the NP-hard minimum set covering problem [23] and is intractable. Its solutions can be approximated greedily as in MAX COVERAGE or Tomo [10, 11] (see appendix). For benchmarking, we compare 007 to the true solution of (3) obtained by a mixed-integer linear program (MILP) solver [24]. Our evaluations showed that 007 (Algorithm 1) significantly outperforms this binary optimization (by more than 50% in the presence of noise). We illustrate this point in Figures 4 and 10, but otherwise omit results for this optimization in §6 for clarity.

The binary program (3) does not provide a ranking of links. We also consider a solution in which we determine the number of packets dropped by each link, thus creating a natural ranking. The *integer program* can be written as

$$\begin{aligned} \text{minimize} \quad & \|\boldsymbol{p}\|_0 \\ \text{subject to} \quad & \boldsymbol{A}\boldsymbol{p} \geq \boldsymbol{c} \\ & \|\boldsymbol{p}\|_1 = \|\boldsymbol{c}\|_1 \\ & p_i \in \mathbb{N} \cup \{0\} \end{aligned} \tag{4}$$

where $\mathbb{N}$ is the set of natural numbers and $\boldsymbol{c}$ is a $C \times 1$ vector that collects the number of retransmissions suffered by each flow during an epoch. The solution $\boldsymbol{p}^\star$ of (4) represents the number of packets dropped by each link, which provides a ranking. The constraint $\|\boldsymbol{p}\|_1 = \|\boldsymbol{c}\|_1$ ensures each failure is explained only once. As with (3), this problem is NP-hard [12] and is only used as a benchmark. As it uses more information than the binary program (the number of failures), (4) performs better (see §6).

In the next three sections, we present our evaluation of 007 in simulations (§6), in a test cluster (§7), and in one of our production datacenters (§8).

6 Evaluations: Simulations

We start by evaluating in simulations where we know the ground truth. 007 first finds flows whose drops were due to noise and marks them as "noise drops". It then finds the link most likely responsible for drops on the remaining set of flows ("failure drops"). A noisy drop is defined as one where the corresponding link only dropped a single packet. 007 never marked a connection into the noisy category incorrectly. We therefore focus on the accuracy for connections that 007 puts into the failure drop class.

Performance metrics. Our measure for the performance of 007 is *accuracy*, which is the proportion of correctly identified drop causes. For evaluating Algorithm 1, we use *recall* and *precision*. Recall is a measure of reliability and shows how many of the failures 007 can detect (false negatives). For example, if there are 100 failed links and 007 detects 90 of them, its recall is 90%. Precision is a measure of accuracy and shows to what extent 007's results can be trusted (false positives). For example, if 007 flags 100 links as bad, but only 90 of those links actually failed, its precision is 90%.

Simulation setup. We use a flow level simulator [25] implemented in MATLAB. Our topology consists of 4160 links, 2 pods, and 20 ToRs per pod. Each host establishes 2 connections per second to a random ToR outside of its rack. The simulator has two types of links. For *good links*, packets are dropped at a very low rate chosen uniformly from $(0, 10^{-6})$ to simulate noise. On the other hand, *failed links* have a higher drop rate to simulate failures. By default, drop rates on failed links are set to vary uniformly from 0.01% to 1%, though to study the impact of drop rates we do allow this rate to vary as an input parameter. The number of good and failed links is also tunable. Every 30 seconds of simulation time, we send up to 100 packets per flow and drop them based on the rates above as they traverse links along the path. The simulator records all flows with at least one drop and for each such flow, the link with the most drops.

We compare 007 against the solutions described in §5.3. We only show results for the binary program (3) in Figures 4 and 10 since its performance is typically inferior to 007 and the integer program (4) due to noise. This also applies to MAX COVERAGE or Tomo [10, 11, 26] *as they are approximations of the binary program* (see [27]).

6.1 In The Optimal Case

The bounds of Theorem 2 are sufficient (not necessary) conditions for accuracy. We first validate that 007 can achieve high levels of accuracy as expected when these bounds hold. We set the drop rates on the failed links to be between $(0.05\%, 1\%)$. We refer the

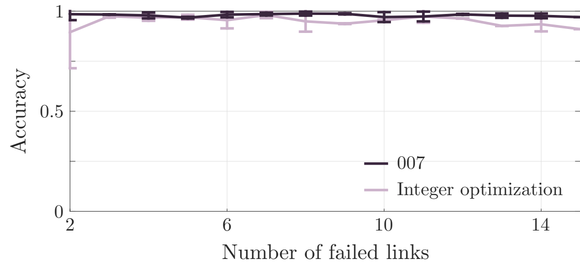


Figure 3: When Theorem 2 holds.

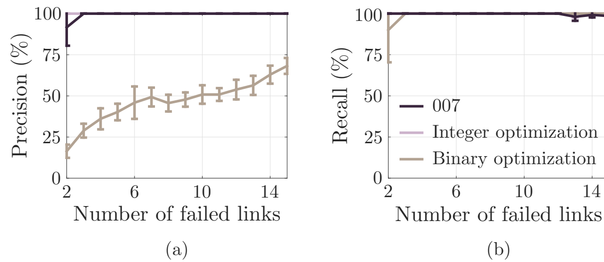


Figure 4: Algorithm 1 when Theorem 2 holds.

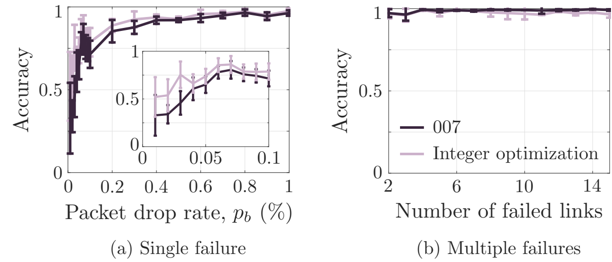


Figure 5: 007's accuracy for varying drop rates.

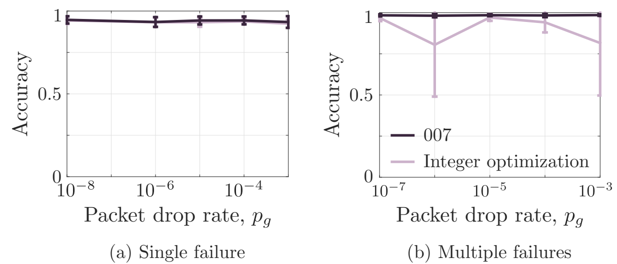


Figure 6: 007's accuracy for varying noise levels.

reader to [2] for why these drop rates are reasonable.

Accuracy. Figure 3 shows that 007 has an average accuracy that is higher than 96% in almost all cases. Due to its robustness to noise, it also outperforms the optimization algorithm (§ 5.3) in most cases.

Recall & precision. Figure 4 shows that even when failed links have low packet drop rates, 007 detects them with high recall/precision.

We proceed to evaluate 007's accuracy when the bounds in Theorem 2 *do not* hold. This shows these conditions are not necessary for good performance.

6.2 Varying Drop Rates

Our next experiment aims to push the boundaries of Theorem 2 by varying the "failed" links drop rates below the conservative bounds of Theorem 2.

Single Failure. Figure 5a shows results for different drop rates on a single failed link. It shows that 007 can find the cause of drops on each flow with high accuracy. Even as the drop rate decreases below the bounds of Theorem 2, we see that 007 can maintain accuracy on par with the optimization.

Multiple Failures. Figure 5b shows that 007 is successful at finding the link responsible for a drop even when links have very different drop rates. Prior work have reported the difficulty of detecting such cases [2]. However, 007's accuracy remains high.

6.3 Impact of Noise

Single Failure. We vary noise levels by changing the drop rate of good links. We see that higher noise levels have little impact on 007's ability to find the cause of drops on individual flows (Figure 6a).

Multiple Failures. We repeat this experiment for the case of 5 failed links. Figure 6b shows the results. 007 shows little sensitivity to the increase in noise when finding the cause of per-flow drops. Note that the large confidence intervals of the optimization is a result of its high sensitivity to noise.

6.4 Varying Number of Connections

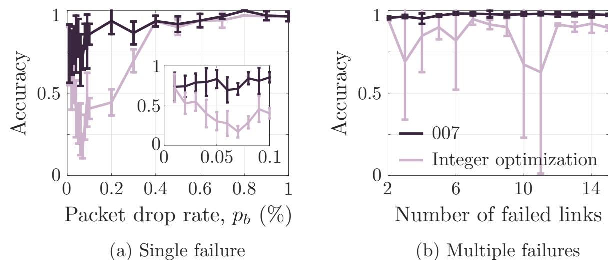
In previous experiments, hosts opened 60 connections per epoch. Here, we allow hosts to choose the number of connections they create per epoch uniformly at random between $(10, 60)$. Recall, from Theorem 2, that a larger number of connections from each host helps 007 improve its accuracy.

Single Failure. Figure 7a shows the results. 007 accurately finds the cause of packet drops on each connection. It also outperforms the optimization when the failed link has a low drop rate. This is because the optimization has multiple optimal points and is not sufficiently constrained.

Multiple Failures. Figure 7b shows the results for multiple failures. The optimization suffers from the lack of information to constrain the set of results. It therefore has a large variance (confidence intervals). 007 on the other hand maintains high probability of detection no matter the number of failures.

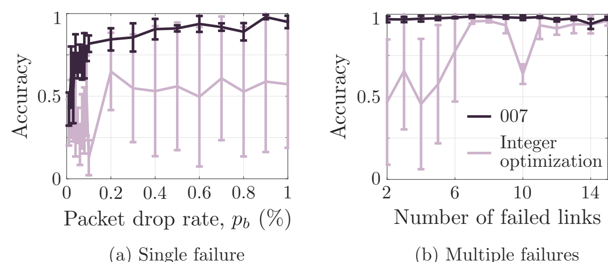
6.5 Impact of Traffic Skews

Single Failure. We next demonstrate 007's ability to detect the cause of drops even under heavily skewed traffic. We pick 10 ToRs at random (25% of the ToRs). To skew the traffic, 80% of the flows have destinations set to hosts under these 10 ToRs. The remaining flows are routed to randomly chosen hosts. Figure 8a shows that the optimization is much more heavily impacted by the skew than 007. 007 continues to detect the cause of drops with high probability



(a) Single failure (b) Multiple failures

Figure 7: Varying the number of connections.



(a) Single failure (b) Multiple failures

Figure 8: 007's accuracy under skewed traffic.

($\geq 85\%$) for drop rates higher than 0.1%.

Multiple Failures. We repeated the above for multiple failures. Figure 8b shows that the optimization's accuracy suffers. It consistently shows a low detection rate as its constraints are not sufficient in guiding the optimizer to the right solution. 007 maintains a detection rate of $\geq 98\%$ at all times.

Hot ToR. A special instance of traffic skew occurs in the presence of a single hot ToR which acts as a sink for a large number of flows. Figure 9 shows how 007 performs in these situations. 007 can tolerate up to 50% skew, i.e., 50% of *all* flows go to the hot ToR, with negligible accuracy degradation. However, skews above 50% negatively impact its accuracy in the presence of a large number of failures (≥ 10). Such scenarios are unlikely as datacenter load balancing mitigates such extreme situations.

6.6 Detecting Bad Links

In our previous experiments, we focused on 007's accuracy on a per connection basis. In our next experiment, we evaluate its ability to detect bad links.

Single Failure. Figure 10 shows the results. 007 outperforms the optimization as it does not require a fully specified set of equations to provide a best guess as to which links failed. We also evaluate the impact of failure location on our results (Figure 11).

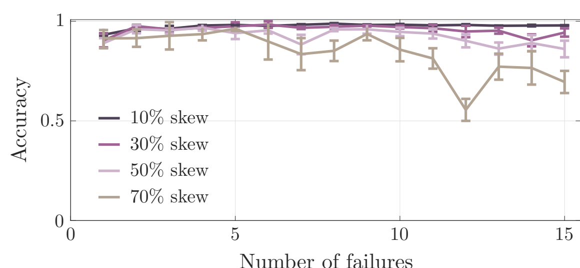
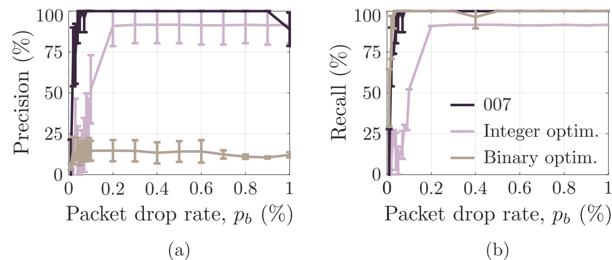
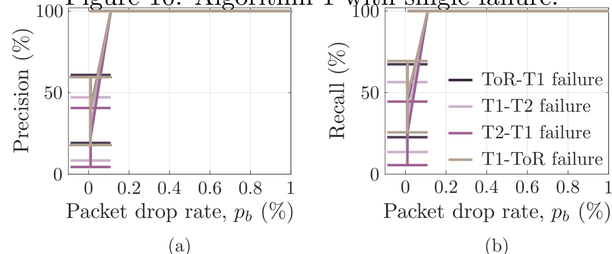


Figure 9: Impact of a hot ToR on 007's accuracy.



(a) (b)

Figure 10: Algorithm 1 with single failure.



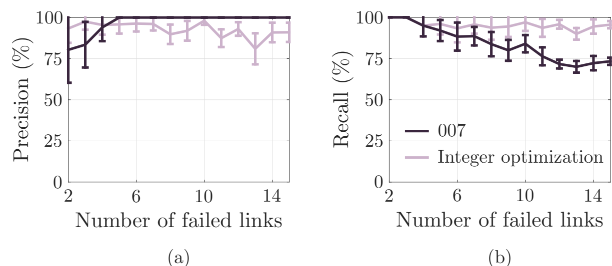
(a) (b)

Figure 11: Impact of link location on Algorithm 1.

Multiple Failures. We heavily skew the drop rates on the failed links. Specifically, at least one failed link has a drop rate between 10 and 100%, while all others have a drop rate in (0.01%, 0.1%). This scenario is one that past approaches have reported as hard to detect [2]. Figure 12 shows that 007 can detect up to 7 failures with accuracy above 90%. Its recall drops as the number of failed links increase. This is because the increase in the number of failures drives up the votes of all other links increasing the cutoff threshold and thus increasing the likelihood of false negatives. In fact if the top k links had been selected 007's recall would have been close to 100%.

6.7 Effects of Network Size

Finally, we evaluate 007 in larger networks. Its accuracy when finding a single failure was 98%, 92%, 91%, and 90% on average in a network with 1, 2, 3, and 4 pods respectively. In contrast, the optimization had an average accuracy of 94%, 72%, 79%, and 77% respectively. Algorithm 1 continues to have Recall



(a) (b)

Figure 12: Algorithm 1 with multiple failures. The drop rates on the links are heavily skewed.

$\geq 98\%$ for up to 6 pods (it drops to 85% for 7 pods). Precision remains 100% for all pod sizes.

We also evaluate both 007 and the optimization's ability to find the cause of per flow drops when the number of failed links is ≥ 30. We observe that both approach's performance remained unchanged for the most part, e.g., the accuracy of 007 in an example with 30 failed links is 98.01%.

7 Evaluations: Test Cluster

We next evaluate 007 on the more realistic environment of a test cluster with 10 ToRs and a total of 80 links. We control 50 hosts in the cluster, while others are production machines. Therefore, the T_1 switches see real production traffic. We recorded 6 hours of traffic from a host in production and replayed it from our hosts in the cluster (with different starting times). Using Everflow-like functionality [3] on the ToR switches, we induced different rates of drops on T_1 to ToR links. Our goal is to find the cause of packet drops on each flow §7.2 and to validate whether Algorithm 1 works in practice §7.3.

7.1 Clean Testbed Validation

We first validate a clean testbed environment. We repave the cluster by setting all devices to a clean state. We then run 007 without injecting any failures. We see that in the newly-repaved cluster, links arriving at a particular ToR switch had abnormally high votes, namely 22.5 ± 3.65 in average. We thus suspected that this ToR is experiencing problems. After rebooting it, the total votes of the links went down to 0, validating our suspicions. This exercise also provides one example of when 007 is extremely effective at identifying links with low drop rates.

7.2 Per-connection Failure Analysis

Can 007 identify the cause of drops when links have very different drop rates? To find out, we induce a drop rate of 0.2% and 0.05% on two different links for an hour. We only know the ground truth when the flow goes through at least one of the two failed links. Thus, we only consider such flows. For 90.47% of these, 007 was able to attribute the packet drop to the correct link (the one with higher drop rate).

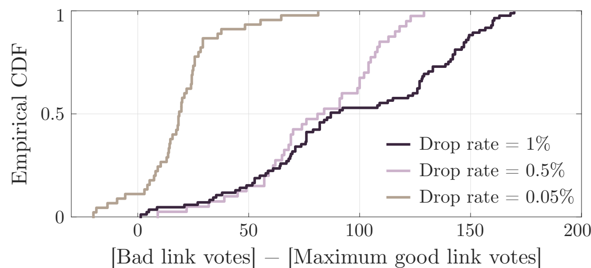


Figure 13: CDF of difference between votes on bad links and the maximum vote on good links.

7.3 Identifying Failed Links

We next validate Algorithm 1 and its ability to detect failed links. We inject different drop rates on a chosen link and determine whether there is a correlation between total votes and drop rates. Specifically, we look at the difference between the vote tally on the bad link and that of the most voted good link. We induced a packet drop rate of 1%, 0.1%, and 0.05% on a T_1 to ToR link in the test cluster.

Figure 13 shows the distribution for the various drop rates. The failed link has the highest vote out of all links when the drop rate is 1% and 0.1%. When the drop rate is lowered to 0.05%, the failed link becomes harder to detect due to the smaller gap between the drop rate of the bad link and that of the normal links. Indeed, the bad link only has the maximum score in 88.89% of the instances (mostly due to occasional lone drops on healthy links). However, it is always one of the 2 links with the highest votes.

Figure 13 also shows the high correlation between the probability of packet drop on a links and its vote tally. This trivially shows that 007 is 100% accurate in finding the cause of packet drops on each flow given a single link failure: the failed link has the highest votes among all links. We compare 007 with the optimization problem in (4). We find that the latter also returns the correct result every time, albeit at the cost of a large number of false positives. To illustrate this point: the number of links marked as bad by (4) on average is 1.5, 1.18, and 1.47 times higher than the number given by 007 for the drop rates of 1%, 0.1%, and 0.05% respectively.

What about multiple failures? This is a harder experiment to configure due to the smaller number of links in this test cluster and its lower path diversity. We induce different drop rates ($p_1 = 0.2\%$ and $p_2 = 0.1\%$) on two links in the cluster. The link with higher drop rate is the most voted 100% of the time. The second link is the second highest ranked 47% of the time and the third 32% of the time. It always remained among the 5 most voted links. This shows that by allowing a single false positive (identifying three instead of two links), 007 can detect all failed links 80% of the time even in a setup where the traffic distribution is highly skewed. This is something past approaches [2] could not achieve. In this example, 007 identifies the true cause of packet drops on each connection 98% of the time.

8 Evaluations: Production

We have deployed 007 in one of our datacenters[2]. Notable examples of problems 007 found include: power

[2] The monitoring agent has been deployed across all our data centers for over 2 years.

$T=0$	$T>0$ & $T \leq 3$	$T>3$	$\max(T)$
69%	30.98%	0.02%	11

Table 1: Number of ICMPs per second per switch (T). We see $\max(T) \leq T_{\max}$.

supply undervoltages [28], FCS errors [29], switch reconfigurations, continuous BGP state changes, link flaps, and software bugs [30]. Also, 007 found every problem that was caught by our previously deployed diagnosis tools.

8.1 Validating Theorem 1

Table 1 shows the distribution of the number of ICMP messages sent by each switch in each epoch over a week. The number of ICMP messages generated by 007 never exceed T_{max} (Theorem 1).

8.2 TCP Connection Diagnosis

In addition to finding problematic links, 007 identifies the most likely cause of drops on each flow. Knowing when each individual packet is dropped in production is hard. We perform a semi-controlled experiment to test the accuracy of 007. Our environment consists of thousands of hosts/links. To find the "ground truth", we compare its results to that obtained by EverFlow. EverFlow captures all packets going through each switch on which it was enabled. It is expensive to run for extended periods of time. We thus only run EverFlow for 5 hours and configure it to capture all outgoing IP traffic from 9 random hosts. The captures for each host were conducted on different days. We filter all flows that were detected to have at least one retransmission during this time and using EverFlow find where their packets were dropped. We then check whether the detected link matches that found by 007. We found that *007 was accurate in every single case*. In this test we also verified that each path recorded by 007 matches exactly the path taken by that flow's packets as captured by EverFlow. This confirms that it is unlikely for paths to change fast enough to cause errors in 007's path discovery.

8.3 VM Reboot Diagnosis

During our deployment, there were 281 VM reboots in the datacenter for which there was no explanation. 007 found a link as the cause of problems in each case. Upon further investigation on the SNMP system logs, we observe that in 262 cases, there were transient drops on the host to ToR link a number of which were correlated with high CPU usage on the host. Two were due to high drop rates on the ToR. In another 15, the endpoints of the links found were undergoing configuration updates. In the remaining 2 instances, the link was flapping.

Finally, we looked at our data for one cluster for one day. 007 identifies an average of 0.45 ± 0.12 links as dropping packets per epoch. The average across all epochs of the maximum vote tally was 2.51 ± 0.33. Out of the links dropping packets 48% are server to ToR links (38% were due to a single ToR switch that was eventually taken out for repair), 24% are T_1-ToR links and 6% were due to T_2-T_1 link failures.

9 Discussion

007 is highly effective in finding the cause of packet drops on individual flows. By doing so, it provides flow-level context which is useful in finding the cause of problems for specific applications. In this section we discuss a number of other factors we considered in its design.

9.1 007's Main Assumptions

The proofs of Theorems 1 and 2 and the design of the path discovery agent (§4) are based on a number of assumptions:

ACK loss on reverse path. It is possible that packet loss on the reverse path is so severe that loss of ACK packets triggers timeout at the sender. If this happens, the traceroute would not be going over any link that triggered the packet drop. Since TCP ACKs are cumulative, this is typically not a problem and 007 assumes retransmissions in such cases are unlikely. This is true unless loss rates are very high, in which case the severity of the problem is such that the cause is apparent. Spurious retransmissions triggered by timeouts may also occur if there is sudden increased delay on forward or reverse paths. This can happen due to rerouting, or large queue buildups. 007 treats these retransmissions like any other.

Source NATs. Source network address translators (SNATs) change the source IP of a packet before it is sent out to a VIP. Our current implementation of 007 assumes connections are SNAT bypassed. However, if flows are SNATed, the ICMP messages will not have the right source address for 007 to get the response to its traceroutes. This can be fixed by a query to the SLB. Details are omitted.

L2 networks. Traceroute is not a viable option to find paths when datacenters operate using L2 routing. In such cases we recommend one of the following: (a) If access to the destination is not a problem and switches can be upgraded one can use the path discovery methods of [2, 31]. 007 is still useful as it allows for finding the cause of failures when multiple failures are present and for individual flows. (b) Alternatively, EverFlow can be used to find path. 007's sampling is necessary here as EverFlow doesn't scale to capture the path of all flows.

Network topology. The calculations in §5 assume a known topology (Clos). The same calculations can be carried out for *any* known topology by updating

the values used for ECMP. The accuracy of 007 is tied to the degree of path diversity and that multiple paths are available at each hop: the higher the degree of path diversity, the better 007 performs. This is also a desired property in any datacenter topology, most of which follow the Clos topology [13, 32, 33].

ICMP rate limit. In rare instances, the severity of a failure or the number of flows impacted by it may be such that it triggers 007's ICMP rate limit which stops sending more traceroute messages in that epoch. This *does not* impact the accuracy of Algorithm 1. By the time 007 reaches its rate limit, it has enough data to localize the problematic links. However, this limits 007's ability to find the cause of drops on flows for which it did not identify the path. We accept this trade-off in accuracy for the simplicity and lower overhead of 007.

Unpredictability of ECMP. If the topology and the ECMP functions on all the routers are known, the path of a packet can be found by inspecting its header. However, ECMP functions are typically proprietary and have initialization "seeds" that change with every reboot of the switch. Furthermore, ECMP functions change after link failures and recoveries. Tracking all link failures/recoveries in real time is not feasible at a datacenter scale.

9.2 Other Factors To Consider

007 has been designed for a specific use case, namely finding the cause of packet drops on individual connections in order to provide application context. This resulted in a number of design choices:

Detecting congestion. 007 *should not* avoid detecting major congestion events as they signal severe traffic imbalance and/or incast and are actionable. However, the more prevalent ($\geq 92\%$) forms of congestion have low drop rates 10^{-8}–10^{-5} [29]. 007 treats these as noise and does not detect them. Standard congestion control protocols can effectively react to them.

007's ranking. 007's ranking approach will naturally bias towards the detection of failed links that are frequently used. This is an intentional design choice as the goal of 007 is to identity high impact failures that affect many connections.

Finding the cause of other problems. 007's goal is to identify the cause of every packet drop, but other problems may also be of interest. 007 can be extended to identify the cause of many such problems. For example, for latency, ETW provides TCP's smooth RTT estimates upon each received ACK. Thresholding on these values allows for identifying "failed" flows and 007's voting scheme can be used to provides a ranked list of suspects. Proving the accuracy of 007 for such problems requires an extension of the analysis presented in this paper.

VM traffic problems. 007's goal is to find the cause of drops on infrastructure connections and through those, find the failed links in the network. In principle, we can build a 007-like system to diagnose TCP failures for connections established by customer VMs as well. For example, we can update the monitoring agent to capture VM TCP statistics through a VFP-like system [34]. However, such a system raises a number of new issues, chief among them being security. This is part of our future work.

In conclusion, we stress that the purpose of 007 is to *explain* the cause of drops when they occur. Many of these are not actionable and do not require operator intervention. The tally of votes on a given link provide a starting point for deciding when such intervention is needed.

10 Related Work

Finding the source of failures in distributed systems, specifically networks, is a mature topic. We outline some of the key differences of 007 with these works.

The most closely related work to ours is perhaps [2], which requires modifications to routers and both endpoints a limitation that 007 does not have. Often services (e.g. storage) are unwilling to incur the additional overhead of new monitoring software on their machines and in many instances the two endpoints are in seperate organizations [4]. Moreover, in order to apply their approach to our datacenter, a number of engineering problems need to be overcome, including finding a substitute for their use of the DSCP bit, which is used for other purposes in our datacenter. Lastly, while the statistical testing method used in [2] (as well as others) are useful when paths of both failed and non-failed flows are available they cannot be used in our setting as the limited number of traceroutes 007 can send prevent it from tracking the path of all flows. In addition 007 allows for diagnosis of individual connections *and* it works well in the presence of multiple simultaneous failures, features that [2] does not provide. Indeed, finding paths only when they are needed is one of the most attractive features of 007 as it minimizes its overhead on the system. Maximum cover algorithms [31, 35] suffer from many of the same limitations described earlier for the binary optimization, since MAX COVERAGE and Tomo are approximations of (3). Other related work can be loosely categorized as follows:

Inference and Trace-Based Algorithms [1, 2, 3, 36, 37, 38, 39, 40, 41, 42] use anomaly detection and trace-based algorithms to find sources of failures. They require knowledge/inference of the location of logical devices, e.g. load balancers in the connection

path. While this information is available to the network operators, it is not clear which instance of these entities a flow will go over. This reduces the accuracy of the results.

Everflow [3] aims to accurately identify the path of packets of interest. However, it does not scale to be used as an always on diagnosis system. Furthermore, it requires additional features to be enabled in the switch. Similarly, [26, 31] provides another means of path discovery, however, such approaches require deploying new applications to the remote end points which we want to avoid (due to reasons described in [4]). Also, they depend on SDN enabled networks and are not applicable to our setting where routing is based on BGP enabled switches.

Some inference approaches aim at *covering* the full topology, e.g. [1]. While this is useful, they typically only provides a sampled view of connection livelihood and do not achieve the type of always on monitoring that 007 provides. The time between probes for [1] for example is currently 5 minutes. It is likely that failures that happen at finer time scales slip through the cracks of its monitoring probes.

Other such work, e.g. [2, 39, 40, 41] require access to both endpoints and/or switches. Such access may not always be possible. Finally, NetPoirot [4] can only identify the general type of a problem (network, client, server) rather than the responsible device.

Network tomography [7, 8, 9, 11, 21, 43, 44, 45, 46, 47, 48, 49, 50] typically consist of two aspects: (i) the gathering and filtering of network traffic data to be used for identifying the points of failure [7, 45] and (ii) using the information found in the previous step to identify where/why failures occurred [8, 9, 10, 21, 43, 49, 51]. 007 utilizes ongoing traffic to detect problems, unlike these approaches which require a much heavier-weight operation of gathering large volumes of data. Tomography-based approaches are also better suited for non-transient failures, while 007 can handle both transient and persistent errors. 007 also has coverage that extends to the entire network infrastructure, and does not limit coverage to only paths between designated monitors as some such approaches do. Work on analyzing failures [7, 21, 43, 45] are complementary and can be applied to 007 to improve our accuracy.

Anomaly detection [52, 53, 54, 55, 56, 57, 58] find when a failure has occurred using machine learning [52, 54] and Fourier transforms [56]. 007 goes a step further by finding the device responsible.

Fault Localization by Consensus [59] assumes that a failure on a node common to the path used by a subset of clients will result in failures on a significant number of them. NetPoirot [4] illustrates why this approach fails in the face of a subset of problems that are common to datacenters. While our work builds on this idea, it provides a confidence measure that identifies how reliable a diagnosis report is.

Fault Localization using TCP statistics [2, 60, 61, 62, 63] use TCP metrics for diagnosis. [60] requires heavyweight active probing. [61] uses learning techniques. Both [61], and T-Rat [62] rely on continuous packet captures which doesn't scale. SNAP [63] identifies performance problems/causes for connections by acquiring TCP information which are gathered by querying socket options. It also gathers routing data combined with topology data to compare the TCP statistics for flows that share the same host, link, ToR, or aggregator switch. Given their lack of continuous monitoring, all of these approaches fail in detecting the type of problems 007 is designed to detect. Furthermore, the goal of 007 is more ambitious, namely to find the link that causes packet drops for each TCP connection.

Learning Based Approaches [4, 64, 65, 66] do failure detection in home and mobile networks. Our application domain is different.

Application diagnosis [67, 68] aim at identifying the cause of problems in a distributed application's execution path. The limitations of diagnosing network level paths and the complexities associated with this task are different. Obtaining all execution paths *seen* by an application, is plausible in such systems but is not an option in ours.

Failure resilience in datacenters [13, 69, 70, 71, 72, 73, 74, 75, 76, 77] target resilience to failures in datacenters. 007 can be helpful to a number of these algorithms as it can find problematic areas which these tools can then help avoid.

Understanding datacenter failures [22, 78] aims to find the various types of failures in datacenters. They are useful in understanding the types of problems that arise in practice and to ensure that our diagnosis engines are well equipped to find them. 007's analysis agent uses the findings of [22].

11 Conclusion

We introduced 007, an always on and scalable monitoring/diagnosis system for datacenters. 007 can accurately identify drop rates as low as 0.05% in datacenters with thousands of links through monitoring the status of ongoing TCP flows.

12 Acknowledgements

This work was was supported by grants NSF CNS-1513679, DARPA/I2O HR0011-15-C-0098. The authors would like to thank T. Adams, D. Dhariwal, A. Aditya, M. Ghobadi, O. Alipourfard, A. Haeberlen, J. Cao, I. Menache, S. Saroiu, and our shepherd H. Madhyastha for their help.

References

[1] Guo, C., Yuan, L., Xiang, D., Dang, Y., Huang, R., Maltz, D., Liu, Z., Wang, V., Pang, B., Chen, H., et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM* (2015), pp. 139–152.

[2] Roy, A., Bagga, J., Zeng, H., and Sneoren, A. Passive realtime datacenter fault detection. In *ACM NSDI* (2017).

[3] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B. Y., et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM* (2015), pp. 479–491.

[4] Arzani, B., Ciraci, S., Loo, B. T., Schuster, A., Outhred, G., et al. Taking the blame game out of data centers operations with NetPoirot. In *ACM SIGCOMM* (2016), pp. 440–453.

[5] Wu, X., Turner, D., Chen, C.-C., Maltz, D. A., Yang, X., Yuan, L., and Zhang, M. NetPilot: Automating datacenter network failure mitigation. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 419–430.

[6] Narayana, S., Sivaraman, A., Nathan, V., Goyal, P., Arun, V., Alizadeh, M., Jeyakumar, V., and Kim, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 85–98.

[7] Zhang, Y., Roughan, M., Willinger, W., and Qiu, L. Spatio-temporal compressive sensing and internet traffic matrices. *ACM SIGCOMM Computer Communication Review 39*, 4 (2009), 267–278.

[8] Ma, L., He, T., Swami, A., Towsley, D., Leung, K. K., and Lowe, J. Node failure localization via network tomography. In *ACM SIGCOMM IMC* (2014), pp. 195–208.

[9] Liu, C., He, T., Swami, A., Towsley, D., Salonidis, T., and Leung, K. K. Measurement design framework for network tomography using fisher information. *ITA AFM* (2013).

[10] Dhamdhere, A., Teixeira, R., Dovrolis, C., and Diot, C. NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *ACM CoNEXT* (2007).

[11] Kompella, R. R., Yates, J., Greenberg, A., and Snoeren, A. C. IP fault localization via risk modeling. In *USENIX NSDI* (2005), pp. 57–70.

[12] Bertsimas, D., and Tsitsiklis, J. N. *Introduction to linear optimization.* Athena Scientific, 1997.

[13] Ports, D. R. K., Li, J., Liu, V., Sharma, N. K., and Krishnamurthy, A. Designing distributed systems using approximate synchrony in data center networks. In *USENIX NSDI* (2015), pp. 43–57.

[14] Microsoft. Windows ETW, 2000. `https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx`.

[15] Hopps, C. E. RFC 2992: Analysis of an Equal-Cost Multi-Path algorithm, 2000.

[16] Patel, P., Bansal, D., Yuan, L., Murthy, A., Greenberg, A., Maltz, D. A., Kern, R., Kumar, H., Zikos, M., Wu, H., et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 207–218.

[17] Liu, H. H., Kandula, S., Mahajan, R., Zhang, M., and Gelernter, D. Traffic engineering with forward fault correction. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 527–538.

[18] Johnson, B. W., Kim, S. H., Leo Jr, E. J., and Lee, D. Link aggregation path selection method, 2003. US Patent 6,535,504.

[19] Gunes, M. H., and Sarac, K. Resolving IP aliases in building traceroute-based internet maps. *IEEE/ACM Transactions on Networking 17*, 6 (2009), 1738–1751.

[20] Institute, I. S. RFC 791: Internet Protocol, 1981. DARPA.

[21] Mysore, R. N., Mahajan, R., Vahdat, A., and Varghese, G. Gestalt: Fast, unified fault localization for networked systems. In *USENIX ATC* (2014), pp. 255–267.

[22] Zhuo, D., Ghobadi, M., Mahajan, R., Phanishayee, A., Zou, X. K., Guan, H., Krishnamurthy, A., and Anderson, T. RAIL: A case for Redundant Arrays of Inexpensive Links in data center networks. In *USENIX NSDI* (2017).

[23] Bernhard, K., and Vygen, J. Combinatorial optimization: Theory and algorithms. *Springer, Third Edition, 2005.* (2008).

[24] Mosek, A. The mosek optimization software. *Online at http://www. mosek. com 54* (2010), 2–1.

[25] Arzani, B. Simulation source codes. Tech. rep., Microsoft Research, 2018. `https://github.com/behnazak/Vigil-007SourceCode.git`.

[26] Tammana, P., Agarwal, R., and Lee, M. Cherrypick: Tracing packet trajectory in software-defined datacenter networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (2015), ACM, p. 23.

[27] Arzani, B., Ciraci, S., Chamon, L., Zhu, Y., Liu, H., Padhye, J., Thau Loo, B., and Outhred, G. 007: Democratically finding the cause of packet drops. *arXiv preprint* (2018).

[28] Arista. Arista eos system message guide. Tech. rep., Arista Networks, 2015. `http://simatinc.com/simatftp/4.14/EOS-4.14.6M/EOS-4.14.6M-SysMsgGuide.pdf`.

[29] Zhuo, D., Ghobadi, M., Mahajan, R., Förster, K.-T., Krishnamurthy, A., and Anderson, T. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 362–375.

[30] Cisco. Cisco bug: Cscut86141 - sfp-h10gb-cu2.255m, hardware type changed to no-transceiver on n3k. Tech. rep., Cisco, 2017. `https://quickview.cloudapps.cisco.com/quickview/bug/CSCut86141`.

[31] Tammana, P., Agarwal, R., and Lee, M. Simplifying datacenter network debugging with pathdump. In *OSDI* (2016), pp. 233–248.

[32] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review* (2008), vol. 38, ACM, pp. 63–74.

[33] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM computer communication review* (2009), vol. 39, ACM, pp. 51–62.

[34] Firestone, D. Vfp: A virtual switch platform for host sdn in the public cloud. In *NSDI* (2017), pp. 315–328.

[35] Kompella, R. R., Yates, J., Greenberg, A., and Snoeren, A. C. Detection and localization of network black holes. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE* (2007), IEEE, pp. 2180–2188.

[36] Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D. A., and Zhang, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review 37*, 4 (2007), 13–24.

[37] Adair, K. L., Levis, A. P., and Hruska, S. I. Expert network development environment for automating machine fault diagnosis. In *SPIE Applications and Science of Artificial Neural Networks* (1996), pp. 506–515.

[38] Ghasemi, M., Benson, T., and Rexford, J. RINC: Real-time Inference-based Network diagnosis in the Cloud. Tech. rep., Princeton University, 2015. https://www.cs.princeton.edu/research/techreps/TR-975-14.

[39] Mahajan, R., Spring, N., Wetherall, D., and Anderson, T. User-level internet path diagnosis. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 106–119.

[40] Liú, Y., Miao, R., Kim, C., and Yuú, M. LossRadar: Fast detection of lost packets in data center networks. In *ACM CoNEXT* (2016), pp. 481–495.

[41] Li, Y., Miao, R., Kim, C., and Yu, M. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI* (2016), pp. 311–324.

[42] Heller, B., Scott, C., McKeown, N., Shenker, S., Wundsam, A., Zeng, H., Whitlock, S., Jeyakumar, V., Handigol, N., McCauley, J., et al. Leveraging SDN layering to systematically troubleshoot networks. In *ACM SIGCOMM HotSDN* (2013), pp. 37–42.

[43] Duffield, N. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory 52*, 12 (2006), 5373–5388.

[44] Kandula, S., Katabi, D., and Vasseur, J.-P. Shrink: A tool for failure diagnosis in IP networks. In *ACM SIGCOMM MineNet* (2005), pp. 173–178.

[45] Ogino, N., Kitahara, T., Arakawa, S., Hasegawa, G., and Murata, M. Decentralized boolean network tomography based on network partitioning. In *IEEE/IFIP NOMS* (2016), pp. 162–170.

[46] Chen, Y., Bindel, D., Song, H., and Katz, R. H. An algebraic approach to practical and scalable overlay network monitoring. *ACM SIGCOMM Computer Communication Review 34*, 4 (2004), 55–66.

[47] Zhao, Y., Chen, Y., and Bindel, D. Towards unbiased end-to-end network diagnosis. *ACM SIGCOMM Computer Communication Review 36*, 4 (2006), 219–230.

[48] Huang, Y., Feamster, N., and Teixeira, R. Practical issues with using network tomography for fault diagnosis. *ACM SIGCOMM Computer Communication Review 38*, 5 (2008), 53–58.

[49] Duffield, N. G., Arya, V., Bellino, R., Friedman, T., Horowitz, J., Towsley, D., and Turletti, T. Network tomography from aggregate loss reports. *Performance Evaluation 62*, 1 (2005), 147–163.

[50] Herodotou, H., Ding, B., Balakrishnan, S., Outhred, G., and Fitter, P. Scalable near real-time failure localization of data center networks. In *ACM KDD* (2014), pp. 1689–1698.

[51] Banerjee, D., Madduri, V., and Srivatsa, M. A framework for distributed monitoring and root cause analysis for large IP networks. In *IEEE SRDS* (2009), pp. 246–255.

[52] Fu, Q., Lou, J.-G., Wang, Y., and Li, J. Execution anomaly detection in distributed systems through unstructured log analysis. In *IEEE ICDM* (2009), pp. 149–158.

[53] Huang, L., Nguyen, X., Garofalakis, M., Jordan, M. I., Joseph, A., and Taft, N. In-network PCA and anomaly detection. In *NIPS* (2006), pp. 617–624.

[54] Gabel, M., Sato, K., Keren, D., Matsuoka, S., and Schuster, A. Latent fault detection with unbalanced workloads. In *EPForDM* (2015).

[55] Ibidunmoye, O., Hernández-Rodriguez, F., and Elmroth, E. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys 48*, 1 (2015).

[56] Zhang, Y., Ge, Z., Greenberg, A., and Roughan, M. Network anomography. In *ACM SIGCOMM IMC* (2005).

[57] Crovella, M., and Lakhina, A. Method and apparatus for whole-network anomaly diagnosis and method to detect and classify network anomalies using traffic feature distributions, 2014. US Patent 8,869,276.

[58] Kind, A., Stoecklin, M. P., and Dimitropoulos, X. Histogram-based traffic anomaly detection. *IEEE Transactions on Network and Service Management 6*, 2 (2009).

[59] Padmanabhan, V. N., Ramabhadran, S., and Padhye, J. Netprofiler: Profiling wide-area networks using peer cooperation. In *IPTPS*. 2005, pp. 80–92.

[60] Mathis, M., Heffner, J., O'Neil, P., and Siemsen, P. Pathdiag: Automated TCP diagnosis. In *PAM* (2008), pp. 152–161.

[61] Widanapathirana, C., Li, J., Sekercioglu, Y. A., Ivanovich, M., and Fitzpatrick, P. Intelligent automated diagnosis of client device bottlenecks in private clouds. In *IEEE UCC* (2011), pp. 261–266.

[62] Zhang, Y., Breslau, L., Paxson, V., and Shenker, S. On the characteristics and origins of internet flow rates. *ACM SIGCOMM Computer Communication Review 32*, 4 (2002), 309–322.

[63] Yu, M., Greenberg, A. G., Maltz, D. A., Rexford, J., Yuan, L., Kandula, S., and Kim, C. Profiling network performance for multi-tier data center applications. In *USENIX NSDI* (2011).

[64] Chen, M., Zheng, A. X., Lloyd, J., Jordan, M., Brewer, E., et al. Failure diagnosis using decision trees. In *IEEE ICAC* (2004), pp. 36–43.

[65] Dimopoulos, G., Leontiadis, I., Barlet-Ros, P., Papagiannaki, K., and Steenkiste, P. Identifying the root cause of video streaming issues on mobile devices.

[66] Agarwal, B., Bhagwan, R., Das, T., Eswaran, S., Padmanabhan, V. N., and Voelker, G. M. NetPrints: Diagnosing home network misconfigurations using shared knowledge. In *USENIX NSDI* (2009), vol. 9, pp. 349–364.

[67] Chen, Y.-Y. M., Accardi, A., Kiciman, E., Patterson, D. A., Fox, A., and Brewer, E. A. Path-based failure and evolution management. In *USENIX NSDI* (2004).

[68] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 74–89.

[69] Liu, J., Panda, A., Singla, A., Godfrey, B., Schapira, M., and Shenker, S. Ensuring connectivity via data plane mechanisms. In *USENIX NSDI* (2013), pp. 113–126.

[70] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Matus, F., Pan, R., Yadav, N., Varghese, G., et al. CONGA: Distributed congestion-aware load balancing for datacenters. *ACM SIGCOMM Computer Communication Review 44*, 4 (2014), 503–514.

[71] Paasch, C., and Bonaventure, O. Multipath TCP. *Communications of the ACM 57*, 4 (2014), 51–57.

[72] Chen, G., Lu, Y., Meng, Y., Li, B., Tan, K., Pei, D., Cheng, P., Luo, L. L., Xiong, Y., Wang, X., et al. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *USENIX ATC* (2016).

[73] Schiff, L., Schmid, S., and Canini, M. Ground control to major faults: Towards a fault tolerant and adaptive SDN control network. In *IEEE/IFIP DSN* (2016), pp. 90–96.

[74] Reitblatt, M., Canini, M., Guha, A., and Foster, N. Fattire: Declarative fault tolerance for software-defined networks. In *ACM SIGCOMM HotSDN* (2013), pp. 109–114.

[75] Kuźniar, M., Perešíni, P., Vasić, N., Canini, M., and Kostić, D. Automatic failure recovery for software-defined networks. In *ACM SIGCOMM HotSDN* (2013), pp. 159–160.

[76] Bodík, P., Menache, I., Chowdhury, M., Mani, P., Maltz, D. A., and Stoica, I. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM* (2012), pp. 431–442.

[77] Wundsam, A., Mehmood, A., Feldmann, A., and Maennel, O. Network troubleshooting with mirror VNets. In *IEEE GLOBECOM* (2010), pp. 283–287.

[78] Gill, P., Jain, N., and Nagappan, N. Understanding network failures in data centers: Measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 350–361.

[79] Arratia, R., and Gordon, L. Tutorial on large deviations for the binomial distribution. *Bulletin of Mathematical Biology 51*, 1 (1989), 125–131.

[80] Feller, W. *An Introduction to Probability Theory and Its Applications*, vol. 1. Wiley, 1968.

[81] Cover, T., and Thomas, J. *Elements of information theory*. Wiley-Interscience, 2006.

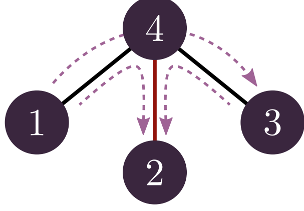


Figure 14: Simple tomography example.

A Network tomography example

Knowing the path of all flows, it is possible to find with confidence which link dropped a packet. To do so, consider the example network in Figure 14. Suppose that the link between nodes 2 and 4 drops packets. Flows 1–2 and 3–2 suffer from drops, but 1–3 does not. A set cover optimization, such as the one used by MAX COVERAGE and Tomo [10, 11], that minimizes the number of "blamed" links will correctly find the cause of drops. This problem is however equivalent to a set covering optimization problem that is known to be NP-complete [23].

B Proofs

Consider a Clos topology with n_{pod} pods each with n_0 ToR switches each with H hosts. Links between tier-0 and tier-1 switches are referred to as *level 1 links* and links between tier-1 and tier-2 switches are called *level 2 links* (see Figure 15). Assume that connection occur uniformly at random between hosts under different ToR switches. Also, assume that link failure and connection routing are independent and that links drop packets independently across links and across packets. To make the derivations clearer, we use calligraphic letter ($\mathcal{A}$) for sets and boldface ($\boldsymbol{A}$) to denote random variables. We also use the notation $[M] = 1,\dots,M$ (see [27] for more detailed proofs).

B.1 Proof of Theorem 1

Proof. First, note that since the number of hosts below each ToR switch is the same, we can consider that traceroutes are sent uniformly at random between ToR switches at a rate C_tH. Also, note that routing probabilities are the same for links on the same level, so the traceroute rate depends only on whether the link is on level 1 or level 2. Given ECMP, the traceroute rates on a level 1 (R_1) and level 2 (R_2) link is given by

$$R_1 = \frac{1}{n_1}C_tH \text{ and } R_2 = \frac{n_0}{n_1 n_2}\frac{n_0(n_{\text{pod}}-1)}{(n_0 n_{\text{pod}}-1)}C_tH.$$

Since n_0 links are connected to a tier-1 switch and n_1 links are connected to a tier-2, the rate of ICMP packets at any links is bounded by $T \leq \max[n_0R_1, n_1R_2]$. Taking $\max[n_0R_1, n_1R_2] \leq T_{\max}$ yields (1). □

B.2 Proof of Theorem 2

We prove a more precise statement of Theorem 2.

Theorem 3. *In a Clos topology with $n_0 \geq n_2$ and $n_{\text{pod}} \geq 1+\max\left[\frac{n_0}{n_1}, \frac{n_2(n_0-1)}{n_0(n_0-n_2)}, 1\right]$, 007 will rank with probability $(1-\epsilon)$ the $k < \frac{n_2(n_0 n_{\text{pod}}-1)}{n_0(n_{\text{pod}}-1)}$ bad links that drop packets with probability p_b above all good links that drop packets with probability p_g as long as*

$$p_g \leq \frac{1-(1-p_b)^{c_l}}{\alpha c_u}, \tag{5}$$

where c_l and c_u are lower and upper bounds, respectively, on the number of packets per connection, $\epsilon \leq 2e^{-\mathcal{O}(N)}$, N is the total number of connections between hosts, and

$$\alpha = \frac{n_0(4n_0-k)(n_{\text{pod}}-1)}{n_2(n_0 n_{\text{pod}}-1)-n_0(n_{\text{pod}}-1)k}. \tag{6}$$

Before proceeding, note that the typical scenario in which $n_0 \geq 2n_2$ and $\frac{n_2(n_0-1)}{n_0(n_0-n_2)} \leq 1$, as in our data center, the condition on the number of pods from Theorem 3 reduces to $n_{\text{pod}} \geq 1+\frac{n_0}{n_1}$.

Proof. The proof proceeds as follows. First, we show that if a link has higher probability of receiving a vote, then it receives more votes if a large enough number of connections (N) are established. We do so using large deviation theory [79], so that we can show that the probability that this does not happen decreases exponentially in N.

Lemma 1. *Let v_b (v_g) be the probability of a bad (good) link receiving a vote. If $v_b \geq v_g$, 007 ranks bad links above good links with probability $1-e^{-\mathcal{O}(N)}$.*

With Lemma 1 in hands, we then need to relate the probabilities of a link receiving a vote (v_b, v_g) to the link drop rates (p_b, p_g). This will allow us to derive the signal-to-noise ratio condition in (5). Note that the probability of a link receiving a vote is the probability of a flow going through the link *and* that a retransmission occurs (i.e., some link in the flow's path drops at least one packet).

Lemma 2. *In a Clos topology with $n_0 \geq n_2$ and $n_{\text{pod}} \geq 1+\max\left[\frac{n_0}{n_1}, \frac{n_2(n_0-1)}{n_0(n_0-n_2)}, 1\right]$, it holds that for $k \leq n_0$ bad links*

$$v_b \geq \frac{r_b}{n_0 n_1 n_{\text{pod}}} \tag{7a}$$

$$v_g \leq \frac{1}{n_1 n_2 n_{\text{pod}}}\frac{n_0(n_{\text{pod}}-1)}{n_0 n_{\text{pod}}-1}\left[(4-\frac{k}{n_0})r_g + \frac{k}{n_0}r_b\right] \tag{7b}$$

where r_b and r_g are the probabilities of a retransmission being due to a bad and a good link, respectively.

Before proving these lemmata, let us see how they imply Theorem 3. From (7) in Lemma 2,

$$r_b \geq \underbrace{\frac{n_0(4n_0 - k)(n_{\text{pod}} - 1)}{n_2(n_0 n_{\text{pod}} - 1) - n_0(n_{\text{pod}} - 1)k}}_{\alpha} r_g \Rightarrow v_b \geq v_g, \tag{8}$$

for $k < \frac{n_2(n_0 n_{\text{pod}} - 1)}{n_0(n_{\text{pod}} - 1)} < n_0$. Thus, in a Clos topology, if $r_b \geq \alpha r_g$ for α as in (6), then $v_b \geq v_g$. However, (8) gives a relation in terms of probabilities of retransmission (r_g, r_b) instead of the packet drop rates (p_g, p_b). Nevertheless, note that the probability r of retransmission during a connection with c packets due to a link that drops packets with probability p is $r = 1 - (1-p)^c$. Since r is monotonically increasing in c, we have $r_b \geq 1 - (1-p_b)^{c_l}$ and $r_g \leq 1 - (1-p_g)^{c_u}$. Using the fact $(1-x)^n \geq 1 - nx$ yields (5). □

Proof of Lemma 1. First, note in a datacenter-sized Clos network, almost every connection has a hop count of 5 (in our datacenter, this happens to 97.5% of connections). We can therefore approximate links votes by assuming all bad votes have the same value.

Since links cause retransmissions independently across connections, the number of votes of a bad link is a binomial random variable $\boldsymbol{B}$ with parameters N, the total number of connections, and v_b, the probability of a bad link receiving a bad vote. Similarly, let $\boldsymbol{G}$ be the number of votes on a good link, a binomial random variable with parameters N and v_g. 007 will correctly rank bad links if $\boldsymbol{B} \geq \boldsymbol{G}$, i.e., if bad links receive more votes than good links. This event contains the event $\mathcal{D} = \{\boldsymbol{G} \leq (1+\delta)Nv_g \cap \boldsymbol{B} \geq (1-\delta)Nv_b\}$ for $\delta \leq \frac{v_b - v_g}{v_b + v_g}$. Using the union bound [80], the probability of 007 correctly identifying bad links obeys

$$\begin{aligned} \mathbb{P}(\boldsymbol{B} \geq \boldsymbol{G}) \geq 1 - \mathbb{P}[\boldsymbol{G} \geq (1+\delta)Nv_g] \\ - \mathbb{P}[\boldsymbol{B} \leq (1-\delta)Nv_b]. \end{aligned} \tag{9}$$

To proceed, bound the probabilities in (9) using the large deviation principle [79], i.e., use the fact that for a binomial random variable $\boldsymbol{S}$ with parameters M and q and for $\delta > 0$ it holds that

$$\mathbb{P}[\boldsymbol{S} \geq (1+\delta)qM] \leq e^{-M\,\mathrm{D_{KL}}((1+\delta)q\|q)} \tag{10a}$$

$$\mathbb{P}[\boldsymbol{S} \leq (1-\delta)qM] \leq e^{-M\,\mathrm{D_{KL}}((1-\delta)q\|q)} \tag{10b}$$

where $\mathrm{D_{KL}}(q\|r)$ is the Kullback-Leibler divergence between two Bernoulli distributions with probabilities of success q and r [81]. The result in Lemma 1 is obtained by substituting (10) into (9). □

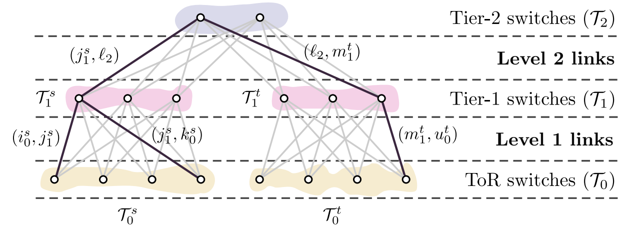


Figure 15: Illustration of notation for Clos topology used in the proof of Lemma 2

Proof of Lemma 2. Let $\mathcal{T}_0$, $\mathcal{T}_1$, and $\mathcal{T}_2$ denote the set of ToR, tier-1, and tier-2 switches respectively (Figure 15). Let $\mathcal{T}_0^s$ and $\mathcal{T}_1^s$, $s = [n_{\text{pod}}]$, denote the tier-0 and tier-1 switches in pod s, respectively. Note that $\mathcal{T}_0 = \mathcal{T}_0^1 \cup \cdots \cup \mathcal{T}_0^{n_{\text{pod}}}$ and $\mathcal{T}_1 = \mathcal{T}_1^1 \cup \cdots \cup \mathcal{T}_1^{n_{\text{pod}}}$. Throughout the derivation, we use subscripts to denote the switch tier and superscripts to denote its pod. For instance, i_0^s is the i-th tier-0 switch from pod s, i.e., $i_0^s \in \mathcal{T}_0^s$, and ℓ_2 is the ℓ-th tier-2 switch (tier-2 switches do not belong to specific pods). We write (i_0^s, j_1^s) for the level 1 link between i_0^s to j_1^s (as in Figure 15) and $r(i_0^s, j_1^s) = r(j_1^s, i_0^s)$ to refer to the probability of link (i_0^s, j_1^s) causing a retransmission.

To get the lower bound in (7a), note that a bad link receives at least as many bad votes as retransmissions it causes. Therefore, the probability of 007 voting for a bad link is larger than the probability of that link causing a retransmission. The bound is obtained by taking the minimum between the probability of a connection going through a level 1 and a level 2 link and that link causing a retransmission, i.e.,

$$v_b \geq \min\left[\frac{1}{n_0 n_1 n_{\text{pod}}}, \frac{1}{n_1 n_2 n_{\text{pod}}} \frac{n_0(n_{\text{pod}} - 1)}{n_0 n_{\text{pod}} - 1}\right] r_b.$$

The assumption that $n_{\text{pod}} \geq 1 + \frac{n_2(n_0 - 1)}{n_0(n_0 - n_2)}$ makes the first term smaller than the second and yields (7a).

In contrast, the upper bound in (7b) is obtained by applying the union bound [80] over all possible ways that level 1 and level 2 link could be voted. Then, finding an adversarial placement of good and bad links that maximizes the probability of the good link receiving a vote, we find that for $n_0 \geq n_2$, $n_{\text{pod}} \geq 2$, and $k \leq n_2$, the worst case is achieved for a level 2 link, yielding (7a). □

Efficient and Correct Test Scheduling for Ensembles of Network Policies

Yifei Yuan Sanjay Chandrasekaran Limin Jia Vyas Sekar
Carnegie Mellon University

Abstract

Testing whether network policies are correctly implemented is critical to ensure a network's safety, performance and availability. Network operators need to test ensembles of network policies using a combination of native and third-party tools in practice, as indicated by our survey. Unfortunately, existing approaches for running tests for ensembles of network policies on stateful networks face fundamental challenges with respect to correctness and efficiency. Running all tests sequentially is inefficient, while naïvely running tests in parallel may lead to incorrect testing results. In this paper, we propose Mikado, a principled scheduling framework for scheduling tests generated by various (blackbox) tools for ensembles of policies. We make two key contributions: (1) we develop a formal correctness criteria for running tests for ensembles of policies; and (2) we design a provably correct and efficient test scheduling algorithm, based on detecting read-write test conflicts. Mikado is open source and can support a range of policies and testing tools. We show that Mikado can generate correct schedules in real-world scenarios, achieve orders of magnitude reduction on the test running time, and schedule tests for thousands of network policies in large networks with 1000+ nodes within minutes.

1 Introduction

Network policies, such as reachability (Can A talk to B?) and dynamic service chaining [14], are implemented via complex network configurations. Testing whether these policies are correctly implemented is becoming increasing important to ensure the security, performance and availability of networks [13, 47, 42, 44, 29, 19].

In recent years, a number of testing techniques have been developed which focus on a variety of network policies, such as ATPG [47], BUZZ [13] and Symnet [42]. Each of these tools can efficiently generate test traces for a single network policy. Our survey indicates, however, that operators have a broad spectrum of policies that the network must implement in practice and thus they need efficient techniques for testing *ensembles* of network policies. Testing such ensembles of policies involves incorporating multiple third-party testing tools (e.g., ATPG for reachability, Symnet for network function correctness, BUZZ for service chaining) as they may offer complementary capabilities and tradeoffs. Looking forward, with emerging trends such as intent-based networking [7, 28, 23], we expect that the policies and testing tools will increase both in diversity and in number.[1]

Unfortunately, testing such ensembles raises fundamental conflicts with respect to *efficiency* and *correctness*. Today, operators often run all tests sequentially in the live network, as indicated by our survey. However, this approach cannot scale up to large-size networks that enforce hundreds or even thousands of network policies. On the other hand, running all tests in parallel may produce incorrect testing results due to interference among the tests. For instance, if a firewall enforces a policy *A* based on connection state, then any test of another policy *B* that changes the relevant connection state will induce incorrect test results for *A* when executing the two tests in parallel (See §2 for more examples).

To achieve both correctness and efficiency, we need a principled framework for *scheduling* such ensembles of test cases. In this respect, strawman solutions such as avoiding specific middleboxes or randomizing the parallelization strategy lead to suboptimal and/or incorrect results; and trivial exhaustive search for optimal schedules may take exponential time. Thus, there are two key challenges in realizing such a framework: 1) how to reason about the correctness of a schedule for ensembles of tests generated by a variety of testing tools; and 2) how to

[1] An alternative to testing is to statically verify networks [37] or synthesizing configurations from intents [43]. However, given the dynamic, stateful nature of processing, the large state space of possible behaviors, testing will still be needed even with these advances to: (1) check correctness for scenarios that cannot be statically verified and (2) to diagnose possible disconnects between the models in the verification/synthesis tools and the real network implementations.

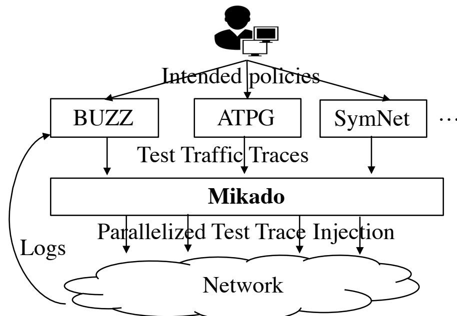


Figure 1: Mikado overview.

efficiently generate optimal schedules for large networks with thousands of nodes and policies.

To this end, we design and implement Mikado, a framework that offers a provably correct and efficient scheduling for tests of ensembles of network policies. As illustrated in Fig. 1, Mikado logically sits between the test generation tools and the network (live or shadow) and is agnostic to the testing tools. Given the test traces for the intended policies, Mikado generates a test schedule that can (near-optimally) reduce the test running time needed to cover those test traces. We note that an alternative design is to generate and schedule test cases in one go. However, given the wide spectrum of network policies, it is unlikely for a single testing tool to support all kinds of policies. Thus, we design Mikado with a blackbox approach to support multiple test generation tools specialized for various policies.

Mikado's design makes two key contributions. First, we develop a formal model for stateful network testing to reason about the correctness of a schedule of test cases (§4). Second, we use our formal model to detect read-write interference among tests and use an efficient graph coloring heuristic on an interference graph to generate provably correct and near-optimal schedules (§5). We also identify opportunities for optimizing the test running time via a combination of random packet header fuzzing and an iterative refinement technique that reduce the likelihood of interference among test traces (§6).

We evaluate Mikado based on both real and synthetic network topologies and policies (§7). We show that: (1) Mikado can generate correct schedules in real-world scenarios; (2) Mikado achieves orders of magnitude reduction on the test running time for thousands of policies on real network topologies; (3) Mikado is scalable to network with 1000+ switches and middlebox and thousands of policies; (4) the proposed extension to reduce interference is both effective and efficient.

2 Motivation

We first describe motivating examples to illustrate the challenges of running tests in stateful networks today. Then we describe our recent survey over network operators, which further confirms these challenges in practice.

2.1 Motivating Examples

Multi-stage IPS: Fig. 2 shows a multi-stage intrusion prevention deployment which consists of a light-weight IPS (IPS1) and a heavy-weight IPS (IPS2). The network operator wants to enforce a set of policies for each department, where traffic from all departments should be sent to IPS1 but traffic from suspicious hosts labeled by IPS1 (e.g., generating 10 bad connections) is sent to IPS2 for payload signature analysis. The topology of the network is at the bottom of Fig. 2 and the policy ensemble is illustrated on top of Fig. 2, which we call policy graph.

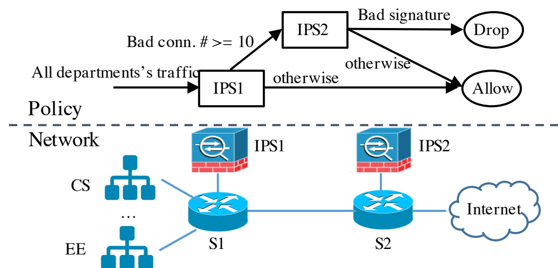


Figure 2: Multi-stage IPS.

To test the intended policies in a live network, a testing tool may generate three test traces for a department, each of which corresponds to a path in the policy graph. For instance, to test the policy ensemble for traffic from the CS department, the tool may generate three test traces from a host H, one containing 6 bad connections and the other two containing 11 bad connections. Further, the second test trace only includes good signatures and the last test trace includes bad signatures. We write $trace_H^6$ to denote the first test trace, $trace_H^{11}$ for the second, and $trace'^{11}_H$ for the third. If the policies are correctly implemented, by injecting the test traces into the network, the network operator would expect to see that packets in the first two traces are successfully forwarded to the Internet and only the latter is directed to IPS2, while $trace'^{11}_H$ is directed to IPS2 and blocked.

Today, operators often need to inject a trace, obtain the results, and then repeat for the next trace. However, such a sequential approach may generate *incorrect* testing results due to the local state maintained by IPS1. For example, injecting $trace_H^6$ after $trace_H^{11}$ will cause $trace_H^6$ to be (mistakenly) directed to IPS2, because of IPS1's stale counter value. An improved sequential testing may wait for a sufficient time T_{to} for the state to expire between each two injections. However, this scheduling can be very inefficient: suppose the time of injecting a trace and waiting for timeout is 30 seconds, running tests for 1,000 policies would take 8 hours!

Injecting all test traces in parallel can reduce the testing time, however, such scheduling could generate incorrect testing results: e.g., injecting $trace_H^6$ and $trace_H^{11}$ together may cause both traces to be directed to IPS2 as

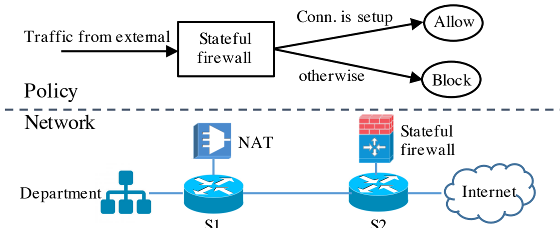


Figure 3: Stateful firewall.

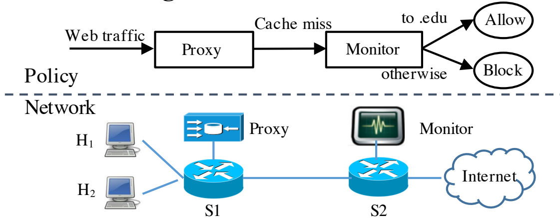


Figure 4: Web monitoring.

the IPS1's counter value reflects the collective connection counts of the traces, not individual ones.

Stateful Firewall: In practice, operators use multiple tools to test different policies. We show that running multiple tools may also lead to incorrect results. Consider the stateful firewall example Fig. 3, where the intended policies are to 1) allow an external IP to access internal network after an internal host is connected to that IP, and 2) block external traffic otherwise.

A tester (e.g., BUZZ [13]) aimed at stateful policies may generate a test trace $trace_{BUZZ}$ that contains a connection from an internal client C_1 to an external server S followed by packets from S to C_1. Meanwhile, a network operator may also use tools like ATPG [47] to test basic reachability. ATPG may generate a test traffic trace $trace_{ATPG}$ from S to a client C_2. However, running the two traces in parallel may produce incorrect testing results: if $trace_{ATPG}$ reaches the firewall after the connection has been set up between C_1 and S by $trace_{BUZZ}$, $trace_{ATPG}$ would go through the firewall while it should have been blocked if it was injected alone. This example demonstrates that simple composition of multiple testing tools may not be safe even if one (ATPG in this example) is aimed at stateless policies such as basic reachability.

Traffic monitoring: Running multiple tests not only raises false alarms, but can also mask configuration bugs in the network. In Fig. 4, a proxy is used to improve the web performance, and a monitoring system is configured to monitor web traffic. The intended policy is to allow `.edu` web traffic for all internal hosts. However, switch S_2 is mistakenly configured to block traffic from H_2.

To test the intended policy, one may run tests that request `xyz.edu` from all hosts. Running the tests in arbitrary order may not uncover the misconfiguration on S_2. For example, consider a test trace containing requests to `xyz.edu` from host H_1, and a second trace containing requests to `xyz.edu` from host H_2. The second trace can uncover the misconfiguration when injected alone to the network, as the request will be blocked by S_2. However, when injected together with the first trace (or immediately after it), the request from H_2 may get a response from the proxy, which caches the content from the first test trace, and the bug is not revealed.

# Policies	%
< 10	19.23%
10-100	57.69%
100-1000	11.54%
>1k	11.54%

# Tools	%
< 5	35%
5-10	50%
>10	15%

Table 1: Number of policies and testing tools.

Summary: We observe key correctness and efficiency challenges in testing an ensemble of policies, and natural strawman solutions face one or both of the challenges. For example, simple isolation heuristics, such as avoiding running tests that access the same middleboxes, will run tests for EE and CS separately for the example in Fig. 2. However, an optimal scheduling can safely run tests from EE and CS together even if they all access IPS1. Exhaustively searching for optimal schedules is not applicable as it takes exponential time and thus incurs prohibitive overhead to the testing workflow.

2.2 Survey on Network Testing

To understand the reality of the aforementioned challenges in network testing today, we conducted a survey in Sept. 2017 among subscribers to the North American Network Operators Group. Among all 30 respondents, 4 manage small networks ($<$ 1k hosts), 6 medium networks (1k-10k hosts), 11 large networks (10k-100k hosts), and 9 very large networks ($>$ 100k hosts). Questions and responses can be found in the link [3]. Here we highlight a few key observations.

Sequential live testing is the dominant testing methodology: When asked the ways of running network tests, 72.97% report running testing on live networks, which is significantly higher than other methodologies (24.32% for emulated network based testing and 2.7% for others). Among those who responded live testing, 86.21% run tests in a sequential way (i.e. run each test case one by one).

Ensembles of policies need to be tested: Table 1 shows that network operators need to test varied number of network policies. While the majority (57.68%) reports policy numbers ranging from 10-100, a significant portion (accounts for 12%) reports the number to be several thousands. A large variety of policies are also reported, ranging from reachability, service chaining, access control, routing, latency/throughput among others.

Multiple tools are used: When asked the number of tools used in testing, all responses report at least 2 different tools. As shown on the right of Table 1, 50% re-

Concerns	%
Test cases may not match the policy intent	30.23%
Testing result maybe incorrect	27.91%
Testing traffic may conflict	25.58%
Testing is slow	16.28%

Table 2: Concerns when running tests

spondents use 5-10 tools for network testing, while some use 10+ tools. While ping, traceroute, iperf are still popular testing tools, we also see responses on using expert-crafted scripts, third-party tools, and custom tools.

Top concerns: Given the large variety and number of policies and testing tools, we hypothesize that correctness of testing results and test conflicts may be one of the concerns for running tests. This is confirmed by our survey. Table 2 lists the concerns network operators report for running network tests under a multi-choice question. A large number (53%) of responses report correctness of testing results and test conflicts as their concerns.

3 Overview

Our goal is to schedule an ensemble of test traces efficient while guaranteeing the correctness of the tests. In other words, we seek to inject as many test traces as possible in parallel such that these traces do not interfere with each other. To this end, we need: (1) a systematic way to reason about the potential for interference of test cases and (2) efficient techniques to identify and schedule non-interfering test cases.

Figure 5 depicts the workflow of our tool Mikado, zoomed in view of Figure 1. The input to Mikado is a set of test traffic traces that are generated from *any* testing tools (e.g., Symnet, ATPG, Buzz), and Mikado outputs an efficient and correct test scheduling plan. To realize the requirements discussed above, Mikado consists of the following components (the dashed are extensions).

Interference Checker: To reason about the potential for interference, we develop a formal model to capture the behavior of the stateful networks (§4). Using this model, Mikado first checks the potential source of interference among the test traces. This offers a provable guarantee that scheduling non-interfering test traces in parallel preserves the testing results as if each test trace was injected on a separate or isolated network.

Trace Scheduler: Base on the pairwise interference relations, Mikado builds an interference graph, where each node in the graph corresponds to a test trace, and an edge connects two interfering traces. Mikado then uses a graph coloring algorithm to generate an optimal scheduling of the test traces by assigning each node in the interference graph a color such that two end nodes of an edge are assigned different colors.

Given a colored interference graph, Mikado runs all tests in k rounds, where k is the number of colors on the interference graph. In the i-th round, Mikado injects

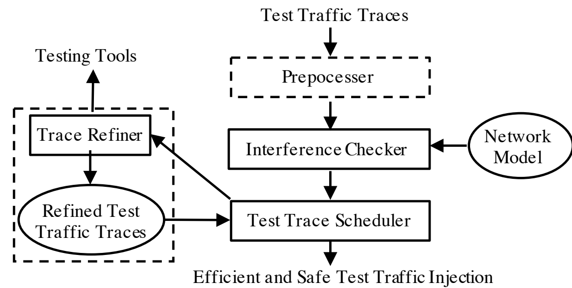


Figure 5: Mikado architecture.

all test traces with the i-th color and reports the executing results to the testing tools. Between two consecutive rounds, Mikado waits for a sufficient amount of time in order for all state to reset.

Preprocessor and Trace Refiner: Optionally, Mikado can further improve the scheduling by random packet fuzzing and test traces refinement.

Before checking interference, Mikado can run heuristic-based preprocessing to reduce the chance of interference among test traces. This process rewrites the test traces using randomly selected values for given fields according to the policy being tested.

After obtaining a correct schedule, Mikado can further refine the test traces to reduce the number of edges in the interference graph, and thus reduce the number of colors to color the refined interference graph. In particular, given a trace that is interfering with a set of traces, Mikado reruns the testing tool for that trace to generate another test trace which is not interfering with the set of traces. For this purpose, Mikado automatically generates a configuration file for the testing tool, without modifying the internal logic of the testing tool.

Illustrative Example: We use the multi-stage IPS example from §2 to illustrate the end-to-end workflow. For brevity, we do not discuss the preprocessing here and only consider the testing of two policies for each department. Suppose $trace_H^6$ and $trace_H^{11}$ are the two traces generated for the CS department and $trace_{H'}^6$ and $trace_{H'}^{11}$ are the two traces for the EE department. First, the interference checker automatically detects the interferences, and builds the interference graph as shown in the left subfigure in Figure 6, where circular nodes correspond to $trace_H^6$ and $trace_H^{11}$, and rectangular nodes correspond to test traces for the EE department. As discussed in §2, $trace_H^6$ and $trace_H^{11}$ cannot be injected together, and thus there is an edge in the interference graph between them. Second, the trace scheduler colors each node with the goal to minimize the number of colors. An example coloring is shown in the middle subfigure. Based on this colored interference graph, Mikado can safely inject blue traces (i.e., $trace_H^6$ and $trace_{H'}^6$) in the first round, and then inject the red traces (i.e., $trace_H^{11}$ and $trace_{H'}^{11}$) in the second round. Additionally, the trace refiner can be used to refine a test trace to further reduce the interference.

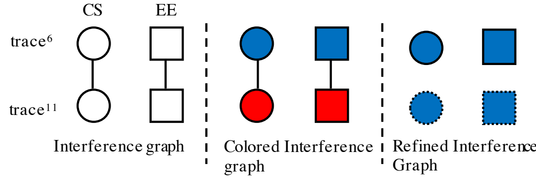


Figure 6: Example run of Mikado on multi-stage IPS.

For instance, to reduce the interference between the two circular nodes, the refiner generates a configuration file that asks the testing tool to generate another test trace that is not from H to refine the trace $trace_H^{11}$. If the testing tool succeeds in doing so, the old node will be replaced with the newly generated trace in the interference graph, and edges from the old nodes are removed. The right subfigure illustrates the final results, where the nodes with dashed line are the refined traces. After the refinement, all test traces can be safely injected in a single round.

4 Problem Formalization

Before we design any tool for scheduling, first we need a formal basis for reasoning about the correctness of a schedule. To this end, we develop a formal notion of correctness and define the test scheduling problem.

4.1 Stateful Network Model

We extend prior work on formal network modeling used in defining stateless networks (c.f. [22, 40]) to model the stateful behaviors we encountered in §2. This model provides a necessary building block for our test correctness definition and proofs for our scheduling algorithm. Our model may also be used in providing semantics to other network testing and verification problems. Note that this model abstracts concrete network functions and thus subsumes models in BUZZ [13] and Symnet [42]. We currently do not model multicast and time.

Syntax. The syntactic constructs used for our model are summarized in Figure 7. We write *pkt* to denote network packets, which are bitstrings. We model locations in the network as ports, denoted p. We use natural numbers n to represent concrete network locations. A special port *exit* models all locations outside the network of concern. To model dropped packets we include a special port *drop*. A located packet, denoted *lp*, is a pair of a port and a packet. (p, pkt) means that the packet *pkt* is at port p waiting to be processed. Packets at port *exit* are the ones that have left the network. All dropped packets are at port *drop*. Multiple packets can co-locate at one port. A packet queue, denoted Q, maps each port p to a list of packets located at p.

Each state in a stateful network (e.g., connection state remembered by a firewall) is labelled with a unique state location, denoted s. We assume s is drawn from a finite set of location symbols Σ. A state map, denoted

Packet	pkt	::=	$[b_1 \dots b_l]$
Port	p	::=	$n \mid exit \mid drop$
Located Packet	lp	::=	(p, pkt)
Port Queue	Q	$\in$	$Port \to Packet\ list$
State. Loc.	s	$\in$	Σ
State Map	M	$\in$	$StateLoc \to Value$
Network Env.	E	::=	(Q, M)
Topo. Func.	ρ	$\in$	$Port \to Port$
Trans. Func.	δ	$\in$	$StateMap \times LocatedPkt$ $\to StateMap \times LocatedPkt$
Configuration	C	::=	(ρ, δ)
Network State	S	::=	(E, C)
Network run	r	::=	$E_1 \xrightarrow{lp/lp'}_C \cdots E_k$
Observation	o	::=	$[lp_1/lp_1; \cdots; lp_n/lp'_n]$

Figure 7: Stateful network model: syntax

M, maps each state location to a value, drawn from a finite set of values *Value*. Intuitively, the state locations represent the variables and data structures that are internally maintained by stateful middleboxes and SDN controllers. We write E to denote network environments, which are pairs of the packet queue and the state map. We write M_{init} to denote the initial state map. M_{init} maps each state location to its pre-defined initial value. The initial port queue, Q_{init}, maps each port to an empty list. We call (Q_{init}, M_{init}) an *initial environment*. A *terminating environment* is one where the port queue is empty for all port p s.t. $p \notin \{exit, drop\}$.

The topology of the network is determined by connections between ports. We use a function ρ to map a port p to another port p', where packets at p are sent to. The process of transforming and forwarding packets are modeled as transfer functions, denoted δ. A transfer function takes as arguments a pair of a state map and a located packet and returns a pair of a new state map and a new located packet. A configuration of the network is a pair (ρ, δ), which include the topology and transfer function of the network. A network state S is a pair of the environment E the configuration C.

Operational semantics. As packets are processed and traverse the network, network states change and affect the processing of future packets. We define small-step operational semantics to model the transitions of network states. We write $E \xrightarrow{lp/lp'}_C E'$ to denote the transition from E to E' using configuration C. Here, *lp* and *lp'* above the arrow denote the located packet processed at the transition and the resulting located packet respectively. We assume that the configuration C does not change during the testing of policies in the network.

The only transition rule NET-TRANS (shown below) states that if the first packet at port p is *pkt*, the transition function changes the located packet (p, pkt) to (p', pkt') and the state map from M to M', and p' is connect to p'', then after *pkt* at p is processed, the state map is M', and the port queue is updated to reflect that *pkt* is no longer

at p and pkt' is added to port p''.

$$\textsc{Net-Trans}\quad \frac{\begin{array}{c} pkt = head(Q(p)) \\ \delta(M,(p,pkt)) = (M',(p',pkt')) \qquad \rho(p') = p'' \\ Q' = Q[p'' \mapsto (Q(p'')++[pkt'])][p \mapsto tail(Q(p))] \end{array}}{(Q,M) \xrightarrow{(p,pkt)/(p',pkt')}_{(\rho,\delta)} (Q',M')}$$

We call a sequence of network transitions a *run*, denoted r. All transitions in a run r from the state (E,C) use C as the configuration. A *complete run* is a run, whose last environment is a terminating one. Given a network state (E,C), we write $\mathscr{R}(E,C)$ to denote the set of complete runs starting from (E,C). We sometimes omit the configuration C from runs as it never changes during the testing and is often clear from the context.

4.2 Correctness of Network Tests

We formalize the correctness criteria for network testing and define the test scheduling problem.

Network testing via packet trace. When testing policies, a network operator first generates a sequence of located packets (using existing test generation tools), then injects the packets into the network, and finally makes a judgement of whether the policy is violated based on observations derived from a complete run of the network.

We call the sequence of located packets $(lp_1,\ldots,lp_m)$ generated by the testing generation tools a *test trace*, denoted t. Injecting a trace t into a network state with port queue Q results in a new port queue where each located packet in t is enqueued at the appropriate port. A test always runs from an initial environment, so we write $queue(t)$ to denote the port queue resulted from injecting t to Q_{init}. Executing the test trace corresponds to generating a complete run from the environment $(queue(t),M_{init})$. One subtlety is that the order in which packets in t are processed matters (e.g., reply to a request can only happen after the request). Our model takes care of this by defining valid runs that comply with the injection order requirement. Given a run r in $\mathscr{R}(queue(t),M_{init})$, we say r is a valid test run w.r.t. t and an ordering requirement $\varphi(t)$, r satisfies $\varphi(t)$. Different test traces have different ordering requirements based on protocol conventions. We assume that such requirements are given as input; our approach and algorithm is agnostic to the specific type of ordering requirement.

We define the observation of a run, written $\mathscr{O}(r)$, as the sequence of pairs of located packets. Given a run r where $r = S_0 \xrightarrow{lp_{i1}/lp_{o1}}_C S_1 \cdots \xrightarrow{lp_{ik}/lp_{ok}}_C S_k$, the observation $\mathscr{O}(r)$ is simply $[lp_{i1}/lp_{o1};\cdots;lp_{ik}/lp_{ok}]$. This corresponds to the trace routes that the network operator sees.

Correct network test. Given a set of test traces $\{t_1,...,t_k\}$ for an ensemble of policies $(pol_1,\ldots,pol_k)$, we have shown in §2 that test traces may interfere with each other if they are allowed to execute concurrently. We provide a strong *correctness* definition of running multiple tests based on a tester's observation of each test.

Intuitively, if it is correct to concurrently run a set of test traces, the observation of each test trace should be the same as the test trace running alone. For the IPS example in §2, the observation of injecting $trace^6_H$ and $trace^{11}_H$ together for $trace^6_H$ may be different from that of injecting $trace^6_H$ alone. In the former case, packets in $trace^6_H$ may be sent to IPS2, while in the latter case all the packets in the trace land in the *exit* port.

To formalize this intuition, we define a projection of a run given a packet trace, written $r\downarrow_t$, as the observation containing only the located packets that are related to t, that is, either it is in t, or it is (transitively) transformed from a located packet in t. The projection of a run r from $(queue(t),M_{init})$ given t is $\mathscr{O}(r)$. Then the correctness definition can be formalized below:

Definition 1 (Correctness) *Given a set $T = \{t_1,\cdots,t_n\}$ of test traces, we say that it is correct to schedule T (T is correct for short) if $\forall r \in \mathscr{R}(queue(\cup_{i=1}^n t_i),M_{init})$ s.t. r is valid w.r.t. each $\varphi_j(t_j)$ where $j \in [1,n]$, $\forall i \in [1,n]$, $\forall r' \in \mathscr{R}(queue(t_i),M_{init})$, $r\downarrow_{t_i} = \mathscr{O}(r')$.*

Test scheduling problem. Given an ensemble of test traces, a natural scheduling is to run all tests in sequential rounds. In each round, we want to run as many tests in parallel as possible, in order to reduce the total runtime of testing. Between two rounds, we can wait sufficiently long for the network state to reset so the execution of each round starts from a clean (the initial) state.

Definition 2 (Correct scheduling) *Given a set of test traces T generated for ensembles of policies by a set of testing tools, a* correct schedule *of T is a partition $\{T_1,..,T_k\}$ of T, such that running each T_i is correct.*

We assume that a round of testing takes roughly the same time which is much less than the waiting time between rounds. Therefore, our goal is to correctly minimize the number of rounds and the *test traces scheduling problem* is defined as follows. That is, given a set of test traces T, the *test trace scheduling problem* seeks a correct schedule with minimal number of rounds.

5 Test Traffic Scheduling

A naïve solution to the test trace scheduling problem is enumerating all possible partitions and then checking whether each set of traces in the partition is correct using Definition 1. However, this is extremely inefficient. First, the number of partitions is exponential to the number of test traces in the set. Second, checking whether a set of test traces is correct directly by comparing projections of every possible run of the composed test traces is also prohibitively expensive. For instance, there are

$\frac{(2m)!}{m!m!}$ possible complete runs of a one-port network and two test traces of size m; and $n!$ runs for a one-port network and n traces of size 1.

Next, we outline our key insights (§5.1) and then present our algorithm and discuss the algorithm's time complexity and correctness guarantees (§5.2).

5.1 Key Insights

Even though checking correctness directly may be complex, we can solve an approximation of the problem much more efficiently. For our three examples from §2, it is not too hard to see why test traces may interfere with each other. The main cause of the problem is that concurrently executing test traces do not preserve the values of each other's state locations that determine their behavior. If we are able to identify the conditions under which test traces may interfere with each other, we can solve the scheduling problem by selecting test traces that do not interfere with each other to be scheduled together.

First, we observe that by examining a run of a test trace, we can identify the fragment of the state map that is key to decide how packets are processed by the network. (Recall from §4.1, a state map M maps each state in the network to a value.) For instance, the value of the bad connection counter of IPS1 (denoted s_H) decides how packets in $trace^6_H$ are processed. Changing the value of s_H changes the observation of the test trace and, more importantly, changing the value of other state locations (e.g., the bad connection counter for host H_1) in the state map will not affect how packets in $trace^6_H$ are processed.

A natural corollary of this observation is that given two test traces, we can determine whether they interfere with each other by examining the state maps that decide the behavior of each trace from independent runs of the test trace. For the IPS example, let $trace^6_{H1}$ be the test trace containing 6 bad connections from host *H1* and $trace^{11}_{H2}$ be the test trace containing 11 bad connections from host *H2*. They modify and depend on two different state locations in IPS1 and thus, don't interfere with each other.

Finally, we observe that instead of considering arbitrary partitions, we can construct a correct schedule from pair-wise noninterfering test traces as none of the test traces interfere with each others' key state maps. Those state maps remain the same as an independent run of a test trace, and therefore, result in the same observation.

Given these observations, we can encode the correct scheduling problem as a graph coloring problem where the nodes correspond to test traces and the edges connect interfering test traces (the lack of edges between two nodes implies compatibility between the two traces). Then, we can use an efficient graph coloring algorithm to generate a near-optimal coloring of the graph, which corresponds to a near-optimal correct schedule: nodes of the same color can be scheduled together.

Algorithm 1 Correct scheduling

1: **function** GEN_INTF_GRAPH(T, C, M_{init})
2: $G \leftarrow \{\}$
3: $(\rho, \delta) \leftarrow C$
4: **for** each trace $t_i \in T$ **do**
5: create a node v_{t_i} for t_i in graph G
6: $r_i \leftarrow$ a run from $(Q^{\{t_i\}}, M_{init})$ under C
7: **for** each pair of traces $t_1, t_2 \in T$ **do**
8: $(S^1_0 \xrightarrow{lp^1_0/..} \cdots \xrightarrow{lp^1_k/..} S^1_{k+1}) \leftarrow r_1$
9: $(S^2_0 \xrightarrow{lp^2_0/..} \cdots \xrightarrow{lp^2_l/..} S^2_{l+1}) \leftarrow r_2$
10: let M^a_b be the state map for S^a_b
11: **if** there exist $i \in [1, \ldots, k+1], j \in [0, \ldots, l]$ and s, such that $s \in dtMap(\delta, M^2_j, lp^2_j)$, $M^1_i(s) \neq M^2_j(s)$, and $M^1_{i-1}(s) \neq M^1_i(s)$ **then**
12: $G \leftarrow G \cup (v_{t_1}, v_{t_2})$
13: **return** G
14:
15: **function** SCHEDULE(T, C, M_{init})
16: $G \leftarrow$ GEN_INTF_GRAPH(T, C, M_{init})
17: $G_c \leftarrow$ GRAPH_COLORING(G)
18: **for** each color i in G_c **do**
19: $T_i \leftarrow$ nodes in G_c of color i
20: **return** $(T_1, \cdots, T_k)$

5.2 Algorithm

The main function of our algorithm (Alg. 1) is SCHEDULE (lines 15-20), which takes a set of test traces T, a network configuration C, and the initial state map M as input and returns a partition of T. The SCHEDULE function calls GEN_INTF_GRAPH to generate the interference graph of T and GRAPH_COLORING to color the graph. The latter can be any efficient coloring algorithm, which we omit. The output partition corresponds to a schedule of the test traces (line 20).

The GEN_INTF_GRAPH function creates a node in the graph for all traces in T (line 5). The edges are supposed to connect two nodes representing test traces that interfere with each other. The key is to decide whether two test traces interfere (not compatible) with each other, which relies on the following two functions: $dtMap(\delta, M, lp)$ and $upd(\delta, M, lp)$. At a highlevel, $dtMap(\delta, M, lp)$ returns a state map M' containing a subset of the mappings in M that determines the result of $\delta(M, lp)$. $upd(\delta, M, lp)$ returns a state map M' that maps the subset of state locations in the domain M that are updated by the transition $\delta(M, lp)$ to new values.

For each concrete transition function δ, it is straightforward to define $dtMap(\delta, M, lp)$. Using the stateful firewall example from §2, the state map remembering whether there exists a prior connection from a host within

the department should be returned by this function. We require that all instances of $dtMap(\delta, M, lp)$ return the *determinant state map* of the transition from M and lp under δ, formally defined below. Intuitively, state locations that are in the determinant state map uniquely determines the result of $\delta(M, lp)$; state locations that are not in the determinant state map are allowed to be altered by other test traces without changing the behavior of the current test trace. Use the stateful firewall example again, other state locations such as the state of TCP connections are not in the determinant state map and can be altered by other test traces while keeping the observation of the current test trace the same.

We write $diff(M_1, M_2)$ to denote the set of state locations that are in the domain of M_1 and M_2 and mapped to different values by M_1 and M_2. The determinant state map of the transition from M and lp under δ is M_d if we construct another state map M' by changing the values that state locations in M but not in M_d are mapped to, the transition from M' and lp generates the same located packet ($lp_1 = lp_2$), and the resulting state maps M_1 and M_2 only differ at state locations that are not updated by the transition and not in the determinant state map. The last condition essentially forces updates to the state maps to be determined by M_d as well.

Definition 3 (Determinant state map) *We say a state map M_d determines (is the determinant state map of) the transition from M and lp under δ if $M = (M_d, M_n)$ and for all M'_n s.t. $\mathsf{dom}(M_n) = \mathsf{dom}(M'_n)$, let $M' = (M_d, M'_n)$, $(M_1, lp_1) = \delta(M, lp)$, $(M_2, lp_2) = \delta(M', lp)$, it is the case that $lp_1 = lp_2$ and $diff(M_1, M_2) \subseteq \mathsf{dom}(M') \setminus \mathsf{dom}(upd(\delta, M, lp) \cup upd(\delta, M', lp))$.*

To build the interference graph, the algorithm executes each test trace t_i independently and stores the run (lines 4-6). Only one run is needed and we pick the one where a located packet in t_i is only processed when the located packet before it has left the network. We call such a run a *sequential* run, written $R_{seq}(queue(t_i), M)$. The if condition on line 11 checks whether a run r_1 contains a state location s that determines the execution and run r_2 writes to s with a different value from the one that determines the transition in r_1. If so, then r_1 and r_2 interfere with each other. This is because the modification by r_2 could change the behavior of r_1. This is a conservative check.

Let us revisit the example shown in Figure 6. We explain how our algorithm detects the interference between $trace^6_H$ and $trace^{11}_H$. The middlebox IPS1 maintains a state of the number of bad connections for each host. Consider both traces, the determinant state map $dtMap(\delta, M, lp)$ and $upd(\delta, M, lp)$ are both the bad connection counter for H. We write s_H to denote this state location. For the run of $trace^6_H$, the determinant state maps are $s_H \mapsto 0$, to $s_H \mapsto 6$. The update state maps for the run of $trace^{11}_H$ are $s_H \mapsto 0$, to $s_H \mapsto 11$. Obviously, the updates by the second trace don't always agree with the determinant state maps of the first trace. Condition on line 11 of Algorithm 1 is true. Therefore, the two traces are interfering with each other.

We prove that Algorithm 1 is correct (Theorem 1), and detailed proofs can be found in Appendix.

Theorem 1 (Correctness)
SCHEDULE(T, C, M_{init}) *is a correct schedule.*

6 Extensions to Basic Algorithm

So far, we have discussed how to schedule the test given a set of test traces for ensembles of policies. As we see in previous examples, testing tools may generate interfering test traces for ensembles of policies. However, it is possible that the cause is not that the policies conflict, but the optimization heuristics that the testing tools employ to improve the efficiency of test trace generation. For instance, BUZZ attempts to pick concrete field values (same values across policies) for test traces to reduce the number of symbolic variables. In this section, we investigate the opportunities to guide test trace generation to further improve scheduling efficiency.

6.1 High-level Idea

Recall the multi-stage IPS example in §2. The three test traces ($trace^6_H$, $trace^{11}_H$, and $trace'^{11}_H$ for each policy path) interfere since they share the same counter on IPS1 for the source H. Intuitively, by altering the source of each test trace, we can obtain test traces for each policy path while avoiding the interference. For example, $trace^6_{H1}$, $trace^{11}_{H2}$, and $trace'^{11}_{H3}$ are three test traces for each policy path and can be safely injected in parallel.

To generalize this intuition, we identify a set of *influencing fields* for a located packet lp (in a test trace t_1) given a test trace t_2 (denoted $I_{lp}(t_2)$). The property of $I_{lp}(t_2)$ is that by altering values for some fields in it, it could be possible to avoid the interference between t_1 and t_2 related to lp. For instance, the set of influencing fields for packets in $trace^6_H$ includes the source, since changing the sources in $trace^6_H$ from H to $H1$, interference with other test traces are avoided.

The precise fields in $I_{lp}(t)$ may require involved static analysis of the model and the test trace. Here, we propose two heuristics for identifying $I_{lp}(t)$ and new field values: one is to fix the set to commonly used fields such as the 5-tuples and randomly select the value of them; the other is to leverage results learned from the interference analysis (Alg. 1). We discuss each in detail next.

6.2 Random Packet Fuzzing

For flow-based policy testing (e.g., flow-based service chaining [39, 13]), Mikado employs a light-weight preprocessing for the test traces to reduce the chance of

interferences. This preprocessing picks random values from the flow space specified in the policy and rewrites the 5-tuple of each packet in the test trace using these random values. In addition, to preserve the flow semantics, this preprocessing ensures that fields with the same original values will be substituted with the same values.

This heuristic does not ensure the coherence of the test trace and the policy to be tested in general, as it may pick wrong fields and field values. However, as we will see in §7, this heuristic works reasonably well in practice for the targeted policies.

6.3 Test Trace Refinement

Mikado also employs a more rigorous analysis to *refine* test traces using information obtained from the initial scheduling process. The refinement process uses concrete runs of test traces to guide the selection of $I_{lp}(t)$ and the new values for fields in $I_{lp}(t)$. This process aims to tell the test generation tool not to use values that are known to cause interference from previous analysis.

In the rest of this section, we first describe how to refine t to be non-interfering with all traces in T, followed by the algorithm that refines given correct schedules.

Test Generation with Configurations. Given a testing tool A, let $GenTest_A(pol, N, Config)$ denote the test traces generated by A for policy *pol* and network N under the tool configuration *Config*. Different tools can be tuned differently. One can treat *Config* as constraints on packet header fields for the test traces to be generated. Given the sequential run r_1 and r_2 obtained from the scheduling process for test trace t_1 and t_2 respectively, we first apply an analysis to identify $I_{lp}(t_2)$ for all $lp \in t_1$, which are essentially the fields in *lp* that are used as meaningful input in the transition function δ. For instance, the source address in the multi-stage IPS example. This is more precise than fixing a set of fields a priori.

Next, given the set of influencing fields $I_{lp}(t_2)$ for all $lp \in t_1$, we generate the following constraint $CS(t_1,t_2)$: $\bigwedge_{lp_i \in t_1} \bigvee_{f \in I_{lp_i}(t_2)} lp'_i(f) \neq lp_i(f)$. That is, every packet lp'_i in the refined trace by A should not have values exactly the same for fields in $I_{lp_i}(t_2)$. This constraint will be translated to suitable configuration files that the test generation tool understands (see §7). Similarly, to refine a trace t_1 for a set of traces T, the configuration is the constraint $CS(t_1,T) = \bigwedge_{t_2 \in T} CS(t_1,t_2)$. For instance, for the multi-stage IPS example, if we refine $trace_H^{11}$, the constraint is $\bigwedge_i src_i \neq H$, where src_i is the source address of the i-th packet in the refined trace.

With the above settings, we use Algorithm 2 to refine a trace t for a policy P, such that the refined trace is non-interfering with all traces in T. In the algorithm, we try to refine t for at most *max_num_try* times. In each try, we invoke $GenTest_A$ to generate a possible trace t'. We return t' if it is non-interfering with all traces in T, otherwise, we invoke $GenTest_A$ again for another trace that is not t'.

Algorithm 2 refineTrace(t, T)

1: $CS = CS(t, T))$
2: **for** $i = 1$ to *max_num_try* **do**
3: **if** $GenTest_A(P, N, CS)$ generates a new trace t' **then**
4: **if** t' is non-interfering with T **then return** t'
5: **else** $CS = CS \wedge \{t'' \neq t'\}$ // t'' is the next generated trace
6: **else return** FAIL
7: **return** FAIL

Refinement Algorithm. Given a correct schedule for a set of policies $\{T_1, .., T_k\}$ as input, our refinement algorithm outputs a refined correct schedule for the policies with potentially a smaller number of injection rounds. At a highlevel, our refinement algorithm takes a greedy heuristic and iterates all rounds T_i, starting from the round with least number of traces. Then it attempts to refine all traces in T_i using the algorithm described above. If all traces in T_i can be refined, we remove T_i and obtain a correct schedule with fewer injection rounds. We omit the details of the algorithm in interest of brevity.

7 Evaluation

We evaluate Mikado via a testbed-based emulation and large-scale simulations and show that Mikado: (1) is able to detect test interferences in a range of scenarios and generate *correct* schedule; (2) achieves orders of magnitude reduction in the test running time compared with alternative test scheduling mechanisms; (3) is scalable to networks with 1000+ switches/middleboxes and thousands of policies.

Implementation: We implement a prototype of Mikado in Python. We consider four testing tools in our framework, namely ATPG [47], BUZZ [13], Pingmesh [19], and Symnet [42]. For ATPG and BUZZ, we reuse their constructs for routing tables and middleboxes; and for Symnet, we manually encoded the middleboxes that are not in the code repository in their language SEFL. To support the refinement extension, we also implement a light-weight helper function (≈100 LoC each tool) that translates the generated constraints to each tool's configuration (e.g., Z3 [9] formulas for Symnet). All experiments are conducted on a server with 20 cores (2.8Ghz) and 128GB RAM.

7.1 Validation

End-to-end correctness. We first validate Mikado's correctness in a variety of use cases. On our testbed, we emulate hosts and (software) middleboxes as separate KVM-based virtual NFs, connected with OpenVSwitch.

We use OpenDayLight [33] as the controller for all emulated networks.

• To emulate the multi-IPS scenario in Fig. 2, we use Snort as both IPS1 and IPS2, and use three hosts to emulate the departments and the Internet. We configure IPS1 to enforce two policies: 1) forwarding a host's traffic to IPS2 if the host issues more than 5 connections in a minute; 2) otherwise the traffic is sent to the Internet. We run BUZZ to test the two policies in parallel, and log the traffic at the interface of IPS2 to check the testing results. We run the experiment 100 runs, and BUZZ reports violation of policy 1) in 80 runs. All reported policy violations are validated to be false positives: the root cause is that BUZZ injects traffic from a single host for two policies, and causes conflicts.

• For the scenario in Fig. 3, we use iptables as both the NAT and the stateful firewall, and use Symnet and ATPG to test the connectivity between the Internet and the department in parallel. In all 100 runs, the security policy on the stateful firewall was reported violated. However, we verify that the testing results are false positives: test cases interfere on the stateful firewall.

• We use a simple "blue team-red team" test to simulate the scenario in Fig. 4. We use Squid as the proxy and Snort as the monitor device to emulate the network. To bind proxy's requests with the origin hosts, we use FlowTag [14] to configure the network. One student ("red") intentionally misconfigures the switch to drop web requests from H2, and the other student ("blue") use BUZZ to test HTTP connectivity of each host. Of 10 times of this experiment, the blue team could only uncover the bug in 5 times.

In all three scenarios, Mikado can successfully detect the interference among the generated tests, and correctly schedule the tests in separate runs. We repeat the three experiments with the schedule from Mikado, and no false positives or negatives are produced.

Interference detection. Next, we evaluate Mikado's capability to detect test interference in practice. We consider two popular types of network policies in our survey: reachability and service chaining policies. To collect real-world service chaining policies, we conduct a survey from a set of industrial and academic sources [38, 14, 13, 44, 20, 1, 2, 27, 30], and build a library of 38 service chaining policy templates. The number of network functions on each service chain ranges from 1 to 5, and the library involves 14 types of network functions in total. Our library is a superset of what has been considered in the prior work in this space.

Table 3 summarizes the key metrics for five different topologies we consider. For each network, we assign hosts into a number of policy groups and enforce 1000 service chaining policies using our library of service chaining templates for randomly selected pairs of policy groups. We use BUZZ and Symnet to generate test cases for each policy (500 policies for each tool). For reachability policies, we consider basic reachability policies using ATPG[2] and TCP reachabilities inspired by Pingmesh [19]. Figure 8 shows the number of test interferences with the number of policies under test.

	# Switches	# Middleboxes	# Links
Internet2	9	9	37
Stanford	16	16	37
Sprint	11	10	28
Oxford	20	20	46
Chinanet	42	39	105

Table 3: Summary of the dataset.

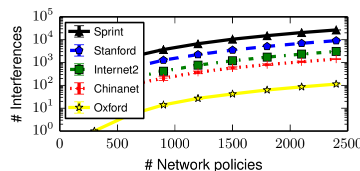


Figure 8: Number of potential interferences.

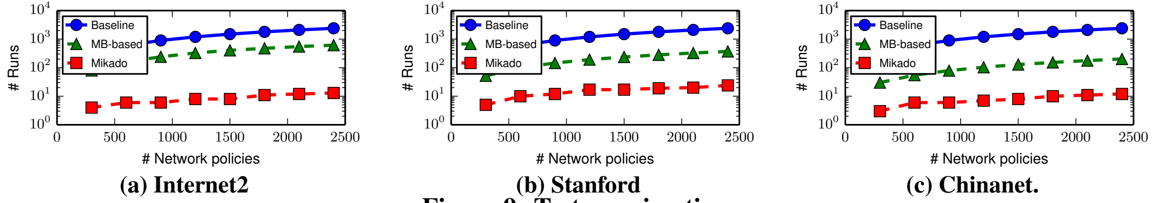
As expected, as the number of policies increases, Mikado detects an increasing number of interferences among policy tests. Further analysis confirms that interferences are caused mostly by multiple tools. For example, when testing all service chaining and TCP reachability policies on Internet2, 3504 interferences are detected and 77% of them happen across different tools. The results further confirm that simply randomizing the parallelization is not likely to generate correct schedules.

7.2 Test Time Reduction

We evaluate the test time reduction using Mikado's scheduling based on the same setup as above. Figure 9 shows the (average) number of runs to complete the tests for different number of policies under test. Recall that all test traffic in each run can be injected in parallel, while multiple runs have to be conducted in sequential. Therefore, the number of runs serves as a reasonable proxy for the test running time. For comparison, we consider two alternative approaches: BASELINE is the basic scheduling approach that runs tests sequentially; MB-based is the heuristic which schedules test traces that do not traverse the same middleboxes together; and Mikado is the proposed scheduling approach (we build a checker that validates the correctness of the schedule). Each data point on the figure is obtained by repeating 100 times.

First, we observe that Mikado significantly reduces the testing running time across all networks (Sprint and Oxford in Appendix) compared to both alternative approaches. For example, when testing 2560 service chaining and TCP reachability policies in Internet2, Mikado

[2] Since ATPG's source code only supports Internet2 and Stanford networks, we only apply ATPG to the two networks.



(a) Internet2 (b) Stanford (c) Chinanet.

Figure 9: Test running time.

can complete all tests in 13 runs by parallelizing most non-interfering traces. In contrast, BASELINE has to inject each trace in a single round (i.e. 2560 runs in total), and MB-based approaches can only complete all test in 663 runs. Both approaches take orders of magnitude more time than Mikado. Putting this result in perspective, if a single run can be complete in 30 seconds, Mikado can effectively reduce the test running time from 21 hours (sequential) to 10 minutes. We also note that the overhead of Mikado's scheduling algorithm is negligible compared with the test generation time: generating all test cases for Internet2 takes 14 hours, while Mikado only takes 2 minutes to generate the test schedule.

Second, Mikado's extensions are effective in further reducing the testing time. Fig. 10 shows the testing running time for the Internet2 network with Mikado's extensions. In general, we observe that our refinement technique achieves greater reduction compared to the random fuzzing heuristic. This is because that the refinement can systematically explore available field values by incorporating with testing tools via simple configurations. With the extensions, the number of runs in Internet2 can be reduced from 13 to 7.

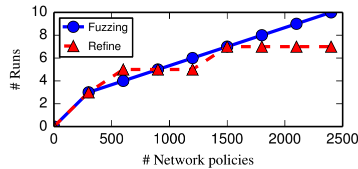


Figure 10: Test running time with Mikado extensions.

Sensitivity to topologies. To evaluate how sensitive Mikado is w.r.t. the network topology, we run our scheduling algorithm on 20 different topologies from TopologyZoo [26]. Figure 11 shows the CDF of the test running time reduction for 200 test traces on each topology. Here, we consider the testing time reduction as the ratio between the test running time using Mikado's correct schedule and that of the sequential testing.

We observe that our scheduling algorithm achieves high reduction for most cases. In particular, for more than half of the topologies, our basic scheduling algorithm achieves at least 87% reduction, while it has 96%+ reduction with the refinement technique.

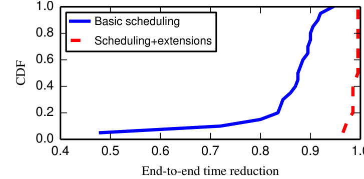


Figure 11: Test running time reduction CDF

7.3 Scalability

Next, we evaluate the scalability of Mikado with respect to the network size, the number and the complexity of policies. Specifically, we evaluate the running time of our scheduling algorithm, including the time of the interference graph generation and graph coloring[3].

With network size. We generate Fattree topologies [4] with varying sizes and augment the topology by adding a middlebox to every switch. We define the network size as the number of switches in it. We run Mikado on each topology with 5000 randomly generated test traces, and each test trace traverses two middleboxes.

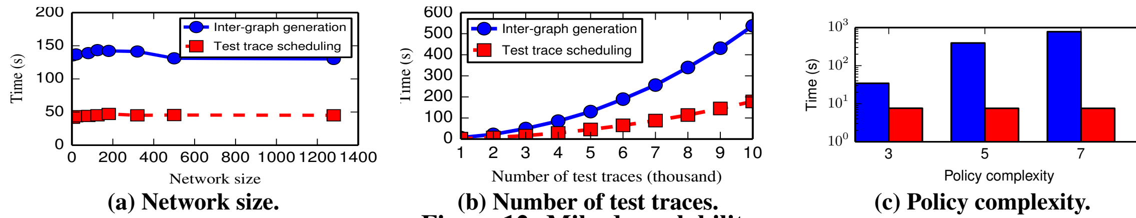
Figure 12a plots the running time of each component of our scheduling algorithm vs. network size. Both the interference graph generation and the graph coloring algorithm can scale up to 1000 switches with negligible increase in the running time. This is expected as the running of both algorithms are dominated by the number of test traces, as shown in §5. We also observe that the running time of the interference graph generation algorithm slightly decreases as the network size increments. This is because on larger networks the chance that two test traces interfere with each other is lower, and leads to faster interference checking.

With the number of test traces. We further run our algorithm on the Fattree topology with 500 switches, and vary the number of test traces from 1000 to 10000.

Figure 12b shows the running time of each algorithm with the number of test traces. As we show in §5, both algorithms run quadratically with the number of test traces; and for 10 thousand test traces, it takes less than 10 minutes for both algorithms to generate a correct schedule.

To put the above results in perspective, we further compare our scheduling algorithm with the naive scheduling approach discussed in §5. Recall that the

[3] We do not consider the refinement heuristic since it relies on other testing tools and not Mikado per se



(a) **Network size.** (b) **Number of test traces.** (c) **Policy complexity.**

Figure 12: Mikado scalability.

naive scheduling approach needs to consider all possible partitions of the test traces, and for each partition it needs to check for all possible runs due to the concurrent injection of all the traces in the partition. The naive approach takes more than 8 hours just for checking compatibility for 10 test traces on one switch.

With policy complexity. We define the policy complexity as the number of middleboxes appeared on the service chain of each policy. We run our algorithm on 2000 test traces, and vary the number of middleboxes each test trace must traverse. Figure 12c shows that both algorithms scales well w.r.t. policy complexity. Even in the case where each test case traverses 7 middleboxes, generating the interference graph takes 13 minutes, while the scheduling algorithm only takes 8 seconds.

8 Related Work

Our work draws ideas from several related areas including network testing, software testing, and formal modeling. We discuss related work in those areas.

Network testing and verification. The high-level goals of network testing [47, 13, 42] and network verification [22, 24, 21, 31, 32, 46, 6, 37, 12, 8, 6, 10, 44] are similar: check whether networks have implemented intended policies correctly. Different from network verification, network testing is better suited for bug finding and in general does not provide guarantees that policies are correctly implemented.

Our work builds upon existing work on network testing and focuses on generating correct and efficient scheduling of test cases. Even though we tested our framework on four existing testing tools, it can be extended as new testing tools are proposed.

Network modeling and abstractions. Formal models of networks are necessary building blocks for formal analysis of the network. There have been a number of existing formal network models (c.f [22, 40, 15, 34, 35, 5, 25, 36]). Some model stateless dataplanes, some model SDN control and dataplanes, and some like ours, model stateful dataplanes. All of the models are defined to facilitate analysis or verification methods that rely on the model. Ours is no exception. Our model, even though straightforward, serves the purpose of providing the basic constructs for defining the correct test ensemble scheduling problem, a key contribution of this work.

Service-chaining policies. The majority of the policies that we use to test Mikado are service-chaining policies. Recently, much work has been done surrounding enforcing service-chaining policies. For example, Simple [39] proposes a static service chain enforcement; FlowTag [14] uses tag bits in packet to implement dynamic service chaining; and [38] offers a composition for service-chaining policies. Rather than enforcement or composition of policies, our work stays at the level of scheduling test cases for those policies and is complementary to the above mentioned projects; Mikado can be used to perform extensive tests of networks that aim to enforce those policies.

Software testing. Our randomized packet fuzzing is borrowed from the software testing literature. Similar to software fuzzing, we also aim to achieve good coverage of the input space, but we do not really care about the coverage of the portions of the network tested. Testing network models is very similar to testing [18, 17, 41, 45, 11, 16], where network model is encoded as a program. However, the type of network testing that we are interested in are live tests (i.e., inject test while into the live network). This provides a set of unique challenges. To carry out an ensemble of tests on software, one can simply perform each test on a copy of the software in parallel. This is not possible for live network tests, since live networks cannot be easily duplicated.

9 Conclusions

We present Mikado, a framework that generates efficient and correct scheduling of test traces for ensembles of network policies. Using a formal model for stateful networks, we develop rigorous definitions of correctness for safely injecting test traces in parallel. We develop an efficient and provably correct algorithm for the test trace scheduling problem. Mikado employs additional heuristics to further improve the testing time reduction. We validate Mikado in a variety of scenarios, and show that Mikado can easily handle large networks with thousands of test traces.

Acknowledgment

We thank all anonymous reviewers and our shepherd Timothy Roscoe for their helpful suggestions and comments. This work is partially supported by NSF CNS-1513961, CNS-1552481, and Intel Labs University Research Office.

References

[1] Cisco NSH Service Chaining Configuration Guide. *https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/wan_nsh/configuration/xe-16/wan-nsh-xe-16-book.html*.

[2] ODL: Service Function Chaining. *http://events.linuxfoundation.org/sites/events/files /slides/odl%20summit%20sfc%20v5.pdf*.

[3] Survey on Network Testing. *http://www.andrew.cmu.edu/user/yifeiy2/mikado/survey.pdf*.

[4] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (New York, NY, USA, 2008), SIGCOMM '08, ACM, pp. 63–74.

[5] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic foundations for networks. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2014), ACM, pp. 113–126.

[6] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), ACM, p. 31.

[7] BUTLER, B. What is intent-based networking? *https://www.networkworld.com/article/3202699/lan-wan/what-is-intent-based-networking.html*.

[8] CANINI, M., VENZANO, D., PERESINI, P., KOSTIC, D., REXFORD, J., ET AL. A NICE Way to Test OpenFlow Applications. In *NSDI* (2012), pp. 127–140.

[9] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[10] DOBRESCU, M., AND ARGYRAKI, K. Software Dataplane Verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 101–114.

[11] EMMI, M., MAJUMDAR, R., AND SEN, K. Dynamic Test Input Generation for Database Applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2007), ISSTA '07, ACM, pp. 151–162.

[12] FAYAZ, S. K., SHARMA, T., FOGEL, A., MAHAJAN, R., MILLSTEIN, T., SEKAR, V., AND VARGHESE, G. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 217–232.

[13] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 275–289.

[14] FAYAZBAKHSH, S. K., CHIANG, L., SEKAR, V., YU, M., AND MOGUL, J. C. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 533–546.

[15] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A network programming language. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 279–291.

[16] GODEFROID, P. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 47–54.

[17] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 213–223.

[18] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Automated Whitebox Fuzz Testing. In *Proceedings of Network and Distributed Systems Security (NDSS 2008)* (November 2008).

[19] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 139–152.

[20] JOSEPH, D. A., TAVAKOLI, A., AND STOICA, I. A policy-aware switching layer for data centers. In *ACM SIGCOMM Computer Communication Review* (2008), vol. 38, ACM, pp. 51–62.

[21] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real Time Network Policy Checking Using Header Space Analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 99–111.

[22] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header Space Analysis: Static Checking for Networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 113–126.

[23] KHAN, F. Intent-based Networking - A Must for SDN. *http://resources.solarwinds.com/intent-based-networking-not-an-option-but-a-must-for-sdn/*.

[24] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 15–27.

[25] KIM, H., REICH, J., GUPTA, A., SHAHBAZ, M., FEAMSTER, N., AND CLARK, R. Kinetic: Verifiable Dynamic Network Control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 59–72.

[26] KNIGHT, S., NGUYEN, H., FALKNER, N., BOWDEN, R., AND ROUGHAN, M. The Internet Topology Zoo. *Selected Areas in Communications, IEEE Journal on 29*, 9 (october 2011), 1765 –1775.

[27] KUMAR, S., TUFAIL, M., MAJEE, S., CAPTARI, C., AND S, H. Service Function Chaining Use Cases In Data Centers. *https://tools.ietf.org/html/draft-ietf-sfc-dc-use-cases-06* (2014).

[28] LERNER, A. Intent-based Networking. *http://blogs.gartner.com/andrew-lerner/2017/02/07/intent-based-networking/*.

[29] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPES, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. MirrorNet: Faithfully Emulating Large Production Networks. In

Proceedings of the 26th Symposium on Operating Systems Principles (New York, NY, USA, 2013), SOSP '17, ACM.

[30] LIU, W., LI, H., HUANG, O., BOUCADAIR, M., LEYMANN, N., CAO, Z., AND HU, J. Service Function Chaining (SFC) Use Cases. *https://tools.ietf.org/html/draft-liu-sfc-use-cases-01* (2014).

[31] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 499–512.

[32] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, ACM, pp. 290–301.

[33] MEDVED, J., VARGA, R., TKACIK, A., AND GRAY, K. Opendaylight: Towards a model-driven sdn controller architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a* (2014), IEEE, pp. 1–6.

[34] MONSANTO, C., FOSTER, N., HARRISON, R., AND WALKER, D. A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices 47*, 1 (2012), 217–230.

[35] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX Association, pp. 1–13.

[36] NELSON, T., FERGUSON, A. D., SCHEER, M. J., AND KRISHNAMURTHI, S. Tierless programming and reasoning for software-defined networks. *NSDI, Apr* (2014).

[37] PANDA, A., LAHAV, O., ARGYRAKI, K., SAGIV, M., AND SHENKER, S. Verifying Reachability in Networks with Mutable Datapaths.

[38] PRAKASH, C., LEE, J., TURNER, Y., KANG, J.-M., AKELLA, A., BANERJEE, S., CLARK, C., MA, Y., SHARMA, P., AND ZHANG, Y. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 29–42.

[39] QAZI, Z. A., TU, C.-C., CHIANG, L., MIAO, R., SEKAR, V., AND YU, M. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 27–38.

[40] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for Network Update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 323–334.

[41] SEN, K., MARINOV, D., AND AGHA, G. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2005), ESEC/FSE-13, ACM, pp. 263–272.

[42] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 314–327.

[43] SUBRAMANIAN, K., D'ANTONI, L., AND AKELLA, A. Genesis: synthesizing forwarding tables in multi-tenant networks. In *POPL* (2017), pp. 572–585.

[44] TSCHAEN, B., ZHANG, Y., BENSON, T., BENERJEE, S., LEE, J., AND KANG, J.-M. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *IEEE SDN-NFV Conference* (2016).

[45] VISSER, W., PĂSĂREANU, C. S., AND KHURSHID, S. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2004), ISSTA '04, ACM, pp. 97–107.

[46] XIE, G. G., ZHAN, J., MALTZ, D. A., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. On Static Reachability Analysis of IP Networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE* (2005), vol. 3, IEEE, pp. 2170–2183.

[47] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic Test Packet Generation. *IEEE/ACM Trans. Netw. 22*, 2 (Apr. 2014), 554–566.

Appendix

Evaluation on Test Running Time

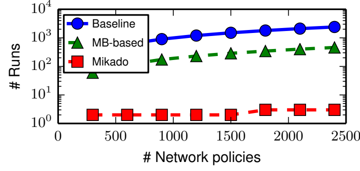


Figure 13: Test running time for Oxford

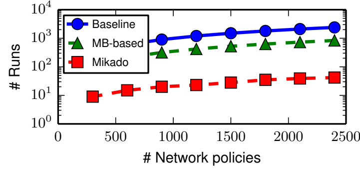


Figure 14: Test running time for Sprint

Proof of Theorem 1

Theorem 1 is a corollary of a stronger lemma (Lemma 2), which we explain next.

We first define the compatibility of two tests traces t_1 and t_2 using just the sequential run of each test trace as follows. The definition essentially is the negation of the condition on line 12 of Algorithm 1.

Definition 4 *Test trace t_1 is compatible with test trace t_2 w.r.t. M_{init} and (ρ,δ) iff $r^1 = R_{seq}(queue(t_1), M_{init})$, $r^2 = R_{seq}(queue(t_2), M_{init})$, $\forall i \in [0, |r^1|-1]$, $\forall j \in [0, |r^2|-1]$, $\forall x \in \mathsf{dom}(dtMap(\delta, M_i^1, lp_i^1)) \cap (upd(\delta, M_j^2, lp_j^2))$, $dtMap(\delta, M_i^1, lp_i^1)(x) = upd(\delta, M_j^2, lp_j^2)(x)$, where M_i^k and lp_i^k denotes the state map and the located packet processed at the ith state of r^k respectively.*

Prove disconnected nodes in the graph returned by the GEN_INTF_GRAPH are compatible is straightforward (Lemma 1).

Lemma 1 *Let G =GEN_INTF_GRAPH(T, C, M_{init}). If v_{t_1} and v_{t_2} are not connected by an edge in G then t_1 is compatible with t_2 w.r.t. M_{init} and C.*

To simplify the proofs, we represent the set of runs $\mathscr{R}(Q,M)$ as an execution tree R, which can be either a leaf node of one network environment, or a node whose content is a network environment E and each of its children is another execution tree obtained from making a transition from E and processing packet lp_i.

$$\begin{array}{llll} \text{Exec. tree} & R & ::= & Lf(E) \\ & & | & Nd(E, \xrightarrow{lp_1/lp'_1} R_1, \cdots, \xrightarrow{lp_n/lp'_n} R_n) \\ \text{Path} & \pi & ::= & \bullet \,|\, n : \pi \end{array}$$

We can identify a unique run in R using a path π, which is a list of numbers indicating which transition to take. For instance, path 1:1:1 identifies the run that takes three transitions following the first branch at each step.

Next we define $r \lesssim_{\pi_1,\cdots,\pi_i,\cdots,\pi_n} R_1;\cdots;R_n$ in Figure 15. The meaning of this judgment is that r simulates an interleaving of runs, each of which is indexed by the path π_i in R_i. For our proofs, R_i is the execution tree for test case t_i and r is a run of the ensemble of tests t_1 to t_n.

The base case is when the indices all point to the initial state. We only need to check that r contains only one state, where the state map is the initial map and the queue only contains the test traces. Here, we $Q_1 \uplus Q_2$ represents an interleaving of packets from Q_1 and Q_2 that preserves the order of packets enforced by Q_1 and Q_2. The inductive case checks that (1) the last transition of r matches a transition in R_i (2) the port queue of the last state of r is an order-perserving interleaving of all the port queues in each R_j at the correct indices and (3) each state location in the state map M in the last state of r preserves the determinant state maps of each R_i. The last condition is the most complex, as we need to reason about who last updates a location s in M. If s is last updated by a packet related to test trace t_m, and s is part of the determinant location of another test trace t_l ($l \neq m$), then $M(s)$ should be the same as the value in that determinant state map. Otherwise, m may interfere with l. However, a test trace can modify its own determinant state locations. It would be too strong to require $M(s)$ to be equal to the values of all determinant state location of the test trace m itself. Instead, we only guarantee that $M(s)$ is equal to the corresponding state in R_m.

Lemma 2 is the key to our correctness proof. It states that any run starting from the ensemble of test traces maintains a simulation relation with each individual runs. Here, $\mathscr{R}_m$ denotes a run of length m.

Lemma 2 *Give a set of test traces $T = \{t_1,\cdots,t_n\}$, s.t. $\forall\ i\ j \in [1,n]$ and $i \neq j$, t_i is compatible with t_j w.r.t. M_{init} and C, let $R_i = \mathscr{R}((queue(t_i), M_{init}), C)$ then let $R = \mathscr{R}_m((queue(flatten(T)), M_{init}), C)$, $\forall r \in R$, $\exists \pi_1,\cdots,\pi_n$, $r \lesssim_{\pi_1,\cdots,\pi_n} R_1;\cdots;R_n$.*

The proof is by induction of $\sum_{i=1}^{n} |\pi_i|$. We rely on the fact that the set of valid runs have the same determinant state maps and updates (Definition 5) to generalize the compatibility conditions on sequential runs to other valid runs.

Definition 5 *We say that the valid runs of test trace t form an equivalence class iff let $r_{seq} = R_{seq}(queue(t), M_{init})$, $\forall r \in \mathscr{R}(queue(t), M_{init})$, $\forall lp \in t$, $r \downarrow_{\{lp\}} = r_{seq} \downarrow_{\{lp\}}$ and $\forall Q, M, lp_1, lp'_1, Q', M'$ s.t. $(Q,M) \xrightarrow{lp/lp'_1} \in r_{seq}$, and $(Q',M') \xrightarrow{lp/lp'_1} \in r$, imply*

$$\frac{R_i = Lf(Q_i, M_i) \text{ or } Nd((Q_i, M_i), _) \qquad M = M_1 = \cdots = M_n = M_{init} \qquad Q = Q_1 \uplus \cdots \uplus Q_n}{(Q, M) \lesssim_{\bullet, \cdots, \bullet} R_1; \cdots; R_n}$$

$$\frac{\begin{array}{l} \pi_i = \pi_i' : \eta \qquad r \lesssim_{\pi_1, \cdots, \pi_i', \cdots, \pi_n} R_1; \cdots; R_n \\ \forall j \in [1,n] \text{ let } R_j \backslash_{\pi_j} = Lf(Q_j, M_j) \text{ or } Nd((Q_j, M_j), _) \\ Q' = \uplus_{j=1}^{n} Q^j \qquad R_i \backslash_{\pi_i'} = Nd((Q_a, M_a), Ch) \qquad Ch.\eta \xrightarrow{lp/lp'} R_i', \\ \forall x \in \mathsf{dom}(M), x \text{ is last updated by } t_m, \\ \quad \forall l \in [1,n], s.t.\ l \neq m, \text{if } x \text{ is in the determinant state map of a state map } M_\alpha \text{ in } R_l, \text{then } M(x) = M_\alpha(x). \\ \quad \text{let } r_m = R_m|_{\pi_m} \text{ let } M_\beta \text{ be the last state in } r_m \text{ where } x \text{ is updated from the previous state}, M(x) = M_\beta(x) \\ \forall x \in \mathsf{dom}(M), x \text{ is not updated by any thread}, \forall j \in [1,n], M_j(x) = M(x) = M_{init}(x) \end{array}}{r \xrightarrow{lp/lp'} (Q, M) \lesssim_{\pi_1, \cdots, \pi_i, \cdots, \pi_n} R_1; \cdots; R_n}$$

Figure 15: Simulation relation

$upd(\delta, M, lp) = upd(\delta, M', lp)$ *and* $dtMap(\delta, M, lp) = dtMap(\delta, M', lp)$.

The correspondence relation established in Lemma 2 ensures that the determinant state maps of individual runs are preserved by the combined run. Theorem 1 follows straightforwardly because observations are determined by the transitions, which in turn, are determined by the state maps.

Distributed Network Monitoring and Debugging with SwitchPointer

Praveen Tammana
University of Edinburgh

Rachit Agarwal
Cornell University

Myungjin Lee
University of Edinburgh

Abstract

Monitoring and debugging large-scale networks remains a challenging problem. Existing solutions operate at one of the two extremes — systems running at end-hosts (more resources but less visibility into the network) or at network switches (more visibility, but limited resources).

We present SwitchPointer, a network monitoring and debugging system that integrates the best of the two worlds. SwitchPointer exploits end-host resources and programmability to collect and monitor telemetry data. The key contribution of SwitchPointer is to efficiently provide network visibility by using switch memory as a "directory service" — each switch, rather than storing the data necessary for monitoring functionalities, stores *pointers* to end-hosts where relevant telemetry data is stored. We demonstrate, via experiments over real-world testbeds, that SwitchPointer can efficiently monitor and debug network problems, many of which were either hard or even infeasible with existing designs.

1 Introduction

Managing large-scale networks is complex. Even short-lived problems due to misconfigurations, failures, load imbalance, faulty hardware and software bugs can severely impact performance and revenue [15, 23, 31].

Existing tools to monitor and debug network problems operate at one of the two extremes. On the one hand, proposals for in-network monitoring argue for capturing telemetry data at switches [7, 20, 30, 21, 18], and querying this data using new switch interfaces [24, 13, 25, 4] and hardware [19, 24]. Such in-network approaches provide visibility into the network that may be necessary to debug a class of network problems; however, these approaches are often limited by data plane resources (switch memory and/or network bandwidth) and thus have to rely on sampling or approximate counters which are not accurate enough for monitoring and diagnosing many network problems (§2).

At the other extreme are recent systems [23, 28] that use end-hosts to collect and monitor telemetry data, and to use this data to debug spurious network events. The motivation behind such end-host based approaches is two folds. First, hosts not only have more available resources than switches but also already need to process packets; thus, monitoring and debugging functionalities can potentially be integrated within the packet processing pipeline with little additional overhead. Second, hosts offer the programmability needed to implement various monitoring and debugging functionalities without any specialized hardware. While well-motivated, such purely end-host based approaches lose the benefits of network visibility offered by in-network approaches.

We present SwitchPointer, a network monitoring and debugging system that integrates the best of the two worlds — resources and programmability of end-host based approaches, and the visibility of in-network approaches. SwitchPointer exploits end-host resources and programmability to collect and monitor telemetry data, and to trigger spurious network events (*e.g.*, using existing end-host based systems like PathDump [28]). The key contribution of SwitchPointer is to efficiently enable network visibility for such end-host based systems by using switch memory as a "directory service" — in contrast to in-network approaches where switches store telemetry data necessary to diagnose network problems, SwitchPointer switches store *pointers* to end-hosts where the relevant telemetry data is stored. The distributed storage at switches thus operates as a distributed directory service; when an end-host triggers a spurious network event, SwitchPointer uses the distributed directory service to quickly filter the data (potentially distributed across multiple end-hosts) necessary to debug the event.

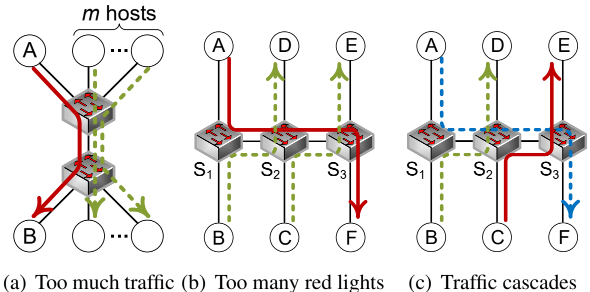
The key design choice of thinking about network switch storage as a directory service rather than a data store allows SwitchPointer to efficiently solve many problems that are hard or even infeasible for existing systems. For instance, consider the network problems shown in Figure 1. We provide an in-depth discussion in §2, but note here that existing systems are insufficient to debug the reasons behind high latency, packet drops or TCP timeout problems for the red flow since this requires maintaining temporal state (that is, flow IDs and packet priorities for all flows that the red flow contends with in Figure 1(a)), combining state distributed across multiple switches (required in Figure 1(b)), and in some cases, maintaining state even for flows that do not trigger network events (for the blue flow in Figure 1(c)).

SwitchPointer is able to solve such problems using a simple design (detailed discussion in §4):

- Switches divide the time into *epochs* and maintain a pointer to all end-hosts to which they forward the packets in each epoch;
- Switches embed their switchID and current epochID into the packet header before forwarding a packet;
- End-hosts maintain a storage and query service that allows filtering the headers for packets that match a (switchID, epochID) pair; and,
- End-hosts trigger spurious events, upon which a controller (or an end-host) uses pointers at the switches to locate the data necessary to debug the event.

While SwitchPointer design is simple at a high-level, realizing it into an end-to-end system requires resolving several technical challenges. The first challenge is to decide the epoch size — too small an epoch would require either large storage (to store pointers for several epochs) or large bandwidth between data plane and control plane (to periodically push the pointers to persistent storage); too large an epoch, on the other hand, may lead to inefficiency (a switch may forward packets to many end-hosts). SwitchPointer resolves this challenge using a hierarchical data structure, where each subsequent level of the hierarchy stores pointers over exponentially larger time scales. We describe the data structure in §4.1.1, and discuss how it offers a favorable tradeoff between switch memory and bandwidth, and system efficiency.

The second challenge in realizing the SwitchPointer design is to efficiently maintain the pointers at switches. The naïve approach of using a hash table for each level of the hierarchy would either require large amount of switch memory or would necessitate one hash operation *per level* per packet for the hierarchical data structure,



(a) Too much traffic (b) Too many red lights (c) Traffic cascades

Figure 1: Three example network problems. Green, blue and red flows have decreasing order of priority. Red flow observes high latency (or even TCP timeout due to excessive packet drops) due to: (a) contention with many high priority flows at a single switch; (b) contention with multiple high priority flows across multiple switches; and (c) cascading problems — green flow (highest priority) delays blue flow, resulting in blue flow contending with and delaying red flow (lowest priority). Please see more details in §2.

making it hard to achieve line rate even for modest size packets. SwitchPointer instead uses a *perfect hash function* [1, 14] to efficiently store and update switch pointers in the hierarchical data structure. Perfect hash functions require only 2.1 bits of storage per end-host per-level for storing pointers and only one hash operation per packet (independent of number of levels in the hierarchical data structure). We discuss storage and computation requirements of perfect hash functions in §4.1.2.

The final two challenges in realizing SwitchPointer design into an end-to-end system are: (a) to efficiently embed switchIDs and epochIDs into packet header; and (b) handle the fact that switch and end-host clocks are typically not synchronized perfectly. For the former, SwitchPointer can of course use clean-slate approaches like INT [4]; however, we also present a design in §4.1.3 that allows SwitchPointer to embed switchIDs and epochIDs into packet header using commodity switches (under certain assumptions). SwitchPointer resolves the latter challenge by exploiting the fact that while the network devices may not be perfectly synchronized, it is typically possible to bound the difference between clocks of any pair of devices within a datacenter. This allows SwitchPointer to handle asynchrony by carefully designing epoch boundaries in its switch data structures.

We have implemented SwitchPointer into an end-to-end system that currently runs over a variety of network testbeds comprising commodity switches and end-hosts. Evaluation of SwitchPointer over these testbeds (§5, §6) demonstrates that SwitchPointer can monitor and debug network events at sub-second timescales while requiring minimal switch and end-host resources.

2 Motivation

In this section we discuss several network problems that motivate the need for SwitchPointer.

2.1 Too much traffic

The first class of problems are related to priority-based and microburst-based contention between flows.

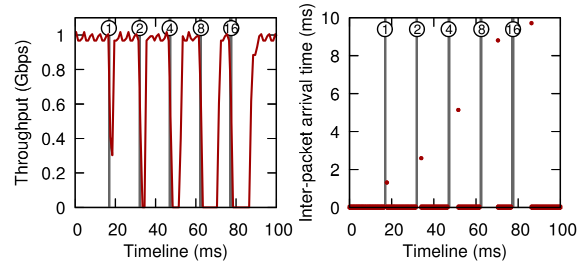
Priority-based flow contention. Consider the case of Figure 1(a), where a low-priority flow competes with many high-priority flows on an output port. As a result, the low priority flow may observe throughput drop, high inter-packet arrival times, or even TCP timeouts.

To demonstrate this problem, we set up an experiment. We create a low-priority TCP flow between two hosts A and B that lasts for 100ms. We then create 5 batches of high-priority UDP bursts; each burst lasts for 1ms and has increasingly larger number of UDP flows (m in Figure 1(a)) all having different source-destination pairs. We use Pica8 P-3297 switches in our experiment; the switch allows us to delay processing of low-priority packets in the presence of a high-priority packet.

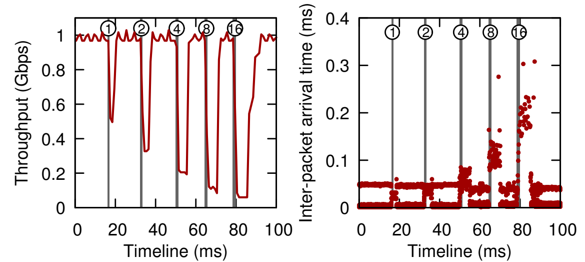
Figure 2(a) demonstrates that high-priority UDP bursts hurt the throughput and latency performance of the TCP flow significantly. With increasingly larger number of high-priority flows in the burst, the TCP flow observes increasingly more throughput drop eventually leading to starvation (*e.g.*, 0 Gbps for ∼10 ms in case of 16 UDP flows). The figure also shows that increasing number of high-priority flows in the burst results in increasingly larger inter-arrival times for packets in the TCP flow. The reduced throughput and increased packet delays may, at the extreme, lead to TCP timeout.

Microburst-based flow contention. We now create a microburst based flow contention scenario, where congestion lasts for short periods, from hundreds of microseconds to a few milliseconds, due to bursty arrival of packets that overflows a switch queue. To achieve this, we use the same set up as priority-based flow contention with the only difference that we use a FIFO queue instead of a priority queue at each switch (thus, all TCP and UDP packets are treated equally). The results in Figure 2(b) show a throughput drop similar to priority-based flow contention, but a slightly different plot for inter-packet arrival times — as expected, the increase in inter-packet delays is not as significant as in priority-based flow contention since all packets get treated equally.

Limitations of existing techniques. The two problems demonstrated above can be detected and diagnosed using specialized switch hardware and interfaces [24]. Without custom designed hardware, these problems can still be detected at the destination of the suffering flow(s), but diagnosing the root cause is significantly more challenging. Packet sampling based techniques would miss microbursts due to undersampling; switch counter based techniques would not be able to differentiate between the priority-based and microburst-based flow contention; and finally, since diagnosing these problems requires looking at flows going to different end-hosts, existing end-host based techniques [23, 28] are insufficient since they only provide visibility at individual end-hosts.



(a) Throughput (left) and inter-packet arrival time (right) of a low-priority TCP flow under priority-based flow contention.



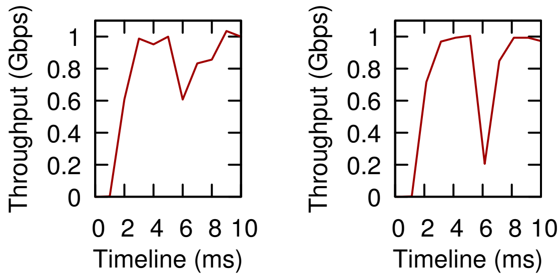
(b) Throughput (left) and inter-packet arrival time (right) of a TCP flow under microburst-based flow contention.

Figure 2: Too much traffic problem depicted in Figure 1(a). Five UDP burst batches are introduced with a gap of 15 ms between each other. The gray lines highlight the five batches, all of which last for 1 ms. The number in circle denotes the number of UDP flows used in each batch.

2.2 Too many red lights

We now consider the network problem shown in Figure 1(b). Our set up uses a low-priority TCP flow from host A to host F (the red flow) that traverses switches S_1, S_2 and S_3. The TCP flow contends with two high-priority UDP flows (B-D and C-E), each lasting for 400μs in a sequential fashion (that is, flow C-E starts right after flow B-D finishes). Consequently, the TCP flow gets delayed for about 400μs at S_1 due to UDP flow B-D and another 400μs at S_2 due to UDP flow C-E.

The result is shown in Figure 3. The destination of the TCP flow sees a sudden throughput drop almost down to



(a) Throughput of flow A-F at S_1 (b) Throughput of flow A-F at S_2

Figure 3: Too many red lights problem depicted in Figure 1(b). UDP is used for flows B-D and C-E and TCP for flow A-F.

200 Mbps. This is a consequence of performance degradation accumulated across two switches S_1 and S_2 — Figures 3(a) and 3(b) show that the throughput is around 600Mbps at S_1 and around 200 Mbps at S_2 (at around 6 ms time point). In fact, the problem is not limited to reduced throughput for the TCP flow — taken to the extreme, adding more "red lights" can easily result in a timeout for the TCP flow.

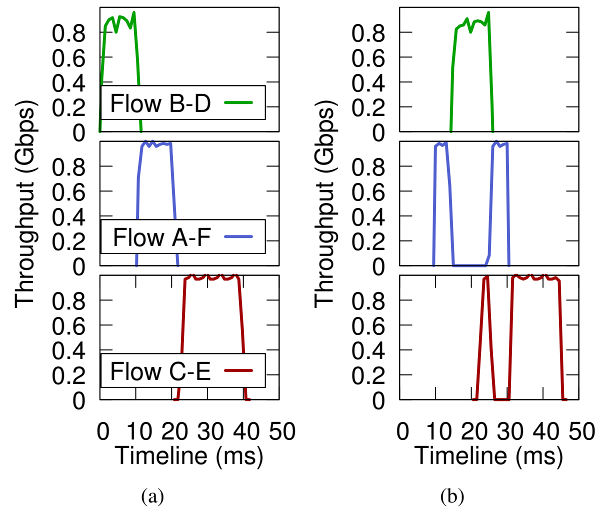
Limitations of existing techniques. The too many red lights problem highlights the importance of combining in-network and end-host based approaches to network monitoring and debugging.

Indeed, it is hard for purely in-network techniques to detect the problem — switches are usually programmed to collect relevant flow- or packet-level telemetry information if a predicate (*e.g.*, throughput drop is more than 50% or queuing delay is larger than 1ms) is satisfied, none of which is the case in the above phenomenon. Since the performance of the TCP flow degrades gradually due to contention across switches, the net effect becomes visible closer to the end-host of the TCP flow.

On the other hand, existing end-host based techniques allow detecting the throughput drop (or for that matter, the TCP timeout); however, these techniques do not provide the network visibility necessary to diagnose the gradual degradation of throughput across switches in the too-many-red-lights phenomenon.

2.3 Traffic cascades

Finally, we discuss the traffic cascade phenomenon from Figure 1(c). Here, we have three flows, B-D, A-F and C-E, with flow priorities being high, middle and low, respectively. Flows B-D and A-F use UDP and last for 10ms each whereas flow C-E uses TCP and transfers 2MB of data. A cascade effect happens when the high-priority flow B-D affects the middle-priority flow A-F



(a) (b)

Figure 4: Traffic cascades problem depicted in Figure 1(c). Throughput of flows (a) without traffic cascades; (b) with traffic cascades. UDP is used for flows B-D and A-F, and TCP for flow C-E.

which subsequently affects the low-priority flow C-E. Specifically, if flow B-D and flow A-F do not contend at switch S_1, the flow A-F will depart from switch S_2 before flow C-E arrives resulting in no flow contention in the network (Figure 4(a)). However, due to contention of flow B-D and flow A-F at switch S_1 (for various reasons, including B-D being rerouted due to failure on a different path), flow A-F is delayed at switch S_1 and ends up reducing the throughput for flow C-E at switch S_2 (Figure 4(b)).

Limitations of existing techniques. Diagnosing the root cause of the traffic cascade problem is challenging for both in-network and for end-hosts based techniques. It not only requires capturing the temporal state (flowIDs and packet priorities for all contending flows) across multiple switches, but also requires to do so even for flows that do not observe any noticeable performance degradation (*e.g.*, the B-D flow). Existing in-network and end-host based techniques fall short of providing such functionality.

2.4 Other SwitchPointer use cases

There are many other network monitoring and debugging problems for which in-network techniques and end-host based techniques, in isolation, are either insufficient or inefficient (in terms of data plane resources). We have compiled a list of such network problems along with a detailed description of how SwitchPointer is able to monitor and diagnose such problems in [8].

3 SwitchPointer Overview

SwitchPointer integrates the benefits of end-host based and in-network approaches into an end-to-end system for network monitoring and debugging. To that end, the SwitchPointer system has three main components. This section provides a high-level overview of these components and how SwitchPointer uses these components to monitor and debug network problems.

SwitchPointer Switches. The first component runs at network switches and is responsible for three main tasks: (1) embedding the telemetry data into packet header; (2) maintaining pointers to end hosts where the telemetry data for packets processed by the switch are stored; and (3) coordinating with an analyzer for monitoring and debugging network problems.

SwitchPointer switches embed at least two pieces of information in packet headers before forwarding a packet. The first is to enable tracing of packet trajectory, that is, the set of switches traversed by the packet; SwitchPointer uses solutions similar to [27, 28] for this purpose. The second piece of information is to efficiently track contending packets and flows at individual ports over fine-grained time intervals. To achieve this, each SwitchPointer switch divides (its local view of) time into epochs and embeds into the packet header the epochID at which the packet is processed. SwitchPointer can of course use clean-slate approaches like INT [4] to embed epochIDs into packet headers; however, we also present a design in §4.1.3 that extends the techniques in [27, 28] to efficiently embed these epochIDs into packet headers along with the packet trajectory tracing information.

Embedding path and epoch information within the packet headers alone does not suffice to debug network problems efficiently. Once a spurious network event is triggered, debugging the problem requires the ability to filter headers contributing to that problem (potentially distributed across multiple end hosts); without any additional state, filtering these headers would require contacting all the end hosts. To enable efficient filtering of headers contributing to the triggered network problem, SwitchPointer uses distributed storage at switches as a directory service — switches store "pointers" to destination end hosts of the packets processed by the switch in different epochs. Once an event is triggered, this directory service can be used to quickly filter out headers for packets and flows contributing to the problem.

Using epochs to track contending packets and flows at switches, and storing pointers to destination end-hosts for packets processed in each epoch leads to several design and performance tradeoffs in SwitchPointer. Indeed, too large an epoch size is not desirable — with increasing epoch size, a switch may forward packets to increasingly many end-hosts within an epoch, leading to inefficiency (at an extreme, this would converge to trivial approach of contacting all end-hosts for filtering relevant headers). Too small an epoch size is also undesirable since with increasing number of epochs, each switch would require either increasingly large memory (SRAM for storing the pointers) or increasingly large bandwidth between the data plane and the control plane (for periodically transferring the pointers to persistent storage).

SwitchPointer achieves a favorable tradeoff between switch memory, bandwidth between the data plane and the control plane, and the efficiency of debugging network problems using a hierarchical data structure, where each subsequent level of the hierarchy stores pointers over exponentially larger time scales. This data structure enables both real-time (potentially automated) debugging of network problems using pointers for more recent epochs, and offline debugging of network problems by transferring only pointers over coarse-grained time scales from the data plane to the control plane. We discuss this data structure in §4.1.1. Maintaining a hierarchy of pointers also leads to challenges in maintaining an updated set of pointers while processing packets at line rate; indeed, a naïve implementation that uses hash tables would require one operation per packet per level of hierarchy to update pointers upon each processed packet. We present, in §4.1.2, an efficient implementation that uses perfect hash functions to efficiently maintain updated pointers across the entire hierarchy using just one operation per packet (independent of number of levels in the hierarchical data structure).

SwitchPointer End-hosts. SwitchPointer, similar to recent end-host based monitoring systems [28, 23], uses end hosts to collect and monitor telemetry data carried in packet headers, and to trigger spurious network events. SwitchPointer uses PathDump [28] to implement its end-host component; however, this requires several extensions to capture additional pieces of information (*e.g.*, epochIDs) carried in SwitchPointer's packet headers and to query headers. We describe SwitchPointer's end-host component design and implementation in §4.2.

SwitchPointer Analyzer. The third component of SwitchPointer is an analyzer that coordinates with SwitchPointer switches and end-hosts. The analyzer can either be colocated with the end-host component, or on a separate controller. A network operator, upon observing a trigger regarding a spurious network event, uses the analyzer to debug the problem. We describe the design and implementation of the SwitchPointer analyzer in §4.3.

An example for using SwitchPointer:
We now describe how a network operator can use SwitchPointer to monitor and debug the too many red lights problem from Figure 1 and §2.2. The destination end-host of the victim TCP flow A-F detects a large throughput drop and triggers the event. The operator, upon observing the trigger, uses the analyzer module to extract the end-hosts that store the telemetry data relevant to the problem — the analyzer module internally queries the destination end-host for flow A-F to extract the trajectory of its packets (switches S_1, S_2 and S_3 in this example) and the corresponding epochIDs, uses this information to extract the pointers from the three switches (for corresponding epochs), and returns the relevant pointers corresponding to the end-hosts that store the relevant headers for flows that contended with the victim TCP flow (D and E in this example). The operator then filters the relevant headers from the end-hosts to learn that flow A-F contended with flow B-D and C-E, and can interactively debug the problem using these headers. SwitchPointer debugs other problems in a similar way (more details in §5).

4 SwitchPointer

In this section, we discuss design and implementation details for various SwitchPointer components.

4.1 SwitchPointer switches

SwitchPointer provides the network visibility necessary for debugging network problems by using the memory at network switches as a distributed directory service, and by embedding telemetry information in the packet headers. We now describe the data structure stored at and packet processing pipeline of SwitchPointer switches.

4.1.1 Hierarchical data structure for pointers

SwitchPointer switches divide their local view of time into epochs and enable tracking of contending packets and flows at switches by storing pointers to destination end-hosts for packets processed in different epochs. SwitchPointer stores these pointers using a hierarchical data structure, where each subsequent level of the hierarchy stores pointers over exponentially larger time scales. We describe this data structure and discuss how it achieves a favorable tradeoff between switch memory (to store pointers) and bandwidth between data plane and control plane (to periodically transfer pointers from switch memory to persistent storage).

Figure 5 shows SwitchPointer's hierarchical data structure with k levels in the hierarchy. Suppose the epoch size is α ms. At the lowermost level, the data

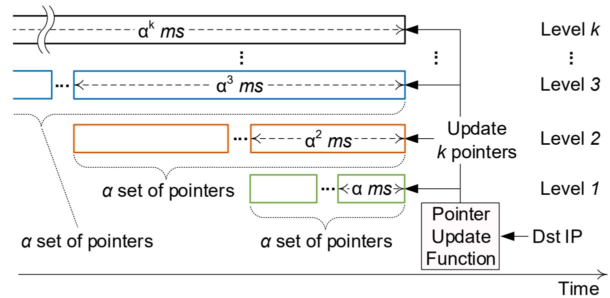


Figure 5: SwitchPointer's hierarchical data structure for storing pointers. For each packet that a switch forwards, SwitchPointer stores a pointer to the packet's destination end-host along a hierarchy of k levels. For epoch size α ms, level h ($1 \leq h \leq k-1$) stores pointers to destination end-hosts for packets processed in last consecutive α^h epochs (that is, α^{h+1} ms) across α set of pointers. The topmost level stores only one set of pointers corresponding to packets processed in last α^k ms.

structure stores α set of pointers, each corresponding to destinations for packets processed in one epoch; thus the set of pointers at the lowermost level provide a per-epoch information on end-hosts storing headers to all contending packets and flows over an α^2 ms period. In general, at level h ($1 \leq h \leq k-1$), the data structure stores α set of pointers corresponding to packets processed in consecutive α^h ms intervals. The top level stores only one set of pointers corresponding to packets processed in last α^k ms of time period.

The hierarchical data structure, by design, maintains some redundant information. For instance, the first set of pointers in level $h+1$ correspond to packets processed in last α^{h+1} ms of time period, collectively similar to all the set of pointers in level h. It is precisely this redundancy that allows SwitchPointer to achieve a favorable tradeoff between switch memory and bandwidth. We return to characterizing this tradeoff below, but note that pointers at the lower level of the hierarchy provide a more fine-grained view of packets and flows contending at a switch and are useful for real-time diagnosis; the set of pointers on the upper levels, on the other hand, provide a more coarse-grained view and are useful for offline diagnosis.

SwitchPointer allows pointers at all levels to be accessed by the analyzer under a *pull model*. For instance, suppose the epoch size is $\alpha = 10$ and the data structure has $k = 3$ levels. Then, each set of pointers at level 1 correspond to 10 ms of time period while those at level 2 correspond to 100 ms of time period. If a network operator wishes to obtain the headers corresponding to packets and flows processed by the switch for last 50 ms (*i.e.*, 5

epochs), it can pull the five most recent set of pointers from level 1; for last 150 ms period, the operator can pull the two most recent pointers from level 2 (which, in fact, correspond to 200 ms time period). In addition to supporting access to the hierarchical data structure using a pull model, SwitchPointer also *pushes* the topmost level of pointers to the control plane for persistent storage every α^k ms which can then be used for offline diagnosis of network events. The toplevel pointers provide coarse-grained view of contending packets and flows at switches which may be sufficient for offline diagnosis but using a push model only for the topmost level pointers significantly reduces the requirements on bandwidth between the data plane and the control plane.

Tradeoff. The hierarchical data structure, as described above, exposes a tradeoff between switch memory and the bandwidth between the data plane and the control plane via two parameters — k and α. Specifically, let the storage needed by a set of pointers to be S bits (this storage requirement depends on the maximum number of end-hosts in the network, and is characterized in next subsection); Then, the overall storage needed by the hierarchical data structure is $\alpha \cdot (k-1) \cdot S + S$ bits. Moreover, since only the topmost pointer is pushed from the data plane to the control plane (once every α^k ms), the bandwidth overhead of SwitchPointer is bounded by $S \times (10^3/\alpha^k)$ bps. For a fixed network size (and hence, fixed S), as k and α are increased, the memory requirements increase and the bandwidth requirements decrease. We evaluate this tradeoff in §6 for varying values of k and α; however, we note that misconfiguration of k and α values may result in longer diagnosis time (the analyzer may touch more end-hosts to filter relevant headers) but does not result in correctness violation.

4.1.2 Maintaining updated pointers at line rate

We now describe the technique used in SwitchPointer to minimize the switch memory requirements for storing the hierarchical data structure and to minimize the number of operations performed for updating all the levels in the hierarchy upon processing each packet.

Strawman: a simple hash table. A plausible solution for storing each set of pointers in the hierarchical data structure is to use a hash table. However, since SwitchPointer requires updating k set of pointers upon processing each packet (one at each level of hierarchy), using a standard hash table would require k operations per packet in the worst case. This may be too high a overhead for high-speed networks (*e.g.*, with 10Gbps links). One way to avoid such overhead is to use hash tables with large number of buckets so as to have a negligible collision probability. Using such a hash table would reduce the number of operations per packet to just one (independent of number of levels in the hierarchy); however, such a hash table would significantly increase the storage requirements. For instance, consider a network with m destinations; given a hash table with n buckets, the expected number of collisions under simple uniform hashing is $m-(n-n(1-1/n)^m)$. Suppose that $m = 100$K and the target number of collisions is $0.001m$ (*i.e.*, 0.1% of 100K keys). To achieve this target, the number of buckets in the hash table should be close to 50 million, 500× larger than the number of keys. Thus, this strawman approach becomes quickly infeasible for our hierarchical data structure — it would either require multiple operations per packet to update the data structure or would require very large switch memory.

Our solution: Minimal perfect hash function. Our key observation is that the maximum number of end-hosts in a typical datacenter is known *a priori* and that it changes at coarse time scales (hours or longer). Therefore, we can construct a minimal perfect hash function to plan ahead on the best way to map destinations to buckets to avoid hash collisions completely. In fact, since each level in the hierarchy uses the same perfect hash function, SwitchPointer needs to perform just one operation per packet to find the index in a bit array of size equal to the maximum number of destinations; the same index needs to be updated across all levels in the hierarchy. Upon processing a packet, the bit at the same index across the bit array is set in parallel. Lookups are also easy — to check if a packet to a particular destination end-host was processed in an epoch, one simply needs to check the corresponding bit (given by the perfect hash function) in the bit array.

The minimal perfect hash function provides $O(1)$ update operation and expresses a 4-byte IP address with 1 bit (*e.g.*, 100Kbits for 100K end-hosts). While an additional space is required to construct a minimal perfect hash function, it is typically small (70 KB and 700 KB for for 100K and 1M end-hosts respectively; see §6.1). Moreover, while constructing a perfect hash function is a computationally expensive task, small storage requirement of perfect hash tables allow us to recompute the hash function only at coarse-grained time intervals — temporary failures of end-hosts do not impact the correctness since the bits corresponding to those end-hosts will simply remain unused. For resetting pointers at level h, an agent at the switch control plane updates a register with the memory address of next pointer every α^h ms and resets its content. The agent conducts this process for pointers at all levels.

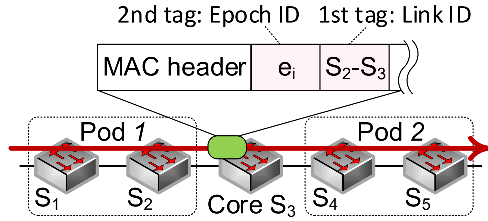


Figure 6: Telemetry data embedding using two VLAN tags using a modified version of the technique in [27]. See §4.1.3 and §4.2.1 for discussion.

4.1.3 Embedding telemetry data

SwitchPointer requires two pieces of information to be embedded in packet headers. The first is the trajectory of a packet, that is, the set of switches (*i.e.*, switchIDs) traversed by the packet between the source and the destination hosts. The second is epoch information (*i.e.*, epochID) on when a packet traverses those switches.

SwitchPointer extends the link sampling idea from [27, 28] to efficiently enable packet trajectory tracing and epoch embedding for commonly used datacenter network topologies (*e.g.*, clos networks like fat-tree, leaf-spine and VL2) without any hardware modifications. Specifically, it is shown in [27, 28] that an end-to-end path in typical datacenter network topologies can be represented by selectively picking a small number of key links. For instance, in a fat-tree topology the technique reconstructs a 5-hop end-to-end path by selecting only one aggregate-core link and embedding its linkID into the packet header. For embedding epochIDs in addition to the linkID, we extend the technique that relies on IEEE 802.1ad double tagging. When a linkID is added to the packet header using a VLAN tag, we add an epochID using another tag (see Figure 6).

The number of rules for embedding linkID increases linearly with respect to the number of switch ports whereas only one flow rule is for epochID embedding. However, the switch needs a rule update once every epoch — as the epoch changes, the switch should be able to increment epochID and add a new epochID for incoming packets. A commodity OpenFlow switch that we use is capable of updating flow rules every 15 ms, giving us a lower bound on α granularity for commodity switches.

We note that the limitations on supported topologies and α granularity in our implementation over commodity switches are merely an artifact of today's switch hardware — it is possible to use SwitchPointer with clean-slate solutions such as INT [4] to support trajectory tracing and epoch embedding over arbitrary topologies.

4.2 SwitchPointer End-hosts

SwitchPointer uses PathDump [28] to collect and monitor telemetry data carried in packet headers, and to trigger spurious network events. In this subsection, we discuss the extensions needed in PathDump to capture additional pieces of information (*e.g.*, epochIDs) carried in SwitchPointer's packet headers and to query headers.

4.2.1 Decoding telemetry data

When a packet arrives at its destination, the destination host extracts the telemetry data from the packet header. If the network supports clean-slate approaches like INT [4], this is fairly straight forward. For implementation using commodity switches (using techniques discussed in §4.1.3), the host extracts two VLAN tags containing the switchID and the epochID associated with the switchID. Using the switchID, the end-to-end path can be constructed using techniques in [27, 28], giving us a list of switches visited by the packet. Next, we decide a list of epochIDs for each of those switches. However, since only one epochID is available at the end-host, it is hard to determine the missing epochIDs for those switches correctly. Thus, we set a range of epochs that the switches should examine. Specifically, we may need to examine max_delay/α number of pointers at each switch due to uncertainty in epoch identification.

Let Δ denote the a maximum one hop delay and ε be a maximum time drift among all switches. Given epochID e_i of switch S and an end-to-end path, the epochIDs for switches along the path are identified as follows.

For the upstream switches of switch S, the epoch range is $[e_i - (\varepsilon + j \cdot \Delta)/\alpha, e_i + \varepsilon/\alpha]$ and for the downstream switches of S, it is $[e_i - \varepsilon/\alpha, e_i + (\varepsilon + j \cdot \Delta)/\alpha]$, where j is hop difference between an upstream (or downstream) switch and switch S. Suppose $\alpha = 10$ ms, $\varepsilon = \alpha$ and $\Delta = 2 \cdot \alpha$. For instance, in the example of Figure 6, we set $[e_i - 3, e_i + 1]$ for switch S_2, $[e_i - 1, e_i + 3]$ for S_4, and so forth. This provides a reasonable bound due to two reasons. First, a maximum queuing delay is within tens of milliseconds in the datacenter network (*e.g.*, 14 ms in [9]). Second, millisecond-level precision is sufficient as SwitchPointer epochs are of similar granularity.

4.2.2 Event trigger and query execution

The end-host also has an agent that communicates with and executes queries on behalf of the analyzer. The agent is implemented using a microframework called flask [3], and implements a variety of techniques (similar to those in existing end-host based systems [28, 23]) to monitor spurious network events.

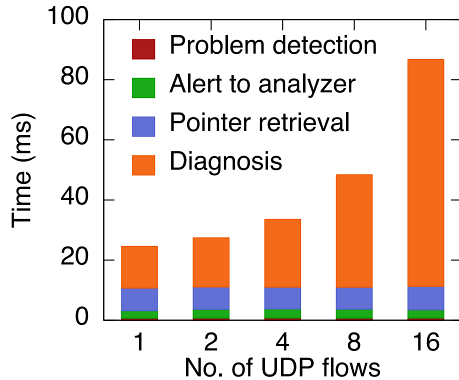


Figure 7: Debugging time of the priority-based flow contention problem depicted in Figure 2(a). SwitchPointer is able to monitor and debug the problem in less than 100ms. We provide a break down of the diagnosis latency later in Figure 12.

4.3 SwitchPointer Analyzer

The analyzer is also implemented using flask microframework. It communicates with both switch and end-host agents. From the switch agent, the analyzer obtains pointers to end-hosts for epoch(s). From the end-host agent, it receives alert messages, and exchanges queries and responses. Another responsibility is that it constructs a minimal perfect hash function whenever there are permanent changes in the number of end-hosts in the network, especially when end-hosts are newly added. It then distributes the minimal perfect hash function to all the switches in the network. The analyzer also does pre-processing of pointers by leveraging network topology, flow rules deployed in the network, etc. For example, to diagnose the network problem experienced by a flow, the analyzer filters out irrelevant end-hosts in the pointer if the paths between the flow's source and those end-hosts do not share any path segment of the flow. This way, the analyzer reduces search radius, *i.e.*, number of end-hosts that it has to contact.

5 SwitchPointer Applications

In this section, we demonstrate some key monitoring applications SwitchPointer supports.

5.1 Too much traffic

We debug the problem discussed in §2 using SwitchPointer. This problem include two different cases: (i) priority-based flow contention and (ii) microburst-based flow contention. The debugging processes of both cases are similar; the only difference is the former case requires the analyzer to examine flow's priority value. Thus, we only discuss the former case.

Figure 7 shows the breakdown of times it took to diagnose the priority-based flow contention case. First, we instrument hosts with a simple trigger that detects drastic throughput changes. The trigger measures throughput every 1 ms interval and generates an alert to the analyzer if throughput drop is more than 50%. The problem detection takes less than 1 ms, thus almost invisible from the figure (3-4 ms for the microburst-based contention case). Then, it takes 2-3 ms to send the analyzer an alert and to receive an acknowledgment. The alert contains a series of <switchID, a list of epochIDs, a list of byte counts per epoch> tuples that tell the analyzer when and where packets of the TCP flow visit. The analyzer uses the switchIDs and epochIDs, and obtains relevant pointers from switches. In this scenario, it only takes about 7-8 ms to retrieve a pointer from one switch.

Next, the analyzer learns hosts encoded in the pointer, and diagnoses the problem by consulting them; it collects telemetry data such as UDP flow's priority, the number of bytes in UDP flow during the epoch when the TCP flow experiences high delay. The analyzer finally draws a conclusion that the presence of high-priority UDP flows aggravated the performance of the low-priority TCP flow. As shown in Figure 7, the time for the diagnosis increases as the number of consulted hosts (*i.e.*, each UDP flow is destined to a different host) increases. Although not too large, the diagnosis overhead inflation pertains to the implementation of connection initiation; we discuss this matter and its optimization in §6.2.

5.2 Too many red lights

This problem illustrated in Figure 1(b) (for its behavior, see Figure 3) requires spatial correlation of telemetry data across multiple switches for diagnosis. While this problem is challenging to existing tools, SwitchPointer easily diagnoses it as follows.

First, destination F triggers an alert to the analyzer in no time ($\sim$1 ms) by using our throughput drop detection heuristic introduced in §5.1. The alert contains IDs for switches S_1,S_2 and S_3 and their corresponding epochID ranges. The analyzer contacts all of the switches and retrieves pointers that match the epoch IDs for each switch in 10 ms, and then conducts diagnosis (another 20 ms) by obtaining telemetry data for UDP flows B-D and C-E from hosts D and E, respectively. The analyzer finds out that low (A-F) and high prority (B-D and C-E) flows have at least one common epochID, and finally concludes (in about 30 ms) that both flows B-D and C-E contributed to the actual impact on the TCP flow.

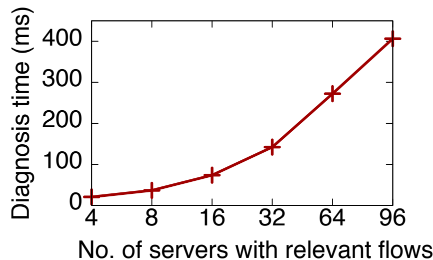


Figure 8: Latency for diagnosing load imbalance problem.

5.3 Traffic cascades

This problem is a more challenging problem to existing tools because debugging it requires spatial and temporal correlation of telemetry data (see Figure 1(c) for the problem illustration and Figure 4(b) for its behavior). SwitchPointer diagnoses the problem as follows.

First, the low-priority TCP flow C-E observes a large throughput drop at around 26 ms (see Figure 4(b)) and triggers an alert along with switchIDs and corresponding epoch details. Then, the analyzer retrieves pointers that match with epochIDs from S_2 and S_3, contacts F and finds out the presence of middle-priority flow A-F on S_2 caused the contention in ∼25 ms. Since flow A-F has middle-priority, the analyzer subsequently examines pointers from switches (*i.e.*, S_1 and S_2) along the path of flow A-F in order to see whether or not the flow was affected by some other flows. From a pointer from switch S_1, the analyzer comes to know that flow B-C made flow A-F delayed, which in turn had flows A-F and C-E collide. This part of debugging takes another 25 ms. Hence, the whole process takes about 50 ms in total.

Of course, in a large datacenter network, debugging this kind of problem can be more complex than the example we studied here. Therefore, in practice the debugging process may be an off-line task (with a pointer at a higher level that covers many epochs) rather than an online task. However, independent of whether it is an off-line or online task, SwitchPointer showcases, with this example, that it is feasible to diagnose network problems that need both spatial and temporal correlation.

5.4 Load imbalance diagnosis

To demonstrate the way SwitchPointer works for diagnosing load imbalance, we create the same problematic setup used in [28]. In that setup, a switch that is configured to malfunction, forwards traffic unevenly to two egress interfaces; specifically, packets from flows whose size is less than 1 MB are output on one interface; otherwise, packets are forwarded to the other interface. We vary the number of flows from 4 to 96. Each flow is destined to a different end-host. Using this setup, we can understand how the number of end-hosts contacted by the analyzer impacts SwitchPointer's performance.

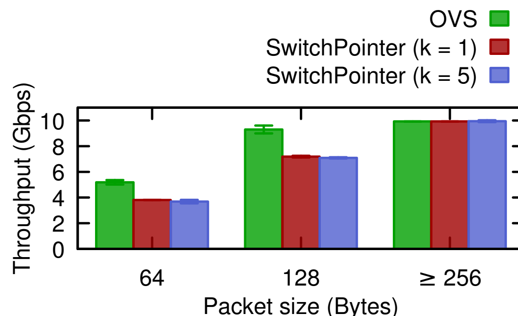
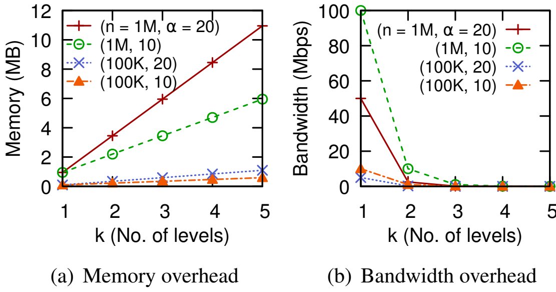


Figure 9: For smaller packet sizes, SwitchPointer is unable to sustain line rate due to overheads of perfect hash function. SwitchPointer is able to achieve line rate for a 10GE interface for packets of size 256bytes and more.

The debugging procedure is similar to that of other problems we already studied. This problem is detected by monitoring interface byte counts per second. The analyzer fetches the pointers corresponding to the most recent 1 sec. It then obtains the end-hosts in the pointers, and sends them a query that computes a flow size distribution for each of the egress interfaces of the switch. Finally, the analyzer finds out that there is a clean separation in flow size between two distributions. Figure 8 shows the diagnosis time of running a query as a function of the number of end-hosts consulted by the analyzer. The diagnosis time increases almost linearly as the analyzer consults more end-hosts. Since this trend comes from the same cause, we refer to §6.2 for understanding individual factors that contribute to the diagnosis time.

6 SwitchPointer Evaluation

We prototype SwitchPointer on top of Open vSwitch [6] over Intel DPDK [2]. To build a minimal perfect hash function, we use the FCH algorithm [14] among others in CMPH library [1]. We also implement the telemetry data extraction and epoch extrapolation module (§4.2.1) on OVS. The module maintains a list of flow records; one record consists of the usual 5-tuple as flowID, a list of switchIDs, a series of epoch ranges that correspond to each switchID, byte/packet counts and a DSCP value as flow priority. This flow record is initially maintained in memory and flushed to a local storage, implemented using MongoDB [5]. We now evaluate SwitchPointer in terms of switch overheads and query performance under real testbeds that consist of 96 servers.



(a) Memory overhead (b) Bandwidth overhead

Figure 10: Overheads of SwitchPointer. At (n, α) in the legend, n denotes the maximum number of IP addresses traced by SwitchPointer, and α is an epoch duration in ms.

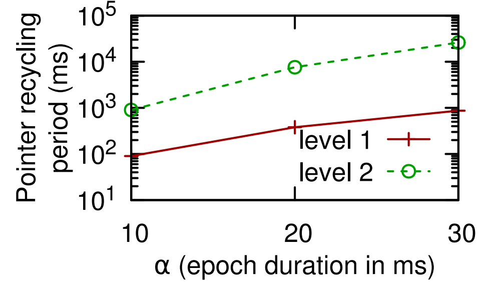


Figure 11: Recycling period of a pointer when $k = 3$.

6.1 Switch overheads

To quantify switch overheads, we vary epoch duration (α ms), the number of levels in a pointer (k), the number of IP addresses (n) and packet size (p). We set up two servers connected via a 10GE link. From one server, we generate 100K packets, each of which has a unique destination IP (hence, 100K flows); we play those 100K packets repeatedly to the other server where SwitchPointer is running using one 3.1 GHz CPU core. Under the setup, we measure i) throughput, ii) the amount of memory to keep pointers on data plane, iii) bandwidth to offload pointers from SRAM (data plane) to off-chip storage (control plane), and iv) pointer recycling period.

Throughput. We compare SwitchPointer's throughput with that of vanilla OVS (baseline) over Intel DPDK. We set $k = 1$ and 5. Here one pointer of SwitchPointer is configured to record 100K unique end-hosts. We then measure the throughput of SwitchPointer while varying p. Our current implementation in OVS processes about 7 million packets per second. From Figure 9, we observe that OVS and both configurations of SwitchPointer achieve a full line rate (~9.99 Gbps) when $p \geq 256$ bytes. In contrast, when $p < 256$ bytes, both OVS and SwitchPointer face throughput degradation. For example, when p is 128 bytes, OVS achieves about 9.29 Gbps whereas SwitchPointer's throughput is about 22% less than that of OVS. However, since an average packet size in data centers is in general larger than 256 bytes (*e.g.*, 850 bytes [10], median value of 250 bytes for hadoop traffic [26]), the throughput of SwitchPointer can be acceptable. We also envision that a hardware implementation atop programmable switch [11, 19] would eliminate the limitation of a software version.

Memory. Perfect hash functions account for about 70 KB (n = 100K) and 700 KB (n = 1M). In addition, n also governs the pointer's size: 12.5 KB (n = 100K) and 125 KB (n = 1M). Together SwitchPointer requires to have 82.5 KB and 825 KB, respectively. These are the minimum amount of memory requirement for SwitchPointer. Figure 10(a) shows the memory overhead; the memory requirement increases in proportion to each of k and α. When n = 1M, $\alpha = 10$ and $k = 3$, SwitchPointer consumes 3.45 MB; for n = 100K, it is only 345 KB.

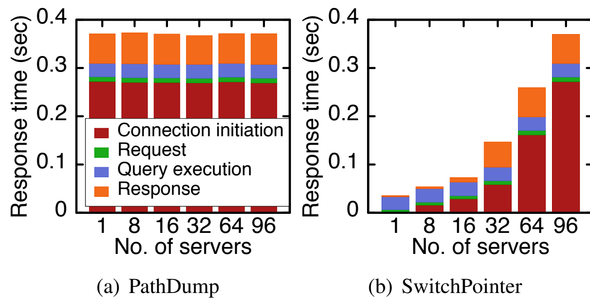
Bandwidth. In contrast to memory overhead, the bandwidth requirement of system bus between SRAM and off-chip storage decreases as we increase k and α because larger values of those parameters make the pointer flush less frequent. In particular, k has a significant impact in controlling the bandwidth requirement; increasing it drops the requirement exponentially. For n = 1M and $\alpha = 10$ (the most demanding setting in Figure 10(b)), the bandwidth requirement reduces from 100 Mbps ($k = 1$) to 10 Mbps ($k = 2$).

The results in Figures 10(a) and 10(b) present a clear tradeoff between memory and bandwidth. Depending on the amount of available resources and user's requirements, SwitchPointer provides a great flexibility in adjusting its parameters. For instance, if memory is a scarce resource, it may be better to keep $k \leq 3$ and $\alpha \leq 10$.

Pointer recycling period. Except for top level pointers, pointers are recycled after all the pointers on the same layer are used. The pointer recycling period at level h is expressed as $\alpha(\alpha^h - 1)$ ms where $1 \leq h < k$. Figure 11 shows a tradeoff between α and k. As expected, the recycling period exponentially increases as the level increases (when $\alpha = 10$, the recycling period of a pointer at level 1 is 90 ms and it is 900 ms at level 2). Because too small α may always let SwitchPointer end up accessing a higher-level pointer, α should be chosen carefully.

6.2 Query performance

We now evaluate the query performance of SwitchPointer, which we compare with that of PathDump [28]



(a) PathDump (b) SwitchPointer

Figure 12: Top-100 query response time. Most of SwitchPointer latency overheads are due to connection initiation requests from the analyzer to the end-hosts and can be improved with a more optimized RPC implementation.

(baseline). We run a query that seeks top-*k* flows in a switch in our testbed where there are 96 servers. The key difference between SwitchPointer and PathDump is that SwitchPointer knows which end-hosts it needs to contact but PathDump does not. Thus, PathDump executes the query from all the servers in the network. To see the impact of the difference, we vary the number of servers that contain telemetry data of flows that traverse the switch.

From Figure 12 we observe that the response time of SwitchPointer gradually increases as the number of servers increases. On the other hand, PathDump always has the longest response time as it has to contact all 96 servers anyway. Both of them only have a similar response time when all the servers have relevant flow records and thus SwitchPointer has to contact all of them.

A closer look reveals that most of the response time is because of connection initiation for both SwitchPointer and PathDump. In our current implementation, the analyzer creates one thread per server to initiate connection when a query should be executed. This on-demand thread creation delays the execution of query at servers. This is an implementation issue, not a fundamental flaw in design. Thus, it can be addressed with proper technique such as thread pull management. However, since PathDump must contact all the servers regardless of whether or not the servers have useful telemetry data, it wastes servers' resources. On the contrary, SwitchPointer only spends right amounts of server resources, thus offering a scalable way of query execution.

7 Related Work

SwitchPointer's goals are related to two key areas of related work on network monitoring and debugging.

End-host based approaches. These approaches [23, 28, 29, 12, 15, 22] typically exploit the fact that end-hosts have ample resources and support for programmability needed to monitor and diagnose spurious network events. As discussed in §2, these approaches lack the network visibility needed to debug a class of network problems. SwitchPointer incorporates such visibility by using switch memory as a directory service thus enabling monitoring and debugging for a larger class of network problems [8].

In-network approaches. In-network approaches to network monitoring and debugging have typically focused on designing novel switch data structures [30, 20, 21, 18, 7], abstractions [17, 24, 16, 25, 13] and even switch hardware [24] to capture telemetry data at switches. While interesting, these approaches are often limited by switch and data plane resources required to store and query the telemetry data. Moreover, as discussed in §2, existing in-network approaches are insufficient to debug network problems that require analyzing data captured across multiple switches. SwitchPointer is able to overcome these limitations of in-network approaches using limited switch resources (4-6 MB of SRAM and 1-2 Mbps of bandwidth between the data plane and the control plane) by delegating the tasks of collecting and monitoring the telemetry data to the end-hosts, and by using switch memory as a distributed directory service.

8 Conclusion

SwitchPointer is a system that integrates the benefits of end-host based approaches and in-network approaches to network monitoring and debugging. SwitchPointer uses end-host resources and programmability to collect and monitor telemetry data, and to trigger spurious network events. The key technical contribution of SwitchPointer is to enable network visibility by using switch memory as a "directory service" — SwitchPointer switches use a hierarchical data structure to efficiently store *pointers* to end-hosts that store relevant telemetry data. Using experiments on real-world testbeds, we have shown that SwitchPointer efficiently monitors and debugs a large class of network problems, many of which were either hard or even infeasible with existing designs.

Acknowledgments

We would like to thank anonymous NSDI reviewers and our shepherd Mohammad Alizadeh for their insightful comments and suggestions. We would also like to thank Minlan Yu for many discussions during the project. This work was in part supported by EPSRC grants EP/L02277X/1 and EP/N033981/1, a Google faculty research award, and NSF grant F568379.

References

[1] CMPH - C Minimal Perfect Hashing Library. http://cmph.sourceforge.net/.

[2] DPDK: Data Plane Development Kit. http://dpdk.org/.

[3] Flask. http://flask.pocoo.org/.

[4] In-band Network Telemetry. https://github.com/p4lang/p4factory/tree/master/apps/int.

[5] MongoDB. https://www.mongodb.org/.

[6] Open vSwitch. http://openvswitch.org/.

[7] Sampled NetFlow. http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html, 2003.

[8] PathDump. https://github.com/PathDump, 2016.

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

[10] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. *ACM SIGCOMM CCR*, 40(1), Jan. 2010.

[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[12] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang. Felix: Implementing traffic measurement on end hosts using program analysis. In *ACM SIGCOMM SOSR*, 2016.

[13] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ACM SIGPLAN ICFP*, 2011.

[14] E. A. Fox, Q. F. Chen, and L. S. Heath. A Faster Algorithm for Constructing Minimal Perfect Hash Functions. In *ACM SIGIR*, 1992.

[15] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *ACM SIGCOMM*, 2015.

[16] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger. Network monitoring as a streaming analytics problem. In *ACM HotNets*, 2016.

[17] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *USENIX NSDI*, 2014.

[18] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, 2017.

[19] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *ACM SIGCOMM*, 2014.

[20] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*, 2016.

[21] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM*, 2016.

[22] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*, 2011.

[23] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *ACM SIGCOMM*, 2016.

[24] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM SIGCOMM*, 2017.

[25] S. Narayana, M. Tahmasbi, J. Rexford, and D. Walker. Compiling Path Queries. In *USENIX NSDI*, 2016.

[26] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[27] P. Tammana, R. Agarwal, and M. Lee. CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *ACM SIGCOMM SOSR*, 2015.

[28] P. Tammana, R. Agarwal, and M. Lee. Simplifying Datacenter Network Debugging with PathDump. In *USENIX OSDI*, 2016.

[29] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *USENIX NSDI*, 2011.

[30] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *USENIX NSDI*, 2013.

[31] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *ACM SIGCOMM*, 2015.

Stroboscope: Declarative Network Monitoring on a Budget

https://stroboscope.ethz.ch

Olivier Tilmans
Université catholique de Louvain

Tobias Bühler
ETH Zürich

Ingmar Poese
BENOCS

Stefano Vissicchio
University College London

Laurent Vanbever
ETH Zürich

Abstract

For an Internet Service Provider (ISP), getting an accurate picture of how its network behaves is challenging. Indeed, given the carried traffic volume and the impossibility to control end-hosts, ISPs often have no other choice but to rely on heavily sampled traffic statistics, which provide them with coarse-grained visibility at a less than ideal time resolution (seconds or minutes).

We present Stroboscope, a system that enables fine-grained monitoring of *any* traffic flow by instructing routers to mirror millisecond-long traffic slices in a programmatic way. Stroboscope takes as input high-level monitoring queries together with a budget and automatically determines: *(i)* which flows to mirror; *(ii)* where to place mirroring rules, using fast and provably correct algorithms; and *(iii)* when to schedule these rules to maximize coverage while meeting the input budget.

We implemented Stroboscope, and show that it scales well: it computes schedules for large networks and query sizes in few seconds, and produces a number of mirroring rules well within the limits of current routers. We also show that Stroboscope works on existing routers and is therefore immediately deployable.

1 Introduction

Not all networks are created equally when it comes to monitoring. ISP networks, in particular, suffer from extremely poor visibility. As they do not control end-hosts and carry huge amounts of traffic, ISP operators often have no choice but to rely on pure in-network solutions based on random packet sampling (i.e., NetFlow [1] or sFlow [2]). By design, random sampling provides no guarantee on which traffic flows will be sampled, by which router and at what time. Except for few heavy-hitters [3], even minutes-long collections of random samples typically provide coarse-grained and inaccurate bandwidth estimations for the large majority of the prefixes. Moreover, the likelihood of randomly sampling the same flow across the network is extremely low; hence, it is basically impossible to use random samples for reasoning on the network-wide forwarding behavior, and monitoring anything else than bandwidth.

We confirmed these limitations in an actual Tier-1 ISP by analyzing the Netflow data collected by hundreds of routers over 10 minutes. We observed that most BGP prefixes (65%) are not observed at all, 15% of them are observed only twice, and just 10% of all prefixes are observed more than 30 times. Even worse, 75% of these observed flows were only seen on a single router, making it *impossible to track flows network-wide*, even for the largest heavy hitters.

As a result, ISP operators are currently incapable of answering practical questions like: *What is the ingress router for a given packet seen at a specific node? Which paths does the traffic follow? Is the network-wide latency acceptable? Is traffic load-balanced as expected?*

Stroboscope This paper presents Stroboscope, a scalable monitoring system that complements existing tools like NetFlow, by enabling fine-grained monitoring of *any* traffic flow. Stroboscope exploits the possibility to extract small traffic samples (i.e., slices) in a programmatic way, by activating and deactivating traffic mirroring for any destination prefix, up to a single IP address, network-wide, and within milliseconds. Our tests confirm that this possibility is available today, on currently deployed routers, making Stroboscope immediately deployable.

By coordinating packet mirroring across routers, Stroboscope implements *deterministic packet sampling*: it collects copies of the same packets from multiple locations, following such packets as they cross the network. This enables Stroboscope to precisely measure the network forwarding behavior including traffic paths, one-way delays and load-balancing ratios. Traffic slices with no packets are also informative: Stroboscope uses them to determine additional forwarding properties, like packet loss and devices not receiving specific flows.

Challenges Given a high-level query, determining which flows to mirror, where and when is both hard and potentially dangerous—especially when considering arbitrary network dynamics (e.g., unexpected traffic shifts). Aggressive mirroring strategies can lead to significant congestion (e.g., if many routers mirror traffic for popular destinations) and inaccurate results (e.g., if congestion affects the mirrored traffic). Conversely, conservative strategies can lead to poor coverage and slow answers.

Compilation Stroboscope tackles those challenges on behalf of operators. From high-level queries, it automatically derives how to mirror traffic so as to maximize monitoring accuracy without exceeding a budget, while also adapting to network dynamics in near real time.

Stroboscope's compilation process follows three steps. First, Stroboscope decides *what* (which prefixes) to mirror for every query, dynamically adapting this decision according to the amount of mirrored packets. Second, Stroboscope computes *where* to activate mirroring rules, in order to maximize coverage while minimizing the impact on the budget. Third, Stroboscope calculates *when* to mirror and for how long, producing a budget-compliant schedule, optimized across all input queries.

Guarantees Stroboscope provides strong guarantees in terms of budget compliance, even in the presence of unpredictable network dynamics. In fact, traffic mirrored by Stroboscope can only exceed the budget for at most the few milliseconds needed to collect a single traffic slice.

Implementation We implemented Stroboscope, and show that it scales well: it computes schedules for large networks and query sizes in few seconds, and produces a number of mirroring rules well within the limits of current routers. We also demonstrate how to build practical monitoring applications on top of Stroboscope, such as estimating one-way delays, loss rates, or load-balancing ratios for any destination prefix.

Contributions Our earlier work [4] showed the benefits of mirroring thin traffic slices to monitor networks. This paper goes further by describing the complete design, implementation and evaluation of the corresponding system. We make the following contributions:

- A novel fine-grained and scalable monitoring approach based on deterministic traffic sampling (§2);
- Practical algorithms to: *(i)* estimate unknown traffic demands in real time (§3); and *(ii)* compute optimally placed mirroring rules (§4), as well as schedule them while adhering to a given budget (§5);
- A full implementation of Stroboscope (§6) along with a thorough evaluation using benchmarks, simulations and tests on Cisco routers (§7);
- A case study demonstrating how to use Stroboscope measurements to estimate one-way delays, loss rates, and load-balancing ratios (§8).

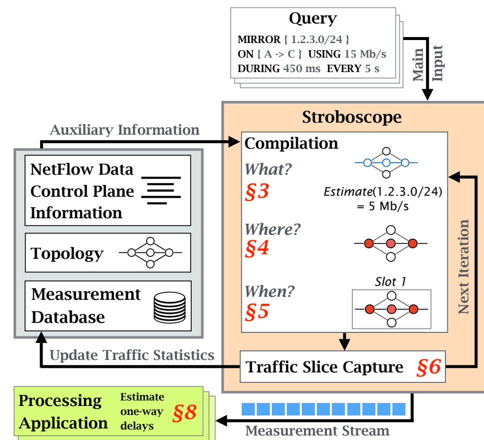


Figure 1: Stroboscope translates high-level queries to measurement streams by capturing packet slices.

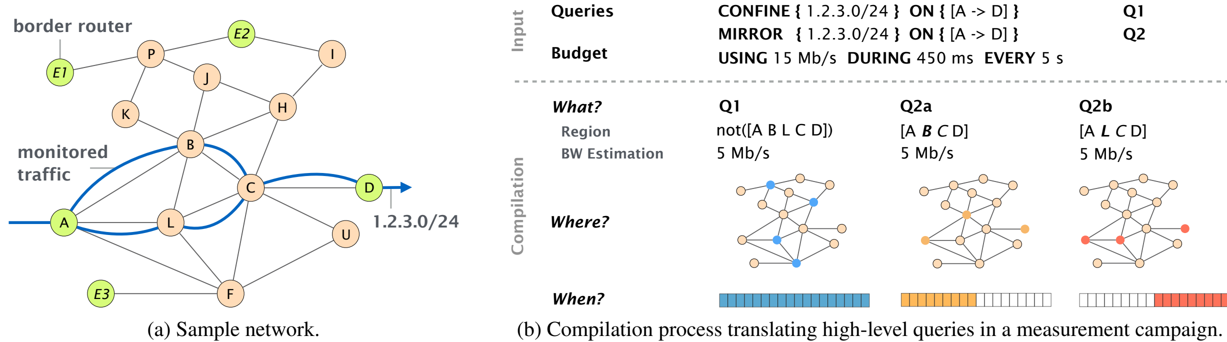
2 Overview

In this section, we provide an intuitive description of Stroboscope (see Fig. 1) using a running example. Specifically, we consider a network operator who receives complaints from customers trying to reach one prefix (1.2.3.0/24) through its infrastructure (see Fig. 2a). Following up on the complaints, the operator wants to: *(i)* check that the corresponding traffic follows the expected paths; and *(ii)* measure key performance indicators, such as packet loss rate and path latencies.

Specifying queries Stroboscope allows operators to define their monitoring goals using an SQL-like language:

```
((MIRROR | CONFINE) <prefixes>
    ON <paths> )+
 USING <Gbps> DURING <sec> EVERY <sec>
```

These monitoring queries specify for which IP prefixes, up to a single IP address, traffic should be mirrored (`MIRROR`) or confined (`CONFINE`), and where (`ON`), e.g., on a specific node, along a specified path or following the ones computed by the routing protocols (indicated with the `->` operator, e.g. *A* `->` *D*). `MIRROR` and `CONFINE` queries differ in when they mirror traffic: the former continuously mirrors traffic while the latter only mirrors traffic that *leaves* a specified region. In addition, operators can specify constraints on: *(i)* the maximum rate of mirrored traffic (`USING`) allowed; *(ii)* the duration of any measurement campaign (`DURING`); and *(iii)* the frequency at which to run measurements (`EVERY`).
Coming back to the example above, the operator can instruct Stroboscope to mirror traffic along all IGP paths between *A* and *D* using a `MIRROR` and a `->` construct (see Fig. 2b). Additionally, she can use a `CONFINE` construct to verify that these paths are the only ones carrying traffic towards 1.2.3.0/24.



(a) Sample network.

(b) Compilation process translating high-level queries in a measurement campaign.

Figure 2: From high-level monitoring queries and the current network state, Stroboscope computes a measurement campaign schedule meeting the monitoring budget.

While simple, our language supports several practical use cases. Among others, **MIRROR** queries enable network-wide path tracing, i.e., following a given packet as it traverses a sequence of nodes. Packet copies can then be analyzed by monitoring applications to estimate data-plane performance, like packet loss or path latency, or to inspect packet payloads. **CONFINE** queries are especially useful to detect unwanted forwarding behavior (e.g., traffic shifts, security policies) at runtime, and to complement information from **MIRROR** queries (e.g., on paths not taken by given traffic flows).

A three-staged compilation process From high-level queries, Stroboscope derives *measurement campaigns*, i.e., schedules of mirroring rule (de-)activations that: *(i)* provide strong guarantees on budget compliance; *(ii)* maximize accuracy by activating mirroring rules as often as possible; *(iii)* minimize the number of mirroring locations to both lower the mirrored traffic volume and decrease the control-plane overhead. Stroboscope derives these measurement campaigns in *three stages*.

Stage 1: Resolving high-level queries (§3) First, Stroboscope translates any input query into a concrete one defined on actual paths and flows. To this end, it collects routing (e.g., IGP and BGP) feeds and NetFlow records whenever available. It also maintains a *measurement database*, storing results from past monitoring campaigns. Based on this information, Stroboscope estimates per-prefix traffic volumes and computes their forwarding paths. In our example, Stroboscope estimates the traffic demand for 1.2.3.0/24 to be 5 Mbps and resolves *[A −> D]* (Q2) in two sub-queries {*[A B C D]*,*[A L C D]*} (Q2a, Q2b), one for each path.

Stage 2: Optimizing mirroring locations (§4) Second, Stroboscope minimizes the number of mirroring rules by optimizing their locations using two provably correct algorithms. Doing so, it reduces the mirrored traffic and the control-plane overhead to activate them.

The first algorithm (§4.1) optimizes the placement of **MIRROR** queries, like Q2a and Q2b in Fig. 2b. The key insight is to leverage properties of the complete network topology to prune mirroring rules. For instance, for Q2a, no mirroring rule is required on router C, as C is the only 1-hop path between B and D. By observing the TTL of mirrored packets at B and D, we can therefore be sure that traffic traversed C, without actually mirroring there.

The second algorithm (§4.2) deals with **CONFINE** queries, like Q1. The key insight is to place heavily rate-limited mirroring rules, all around the region specified in the query. This way, no packets are mirrored for correct **CONFINE** queries, and few packets per location are mirrored for incorrect queries. Our algorithm optimizes the position of surrounding rules, as exemplified in Fig. 2b. For example, the algorithm places only one mirroring rule on P to detect possible packets crossing *[A B]* and leaving the network at $E1$ or $E2$.

Stage 3: Computing measurement campaigns (§5) Third, Stroboscope schedules mirroring rules over time. These schedules use the estimated traffic volumes to meet the budget, while packing as many measurements as possible to increase monitoring accuracy. Computing such schedules is a variant of the bin-packing problem, which is *NP*-hard. To scale, Stroboscope encompasses fast approximation heuristics ($O(n \log n)$ where n is the number of queries) whose results are close to optimal. Our scheduling approach enforces deterministic sampling: packets for one specific query are mirrored from well-defined locations for a given amount of time.

In our example, Q2a and Q2b in Fig. 2 cannot be scheduled at the same time given the specified budget of 15 Mbps. Indeed, with 4 different mirroring rules, they would require a total of 20 Mbps. Stroboscope therefore schedules Q2a and Q2b each for half of the timeslots. In addition, as Q1 does not mirror any traffic unless a violation is detected, Stroboscope schedules Q1 for all the timeslots, so that any violation to Q1 can be detected.

Budget Guarantees The ability of a Stroboscope schedule to meet the budget requirements inherently depends on two assumptions, both checked at runtime. First, Stroboscope checks its demand estimations by monitoring the total traffic being mirrored and stops the measurement campaign when detecting a budget violation. Such a premature termination is enforced within one mirroring timeslot (a few milliseconds). Second, as **CONFINE** queries are not expected to mirror traffic, and only require one packet when violated, they are rate-limited.

Outputs Stroboscope's runtime (§6) carries out measurement campaigns instructing routers to mirror query-defined traffic flows for a specific amount of time. Stroboscope outputs a stream of collected packets with their meta-data (e.g., timestamp, corresponding query), meant to be processed by the operators or external applications.

3 From Abstract to Concrete Queries

Given an input query, the first operation performed by Stroboscope is to concretely define the prefix and the region to monitor. We now detail how this happens.

Resolving loosely defined regions In Stroboscope's input queries (like Q2 in Fig. 2b), regions to monitor can be specified using the -> operator. Stroboscope replaces any expression s->t with the forwarding paths from router s to router t as provided by the routing protocols (e.g., the IGP) running in the network. If no IGP path can be found, Stroboscope returns a compilation error. For example, in Fig. 2, *[A ->D]* will be translated into *[A B C D]* and *[A L C D]* if those are all the IGP forwarding paths from A to D for 1.2.3.0/24.

Whenever the -> operator is present at the start (resp. end) of a query, the Stroboscope replaces it with the set of all ingress (resp. egress) routers that receive traffic for the prefix in the query—e.g., leveraging BGP information if present, or static knowledge of all network border routers. Using this feature, the queries from Fig. 2b can be generalized to all paths terminating in D: it would be sufficient to replace *[A ->D]* with *[->D]* in the queries, *discovering* on the fly which ingresses are active. Those translations are updated at the start of each measurement campaign, so that Stroboscope performs the following measurements consistently with the latest available routing information, and flags the previous measurements if collected during routing changes.

Estimating traffic volumes In order to match the budget, Stroboscope needs information about traffic volumes for every prefix specified in the input queries. It is *fundamentally impossible* to exactly know how much traffic will be destined to any prefix ahead of measurements: in theory, any flow can unpredictably vary over time. Stroboscope does not require traffic estimation to be 100% accurate, as it includes runtime mechanisms to bound the amount of excessive traffic (see §5.2). Yet, for Stroboscope to avoid computing infeasible schedules, we would like traffic estimation to be as close to the real demands as possible. To this end, Stroboscope implements a dynamic traffic estimation technique, based on data collected during past measurement campaigns. For each prefix involved in any input query, the measurement database stores the maximum demand measured by Stroboscope over a customizable number of minutes (5, by default). Stroboscope then uses such value as a conservative estimate of the traffic that will be received for that prefix during the next iteration.

The above procedure is applicable if Stroboscope has historical data for all the queried prefixes. This condition might not hold in several cases, e.g., for prefixes not recently mirrored and those in **CONFINE** queries (for which no or few packets are mirrored, as discussed in §2). Stroboscope solves the absence of historical data in two ways. First, it can infer estimates from sampled traffic (e.g., as collected by NetFlow). In this case, Stroboscope sets the peak value recorded by random sampling as initial traffic estimation for the prefixes tracked in a significant number of samples (e.g., more than 30 in 5 minutes). This way, it exploits random sampling for what it is good at: bandwidth estimation for heavy hitters [3]. Second, for all the prefixes not covered by enough random sampling data, Stroboscope runs a specific, bootstrapping measurement campaign to estimate their traffic volume. In particular, Stroboscope reserves one minimal timeslot per prefix, and activates mirroring on all the routers in the region specified by the query (e.g., on all routers in *[A, B, C, D]* for Q2a in Fig. 2). If no traffic is captured, more timeslots are reserved to the same prefix.

We stress that the risk of significantly exceeding the budget by running bootstrapping campaigns is limited. First, those measurements are targeted to prefixes that are likely to carry a limited amount of traffic since they generated few or no observations over minutes of random sampling. In addition, traffic for each prefix is mirrored for a minimal timeslot, which would last about 25 ms in our current implementation (see §7).

Stroboscope also outputs packets collected during bootstrapping campaigns with convenient meta-data. This enables operators and special-purpose applications to select sub-prefixes of the queried destinations that best match the query purpose.

4 Optimizing Mirroring Locations

Stroboscope runs distinct algorithms to select mirroring locations for **MIRROR** (§4.1) and **CONFINE** (§4.2) queries. These algorithms minimize the number of mirroring locations while also providing *high accuracy guarantees*

of the produced measurements (e.g., packets violating **CONFINE** queries are never missed). Reducing the mirroring locations let Stroboscope: *(i)* answer more queries at the same time within the input budget; and *(ii)* decrease the control-plane overhead by changing less mirroring rules during measurement campaigns.

Stroboscope's algorithms take as input the operator-specified queries and the complete network topology, including all currently down links and nodes. Considering all possible links and nodes ensures that the algorithms always guard against all possible network paths, and never select mirroring locations breaking the accuracy guarantees due to transient topology changes.

4.1 Key-points Sampling algorithm

We developed the key-points sampling (KPS) algorithm. Stroboscope uses the KPS algorithm to select mirroring locations for **MIRROR** queries.

Goal Given a **MIRROR** query on a path P, KPS selects a set of routers which will capture traffic crossing P, while also enabling to distinguish packets forwarded outside P, which would violate the query.

Note that this goal cannot be achieved through the naive solution of mirroring only at the extremes of the path. For example, if a **MIRROR** query is defined on path *[A B C D]* in Fig. 2a would only place mirroring rules on A and D, packets forwarded over *[A B C D]* would be indistinguishable from those flowing over *[A L C D]*.

General solution By default, KPS returns all the routers in the path. This guarantees that the resulting measurement campaigns track all packets crossing any subset of routers in the path. For each mirrored packet, Stroboscope checks if there exists a sequence of routers such that the Time-To-Live (TTL) of the packet is decreased exactly by 1 at each hop in the sequence. Assuming that every router decreases packets' TTL by 1[1], a mirrored packet must have followed the path in the query if and only if Stroboscope finds such a sequence for that packet.

Optimizations KPS goes beyond this general solution whenever mirroring locations can be reduced according to the complete network topology. To this end, it exploits the following theorem, proven in Appendix A.1.

Theorem 1. *Let a forwarding path P be the concatenation of sub-paths* $Q_1,\dots,Q_n$. ***MIRROR*** *queries on P can be correctly answered by mirroring only on the endpoints* s_i *and* t_i *of all* Q_i *such that no other forwarding path from* s_i *to* t_i *has the same length as* Q_i.

As an illustration, consider Fig. 2a. The path *[A B C D]* can be seen as the concatenation of *[A B]* and *[B C D]*. Also, *[B C D]* is the only path in the topology of length 3 from B to D. Theorem 1 states that we can skip C as mirroring location. This is intuitively true because we can distinguish packets traversing *[B C D]* as the only ones whose TTL in D is equal to the TTL in B minus 2.

Algorithm Given a path P, KPS checks all the concatenations of sub-paths that result in P. For each concatenation, KPS checks if Theorem 1 holds on each sub-path, performing a depth-first search on the network graph truncated at a depth equal to the sub-path length[2]. KPS then stores the first and last router in the sub-paths compliant with Theorem 1 plus all the routers in the other sub-paths as the set of mirroring points for that concatenation. Finally, it returns a set with minimal cardinality.

For example, for the path *[A B C D]*, KPS sequentially considers the concatenations *[A B C D]*, *[A B][B C D]*, *[A B C][C D]*, and *[A B][B C][C D]*. By following this order, the first concatenation is the minimal one, as a concatenation with n elements requires at least $n+1$ mirroring locations, corresponding to the first and last routers in every sub-path. For instance, if Theorem 1 was applying to *[A B C D]*, KPS would immediately return A and D as mirroring locations. Instead, as *[A L C D]* has the same length than *[A B C D]*, Theorem 1 does not hold. This implies that at least 3 mirroring locations will be needed (e.g., *{A,B,D}* is the minimum set of locations for *[A B][B C D]*).

KPS is theoretically inefficient, since any of the depth-first search it runs can potentially explore an exponential number of paths. However, our evaluation (§7.1) shows that KPS takes milliseconds to process paths in real networks, due to their sparsity and the limited path lengths.

Stroboscope also supports **MIRROR** queries defined on *regions*, i.e., connected components of the network graph. Such queries are answered by creating sub-queries for all the paths in the region and applying the above procedure to each sub-query.

4.2 Surrounding algorithm

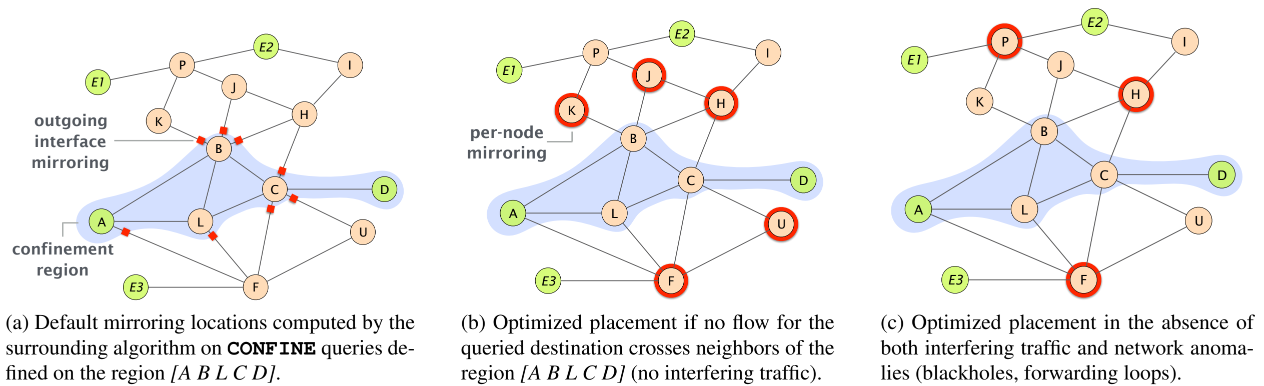
To find mirroring locations to answer **CONFINE** queries, Stroboscope runs the surrounding algorithm.

Goal Given a **CONFINE** query on a region R (which we call *confinement region*), the surrounding algorithm selects mirroring locations (routers or network interfaces) which will mirror any packet exiting the region.

Computing these locations while complying with the above goal is trickier than what it may look like. One challenge is to avoid capturing interfering traffic, that is, packets for the prefix in the query not traversing the confinement region. In Fig. 2, for example, P could not be a

[1]This assumption is consistent with the default behavior of commercial routers for both IP and MPLS packets [5, 6].

[2]The result of this check is cached, to possibly skip the depth-first search while re-evaluating the same sub-path in other concatenations.



(a) Default mirroring locations computed by the surrounding algorithm on **CONFINE** queries defined on the region *[A B L C D]*.

(b) Optimized placement if no flow for the queried destination crosses neighbors of the region *[A B L C D]* (no interfering traffic).

(c) Optimized placement in the absence of both interfering traffic and network anomalies (blackholes, forwarding loops).

Figure 3: Depending on properties of the network graph and knowledge about the correctness of the network, we can reduce the number of mirroring locations and keep the same guarantees.

mirroring location for Q1 if additional traffic (not shown in the figure) for 1.2.3.0/24 enters in *E1* and is forwarded on the path *[E1 P E2]*—remember that the region specified in the query may not include all and only the actual forwarding paths for a prefix.

General solution Given a confinement region *R*, we define the *edge surrounding* of *R* as the set of directed edges (r,n) such that $r \in R$ and $n \notin R$. By default, the surrounding algorithm returns as mirroring locations the set of outgoing interfaces of routers in *R* that correspond to any element of the edge surrounding. Fig. 3a visualizes the output of this algorithm for Q1 in Fig. 2. The following theorem (proved in Appendix A.2) states that the default output of the surrounding algorithm is correct.

Theorem 2. ***CONFINE*** *queries on a region R can be correctly answered if and only if the set of mirroring locations is the edge surrounding of R.*

Intuitively, the theorem holds because: *(i)* to exit *R*, any packet must be forwarded from a router in *R* to another outside *R*, hence over a link in the edge surrounding; and *(ii)* all the captured packets are mirrored when exiting *R*.

First optimizations Mirroring on links in the edge surrounding of the confinement region copies no packet if traffic is indeed confined to that specified region. Nevertheless, a minimal number of locations would reduce the control-plane overhead, as Stroboscope periodically reinstalls mirroring rules—keeping them alive while guaranteeing their autonomous deactivation (§6).

The surrounding algorithm uses routing information (when available) to reduce the number of mirroring locations. Knowing all the possible forwarding paths for the queried prefixes can indeed enable to safely push mirroring locations one hop away, from outgoing interfaces of routers in *R* to neighboring routers. In Fig. 3a for instance, if no forwarding path for 1.2.3.0/24 (from Fig. 2a) crosses *F*, *F* itself can be added to the set of mirroring locations and we can remove all the outgoing interfaces facing *F*—saving 2 mirroring rules.

We define the *node surrounding* of a region *R* as the set of routers that are directly connected to at least one router in *R*. Starting from the edge surrounding of *R*, the surrounding algorithm systematically replaces links ending on any router *x* in the node surrounding of *R* every time *x* is part of no forwarding path for the prefix in the query. Possibly, the entire edge surrounding is replaced by the node surrounding, as shown in Fig. 3b assuming that the region defined by *A*, *B*, *L*, *C* and *D* contains all forwarding paths for 1.2.3.0/24. A simple extension of Theorem 2 proves the correctness of this selection.

Optimal solution The surrounding algorithm further reduces the number of mirroring rules in the guaranteed absence of forwarding anomalies[3], that is, no blackholes and no forwarding loops within the monitored network.

We define a *mixed-egress path* for a region *R* as a simple path starting from a router in *R*, traversing at least one router outside *R* and ending in any egress point. The following theorem holds, as proved in Appendix A.3.

Theorem 3. *In the absence of forwarding anomalies, a* ***CONFINE*** *query on a region R can be correctly answered if and only if every mixed-egress path for R contains at least one mirroring location.* [4]

The proof of Theorem 3 is based on the fact that in general, any packet exiting a region *R* either reaches an egress point (including those in *R*), or is dropped before. In the absence of forwarding anomalies, only the former case can happen, hence it is sufficient and necessary for mirroring locations to cover all the paths ending in an

[3]This property can for example be checked by leveraging the results of other **MIRROR** queries given as input to our system.

[4]This statement does not conflict with Theorem 2, since edge and node surroundings guarantee that the condition of Theorem 3 holds.

egress point and not entirely in R. Consider, for example, Fig. 3c. If nodes P, H, and F mirror traffic, then no packet can exit the region *[A B L C D]* without traversing some mirroring location, or incurring a forwarding-anomaly—e.g., looping on some routers to re-enter the region, or be incorrectly discarded by an internal router.

Algorithm Determining the default set of mirroring locations in the presence and absence of interfering traffic mainly requires to compute edge and node surroundings, respectively: both sets can be calculated by simply iterating over all the links of the input network.

In the absence of forwarding anomaly, the surrounding algorithm returns a minimal set of locations compliant with Theorem 3. To this end, it computes a set of nodes disconnecting the input region from every egress. This is a variant of the *minimal multi-terminals cut* problem. Stroboscope solves this variant in polynomial time running the algorithm described in [7].

The algorithm in [7] requires an upper bound of the size of the cut to be computed. In our case, the cardinality of the node surrounding would provide such a bound. To further improve its efficiency, Stroboscope however computes a tighter bound by heuristically removing redundant elements from the node surrounding. It initializes the cut to the node surrounding. For every node n in the current cut, Stroboscope computes a simplified graph that does not include any node in the current cut except n, nor any link in the confinement region. For example, when considering router U in Fig. 3b, the algorithm removes F, K, J, H (as they are in the node surrounding) and all the links in the region $\{A, B, L, C, D\}$. On this simplified graph, the algorithm computes the connected component including n—which is, U, C in our example. If there is no path in this connected component between any node in the component and an egress point (as it is for U in our example), all mixed-egress paths must include at least another router in the current cut; hence, being redundant n is removed from the current cut.

5 Computing Measurement Campaigns

Combining the information on the prefix to monitor (§3) and the result of the location algorithms (§4), we end up with a group of mirroring rules for every query. The next step performed by Stroboscope is to schedule those groups of rules, producing measurement campaigns.

Answering a query requires to simultaneously activate all its mirroring rules during a given amount of time. Also, to maximize the accuracy of measurements across queries, different groups of rules should be packed together as much as possible, but respecting the traffic and time budget. We detail how Stroboscope computes a mirroring schedule in §5.1, and adapts it at runtime in §5.2.

5.1 Building a Measurement Schedule

Any schedule computed by Stroboscope is made of a finite number of timeslots, and assigns every group of mirroring rules to one or more timeslots. A *timeslot* represents an interval of time, not overlapping with any other timeslot; all the mirroring rules assigned to a timeslot s must be active during the time corresponding to s.

To meet the traffic budget, Stroboscope assigns a *cost* to every rule, reflecting the expected rate (e.g., 5 Mb/s) of traffic mirrored when the rule is active. For every **MIRROR** query on a prefix p, the corresponding rules are expected to mirror traffic for p; hence, the cost assigned to such rules is equal to the traffic rate for p, as estimated in the query pre-processing (see §3), multiplied by the number of mirroring locations (see §4.1). The cost of any **CONFINE** query is set to zero. In fact, mirroring rules for a **CONFINE** query are heavily rate-limited, hence at most a few packets per mirroring location are mirrored in the worst case—and zero if the query is correct. Note that setting the cost of **CONFINE** queries to zero implies that Stroboscope always schedules these queries in all timeslots. Stroboscope's scheduling problem then consists in assigning the groups of rules corresponding to **MIRROR** queries to every timeslot, so that the sum of the costs of all rules scheduled at every timeslot does not exceed the traffic budget defined in the queries.

Stroboscope first derives the number of timeslots, duration and spacing from the router-to-collector latencies and the monitoring time defined in the query through the **DURING** keyword. Then, to scale to a large number of queries and schedule sizes, Stroboscope splits the scheduling problem in two phases (see Fig. 4): a first phase where rules are scheduled as tight as possible, in a schedule of minimal duration; and a second phase, where the minimal schedule is replicated as much as possible, to maximize the usage of the budget, hence increasing monitoring accuracy. In both phases, Stroboscope can restrict to an approximate solution (e.g., for fast inclusion of new queries), as shown in the bottom part of Fig. 4.

Timeslot duration and spacing Timeslot durations must be long enough to ensure that packets copied at the ingress routers of any **MIRROR** query can also be mirrored at the corresponding egress routers. Stroboscope derives the duration of timeslots in a schedule from: *(i)* the minimal traffic slice duration, according to the used mirroring technology (see §6 and §7.3); and *(ii)* the observed maximal latency in the network, either defined statically or estimated as shown in §8. Also, to let in-flights packets arrive at the collector at the end of a timeslot, schedules generated by Stroboscope must include spacing between consecutive timeslots. We conservatively set this spacing to the maximum router-to-collector latency.

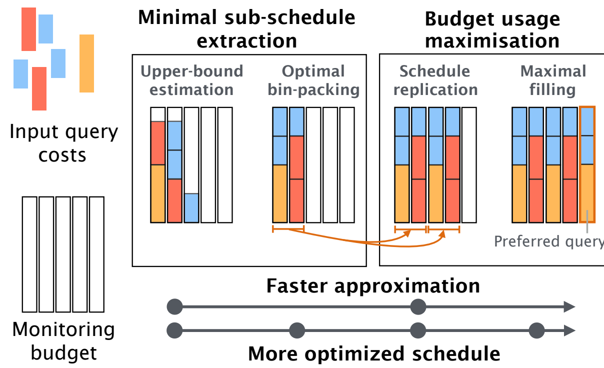


Figure 4: Stroboscope scheduling algorithm.

Minimal schedule extraction In the first phase, the scheduling algorithm assigns each group of rules to exactly one timeslot, with the goal of minimizing the number of timeslots. This is a bin packing problem, and it is therefore NP-hard. To improve time efficiency, Stroboscope first computes an upper bound on the size of the minimal schedule, using the well studied First-Fit-Decreasing heuristic—which has been proved to approximate optimum solutions with a tight bound of $\sim 1.22OPT$ [8]. The computed upper bound is then exploited to compute a minimal schedule, using a standard Integer Linear Program (ILP) formulation for bin packing problems (as detailed in Appendix B.1).

Budget usage maximization In the second phase, the scheduling algorithm replicates the minimal schedule as much as possible, increasing the number of timeslots allocated to all queries in an uniform way. The duration of the schedule might not be fully consumed by such replication—for example, the schedule in Fig.4 encompasses 5 timeslots, which allows to replicate its minimal schedule at most twice while wasting 1 timeslot. To fill the remaining timeslot(s), Stroboscope solves another ILP, whose objective function is to fit the maximum number of groups of rules into the input timeslots. Appendix B.2 contains a detailed description of this ILP.

5.2 Adapting the Schedule at Runtime

There are two possible outcomes for the scheduling just described. If Stroboscope cannot compute a schedule, it returns an error to the operator specifying the reason why the schedule could not be computed (e.g., because the time or bandwidth budget are too low). Otherwise, it starts mirroring packets according to the computed schedule. While collecting packets, it further adapts the schedule in specific cases, that we now detail.

Guarantees on limited budget overflow Stroboscope schedules rule activations to match the budget on the basis of traffic estimations which can be wrong (e.g., unpredictable traffic variations). While estimation errors can balance across different prefixes, using a static schedule comes with the risk of mirroring much more traffic than the budget if the predictions are greatly underestimating the actual traffic volume for some prefix.

To minimize budget overflow, Stroboscope tracks the total amount of traffic mirrored after every timeslot. Then, it compares such a total with the budget for 1 second (e.g., 1 Gb if the budget is 1 Gbps). Whenever the total mirrored traffic exceeds the 1-second budget, Stroboscope stops the ongoing measurement campaign, waits for the remaining time in the 1-second interval while computing a new schedule, and finally runs a new campaign. For example, if it detects that 1.1 Gb of traffic are mirrored in 0.7 seconds, for queries with a budget of 1 Gb/s, Stroboscope stops the measurement campaign, waits for 0.3 seconds, and then starts a new campaign.

Since the inter-timeslot spacing ensures that the collector receives all the mirrored packets before starting the next measurements (see §5.1), the runtime behavior just described yields the following property.

Property 1. *Stroboscope exceeds the budget in any query for at most 1 timeslot per measurement campaign.*

Note that traffic estimates are updated after the stopped campaign, so the successive campaign is much more likely not to exceed the budget again.

6 Implementation

We built a complete prototype of Stroboscope in $\sim 5,000$ lines of Python code, and 650 of C code[5]. Our implementation covers the entire compilation pipeline along with the logic to trigger mirroring rules on routers (Cisco or Linux-based), as well as benchmarks.

Mirroring packets Packet mirroring is supported by most commercial routers [9, 10]. It enables routers to duplicate packets matching given criteria (expressed using route-maps) and to send such copies to another device (e.g., over a GRE tunnel) directly in the data plane. Since packet mirroring is typically implemented in hardware, it has been experimentally shown to work at scale, with negligible CPU load and without degrading forwarding performance of mirroring routers [11].

Unfortunately, most routers only support 2 criteria to be used on all mirroring rules at the same time [9], which would prevent Stroboscope from capturing more than 2 flows per router, hence answer many real queries.

Stroboscope overcomes this limitation by indirectly triggering the mirroring of a flow, complementing mirror matching criteria with dynamic ACLs. More specifically, packet duplication primitives are pre-configured

[5] available at `https://github.com/net-stroboscope`

to match a single specific tag (e.g., a VLAN tag or a DSCP value), which we call *mirroring tag*. Stroboscope then dynamically updates ACLs to add that tag to all and only the packets to be mirrored. We use two different tag values: one for **MIRROR** queries, and another one for **CONFINE** queries which is heavily rate-limited. As **CONFINE** queries only need a single packet from a flow to report a violation, this mitigates the increase of mirrored traffic without losing information.

Our implementation activates mirroring rules by executing a pre-loaded script on each router, as readily possible in commercial routers (see, e.g., [12, 13]). The script takes two arguments: *(i)* a list of flows; and *(ii)* a mirroring duration. When invoked, it dynamically configures the ACL to tag all the packets in the input flow list. It then sets a timer based on the provided mirroring duration. On its expiration, it removes the configured ACL, deactivating the mirroring process. This technique requires only a *single interaction* between Stroboscope and the router, and *no separate deactivation message*. The deactivation after the predefined time interval is guaranteed.

Mechanisms like configuring ACLs through BGP Flowspec [14] or Netconf [15], or directly programming the IGP [16] to switch between VLANs [4], can all be used in Stroboscope, instead of our current in-router scripting approach. However, such alternatives impose a bigger overhead and cannot guarantee a slice duration (as the mirroring process has to be stopped remotely). We experimentally confirmed that our implementation can activate a large number of mirroring rules in a short amount of time (e.g., consistently with [17]), and evaluated the control over the slice duration in §7.3.

Processing mirrored packets For each mirrored packet, Stroboscope's implementation extracts: *(i)* the router ID originating it; *(ii)* its original destination IP; and *(iii)* the NIC timestamp at which it was received. At the end of each timeslot, Stroboscope outputs the collected traffic slices (possibly empty), grouped by queries, with all meta-data associated to the mirrored packets. Furthermore, it includes if packets were following the expected paths, and which packets match others.

7 Evaluation

We now evaluate our implementation of Stroboscope. First, we start by evaluating the algorithmic pipeline using synthetic benchmarks on realistic ISP topologies, to confirm that: *(i)* it can compute measurement campaigns in a timeframe suitable for online use; and *(ii)* it is able to maximize the accuracy of each query. We observe that the placement algorithms (§7.1) optimize mirroring locations in milliseconds, and reduces the number of mirroring rules by up to 50%. While the scheduling algorithm (§7.2) approximates schedules in milliseconds, optimized schedules increase accuracy by 15% for half the experiments. Second, we present measurements on real routers (§7.3) which confirm their ability to capture traffic slices as small as 23 ms. Finally, we validate the ability of Stroboscope to react to unexpected traffic changes within one timeslot using Mininet [18] (§7.4).

7.1 Placement algorithms performance

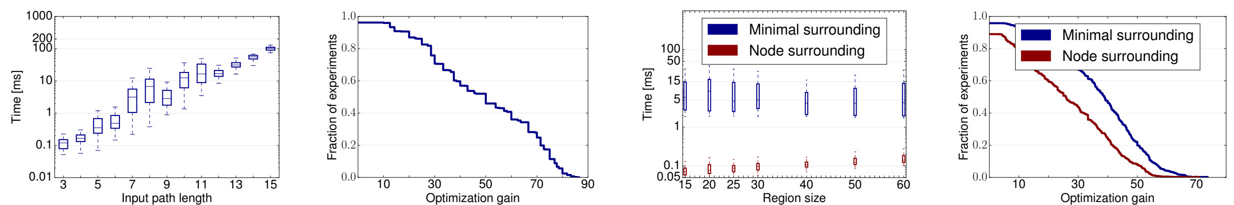
Fig. 5 shows execution time and mirroring location reduction of the placement algorithms (see §4) when run on all Rocketfuel topologies [19] and on the largest topologies from the Internet Topology Zoo [20]. We performed more than $4,000$ experiments for each algorithm. We remind that the speed of the algorithms affects Stroboscope's ability to recompute a new schedule online (i.e., to react to routing changes). However, reducing mirroring locations enables to increase the number of timeslots per schedule, hence the accuracy of query answers.

Key-points sampling (§4.1) We evaluate the algorithm by defining monitoring paths as random shortest-paths (according to the IGP weights for the Rocketfuel topologies, and edge count on the Topology Zoo ones), and random deviations from these (i.e., paths longer by up to 50% with the same end points).

Fig. 5a shows box plots of the measured execution time in function of the path length. As expected, the algorithm exhibits an exponential behavior. Yet, even for longer paths, it still completes in milliseconds: paths of 13 hops have a median runtime of only ~ 20 ms. Fig. 5b displays the CDF of the mirroring-rule reduction with respect to mirroring on every hop in the input path (i.e., $1-\frac{\text{output}}{\text{input}}$). We see that $\sim 80\%$ of the experiments resulted in a gain of over 30%. KPS returned only 2 to 4 mirroring rules in most of the experiments.

Surrounding algorithm (§4.2) We run the surrounding algorithm in similar experiments as above, except that for each topology, we randomly select connected components as regions to monitor, and 25% of the nodes having 2 or less outgoing edges as egress points.

Fig. 5c shows the measured execution times, in function of the region size. We observe that: *(i)* computing node surrounding runs in hundreds of microseconds, and is an order of magnitude faster than the further optimized placement; and *(ii)* execution times do not depend on the input region but rather on the network size and its average node degree. Fig. 5d shows the measured optimization gains with respect to edge surrounding. Both algorithms reduce the number of mirroring locations by at least 30% in half of the experiments, and the optimal one can provide an extra gain of 20%.



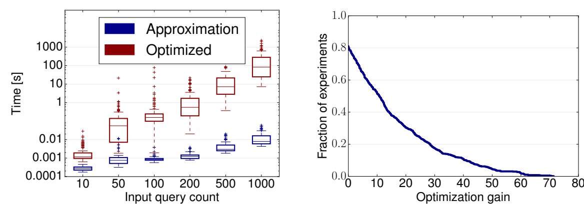
(a) The key-points sampling algorithm runs quickly, even for longer input paths in large graphs.

(b) The key-points sampling algorithm reduced the number of mirroring rules in all experiments.

(c) The surrounding algorithm running times depend on the size of the graph and not the input region.

(d) The surrounding algorithm reduce greatly the number of mirroring rules.

Figure 5: Computing mirroring rule locations is fast, and the total number of mirroring rules can be drastically reduced, minimizing the mirroring cost for each query and enabling to increase the overall accuracy.



(a) The scheduling pipeline can produce an approximation at a time-scale suited for online recomputations, even on larger inputs.

(b) By increasing the total number of slot allocations, the optimized schedule maximizes the accuracy of the measurement campaign.

Figure 6: Stroboscope can compute a quick approximated schedule, or one that maximizes accuracy.

7.2 Scheduling performance

We now evaluate the scalability of our scheduling pipeline on an increasing number of queries. We select a random, normally distributed cost for each query. We vary the time budget for all queries between 20 and 400 timeslots (corresponding to 10 s and 500 ms respectively), and the maximal bandwidth usage per slot between 2 to 100 times the average query cost. We use 10 of those selections per query size.

Fig. 6a shows the running times of the approximated and optimized scheduling algorithm. We confirm that the approximated schedule can indeed be used for online events, as it is computed in microseconds, even for $1,000$ queries. The large variance of the optimized pipeline is due to the variation of the maximal bandwidth usage across experiments. If this value is low, it increases the estimated upper bound for the bin-packing problem, which makes computing an optimized schedule exponentially slower. The optimized schedule, however, leads to improved accuracy. Fig. 6b shows the CDF of the relative increase of slot allocation (number of times a query is scheduled in a timeslot), when using the optimized pipeline instead of the approximation. For about half of the experiments the optimized schedule contains 15% more slot allocations than the approximated one, up to 40% for 10% of the experiments.

7.3 Real routers mirroring performance

We now present experimental measures on two physical routers (Cisco C7018). Each router mirrors packets to Stroboscope. We connect a traffic generator on the first router, and send test traffic towards the IP address of Stroboscope which is connected to the second router.

Slice size We first measure the minimal achievable traffic slice (i.e., activating and immediately deactivating the underlying ACL) and estimate the precision with which we could control the slice duration (by delaying the deactivation of the ACL). Fig. 7 shows the measured duration of the traffic slices depending on the deactivation delay. Each experiment is repeated 50 times. The *minimal slice duration* is $23-25\,ms$. We verify that we precisely control the duration of the traffic slice as it linearly increases with the deactivation delay.

Mirroring delay We then measure the time needed by routers to mirror packets by computing the delay between the arrival time of the original and the mirrored packet. The mean mirroring delay over roughly $100,000$ measurements is $\mu = 2.6\,\mu s$, with a standard deviation of $\sigma = 1.6\,\mu s$. Such small values indicate that routers mirror packets in constant time.

7.4 Reaction to unexpected traffic volume

Finally, we experimentally validate the ability of Stroboscope to react to unexpected traffic increases. In an emulated environment, we configured Stroboscope to mirror a flow of 1 Mb/s at two locations, using at most 5 Mb/s. We then evaluate the ability of Stroboscope to quickly adapt traffic mirroring during a sudden throughput increase. We configured the time during which recorded peak values are used for traffic estimations to 5 seconds.

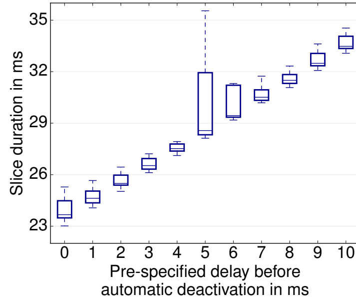


Figure 7: Existing Cisco routers (C7018) have a minimal slice duration of 23 ms.

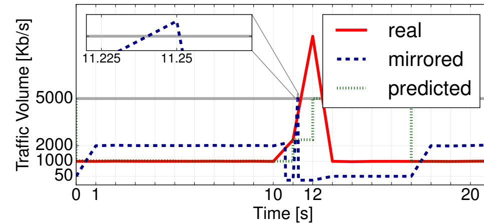


Figure 8: Stroboscope dynamically estimates traffic demands and swiftly reacts upon budget violation.

Fig. 8 shows the evolution of *(i)* the real traffic volume of the monitored flow; *(ii)* its predicted traffic demand; and *(iii)* the volume of mirrored traffic. Initially, the prediction starts at the budget value, causing little mirrored traffic as Stroboscope performs the estimation described in §3. After 1 s, the prediction is updated to reflect the last observed peak demand. This increases the amount of mirrored traffic as the query is scheduled more often. At $t = 10s$, the real traffic volume spikes, increasing the mirrored traffic. Eventually, the mirrored traffic exceeds the predicted volume, and measurements are interrupted. Traffic prediction is then updated. The same happens after $t = 11\,s$, where the query exceeds the monitoring budget during one timeslot. This causes the traffic estimation from $t = 12\,s$ to $t = 17s$ to be the whole budget, scheduling the query in a single timeslot. In total, the mirrored traffic exceeded the budget for 25 ms.

8 Case study: Monitoring transit traffic

We implemented three monitoring applications building upon the measurement stream provided by Stroboscope. Namely, we run our Stroboscope implementation on the queries from Fig. 2 and attach it to router U in the emulated network. Each link has a delay of 5 ms and a loss probability of 1%. We stress that the flow towards 1.2.3.0/24 in the example can be *any* flow—even a tiny one, extremely unlikely to be captured by NetFlow.

Estimating loss rates Stroboscope can estimate losses over paths by combining **MIRROR** and **CONFINE** queries. Indeed, there are only three reasons causing a packet captured at the ingress of a path (A) to not have a matching copy at the egress (D): *(i)* the timeslot completed before the packet reached the egress, which only happens if no packet afterwards is seen at both A and D; *(ii)* the **CONFINE** query detected a violation; or *(iii)* the packet was dropped. Using this information, we estimated loss rates across *[A −> D]* to be 7%—slightly higher than the real value (5%) as some mirrored packets were also lost.

Estimating load-balancing ratios ECMP hash function polarization [21] causes suboptimal network usage and is hard to detect. We confirmed that Stroboscope can detect such issues by computing a load-balancing ratio: in this setup, the ratio of matching packets seen at $\{A,B,C\}$ over those seen only at $\{A,C\}$, which should be close to 50%. In our case, the monitored prefix had a single flow causing the computed ratio to be about 90% (recall that there are losses in the network). This unusual ratio should prompt operators to observe the captured packet headers.

Estimating one-way delays First, Stroboscope estimates router-to-collector latencies. For that, each router has a mirroring rule matching its own loopback address. The collector sends probes towards these loopbacks, and receives copies echoed by the NIC (with no CPU fallback on the routers). Stroboscope finds the router-to-collector latency by comparing the probe and echo timestamps. Second, Stroboscope estimates one-way delays between routers (A and D) by: *(i)* identifying matching packets in their traffic slices; *(ii)* reconstructing the time at which the packets traversed each router by subtracting the router-to-collector latency from the time at which the mirrored packet copy was received at the collector; *(iii)* computing the difference between these traversal times. Using this procedure, we confirmed that the latency of *[A −> D]* was 15 ms. Note that this estimation does not require to use any form of clock synchronization between the routers and the collector.

9 Related Work

Stream-based monitoring Stroboscope relates to Gigascope [24], a stream-based system which provides a SQL-like query language to stream packet-based measurements from any router interface. In contrast to Stroboscope, Gigascope lacks higher-level constructs such as path-based queries and the ability to adhere to a monitoring budget. It also supports fewer concurrent queries as changing the packet dissectors they execute on the routers is slow.

Feature	Stroboscope	Everflow [22]	Planck [23]
Query-based mirroring	✓	✓	✗
Monitoring on a budget	✓	✗	✗
Runs on commodity hardware	✓	✓	✓
Independence from active probing	✓	✗	✓
Independence from header bits	✓	✗	✓

Table 1: Comparison between Stroboscope and other mirroring-based techniques.

Mirroring-based monitoring Stroboscope is not the first system to use packet mirroring for monitoring purposes. For example, [11] relies on packet mirroring to selectively monitor control-plane traffic. In Table 1, we compare Stroboscope with Everflow [22] and Planck [23], the two mirroring-based systems which are the closest to Stroboscope. Only Stroboscope can comply with a mirroring budget. Also, Stroboscope does not require active packet marking or special header flags as Everflow's [22] "guided probe" approach does.

Monitoring with programmable hardware Progress in programmable hardware (e.g., P4 [25]) and virtual network devices (e.g., Open vSwitch [26]) enables new monitoring possibilities. SketchVisor [27] is a sketch-based measurement framework built on virtual switches. Basat et al. [28] present a randomized constant time algorithm to identify hierarchical heavy hitters. NetQRE [29] uses regular expressions over packet streams to express flow-level and application-level policies. All three approaches could directly be built on top of Stroboscope. Other works focus on compiling high-level queries into specific actions of programmable devices. In [30, 31], path queries are supported by encoding the path traversed by packets in the packets header. Narayana et al. [32] introduce a performance query language, Marple, interacting with a key-value store running on the switches. By scheduling mirroring rules network-wide, Stroboscope supports path or Marple queries without the need for rewriting packets or special network data structures. More generally, our work shows that hardware capabilities of current routers are sufficient to build programmable monitoring systems.

Monitoring flow statistics Tools like NetFlow [1] are often used in ISP networks and provide coarse-grained flow statistics by randomly sampling traffic. FlowRadar [33] and ProgME [34] provide per-flow packet counters. While they can also bound the monitoring overhead, these approaches lack the capability of Stroboscope to track individual packets across the network, and thus cannot measure fine-grained statistics such as one-way delays or load-balancing ratios.

Data-center monitoring Many research contributions on network monitoring provide fine-grained traffic visibility in settings different from ISPs, mainly data centers. They exploit degrees of freedom that are unavailable in ISP networks, especially control of end-hosts, e.g., to collect fine-grained statistics [35] or probe the network [36]. Stroboscope is a more general in-network solution, viable in any network, *including ISP ones*. We note that some Stroboscope building blocks can be useful in other settings as well. For example, its internal algorithms could be used in a new version of Everflow [22] which keeps the mirrored traffic volume under control.

Network Verification Stroboscope complements recent initiatives in data-plane [37, 38, 30, 39, 40, 41] and control-plane [42, 43, 44, 45] verification by enabling dynamic testing of runtime-based predicates such as performance metrics (e.g., measuring packet loss). Stroboscope similarly complements recent efforts for building debugging tools for software defined networks [46, 47].

10 Conclusions

As networks grow in complexity, they require flexible monitoring tools able to measure precise metrics about their traffic flows while scaling to ever-growing traffic volumes. In this paper, we show how Stroboscope achieves these objectives by combining the visibility benefits of traffic mirroring with the scalability of traffic sampling. Specifically, Stroboscope enables to collect fined-grained measurements of any traffic flow while adhering to a monitoring budget. Stroboscope works with existing routers, and is well-suited for ISP networks. We believe that Stroboscope monitoring capabilities could address the visibility needs of many future network applications, including self-driving network control loops.

Acknowledgements

We are grateful to NSDI anonymous reviewers, our shepherd Boon Thau Loo, Lynne Salameh and Roland Meier for their insightful comments. O. Tilmans is supported by a grant from F.R.S.-FNRS FRIA. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 688421, and was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0268. The opinions expressed and arguments employed reflect only the authors' views. The European Commission is not responsible for any use that may be made of that information. Further, the opinions expressed and arguments employed herein do not necessarily reflect the official views of the Swiss Government.

References

[1] Benoit Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004. http://www.ietf.org/rfc/rfc3954.txt.

[2] Peter Phaal, Sonia Panchen, and Neil McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176 (Informational), September 2001. http://www.ietf.org/rfc/rfc3176.txt.

[3] Yin Zhang, Sumeet Singh, Subhabrata Sen, Nick Duffield, and Carsten Lund. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *SIGCOMM*, pages 101–114, 2004.

[4] Olivier Tilmans, Tobias Bühler, Stefano Vissicchio, and Laurent Vanbever. Mille-Feuille: Putting ISP Traffic Under the Scalpel. In *HotNets*, pages 113–119, 2016.

[5] Jon Postel. Internet protocol darpa internet program protocol specification. RFC 791, September 1981. https://tools.ietf.org/rfc/rfc791.txt.

[6] Puneet Agarwal and Bora Akyol. Time To Live (TTL) Processing in Multi-Protocol Label Switching (MPLS) Networks. RFC 3443, January 2003. https://tools.ietf.org/rfc/rfc3443.txt.

[7] Jianer Chen, Yang Liu, and Songjian Lu. An improved parameterized algorithm for the minimum node multiway cut problem. *Algorithmica*, 55:1–13, 2009.

[8] György Dósa. The tight bound of first fit decreasing bin-packing algorithm is ffd (i) $\leq$ 11/9 opt (i)+ 6/9. *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11, 2007.

[9] Cisco Systems. Configuring ERSPAN, 2016. https://goo.gl/h3qaGL.

[10] Juniper Networks. Layer 2 Port Mirroring Overview, 2014. https://goo.gl/YxgZuY.

[11] Stefano Vissicchio, Luca Vergantini, Luca Cittadini, Valerio Mezzapesa, Maurizio Pizzonia, and Maria Luisa Papagni. Beyond the Best: Real-time Non-invasive Collection of BGP Messages. In *INM/WREN*, 2010.

[12] Cisco Python API. http://bit.ly/2fMgyKP.

[13] Junos Automation Scripts Overview, 2017. https://goo.gl/WpjAcX.

[14] Pedro Marques, Nischal Sheth, Robert Raszuk, Barry Greene, Jared Mauch, and Danny McPherson. Dissemination of Flow Specification Rules. RFC 5575 (Proposed Standard), August 2009. http://www.ietf.org/rfc/rfc5575.txt.

[15] R. Enns et al. NETCONF Configuration Protocol. RFC 4741, December 2006. https://tools.ietf.org/rfc/rfc4741.txt.

[16] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central Control Over Distributed Routing. In *SIGCOMM*, pages 43–56, 2015.

[17] A. Bobyshev, P. DeMar, and D. Lamore. Effect of dynamic ACL (access control list) loading on performance of Cisco routers. In *Computing in High Energy Physics*, 2004.

[18] Mininet: An Instant Virtual Network on your Laptop (or other PC). 2012. http://www.mininet.org/.

[19] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP Topologies with Rocketfuel. In *SIGCOMM*, pages 133–145, 2002.

[20] Simon Knight, Hung X. Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29:1765 –1775, 2011.

[21] Cisco Tech Support. CEF Polarization. 2013. https://goo.gl/b7ZSMy.

[22] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-Level Telemetry in Large Datacenter Networks. In *SIGCOMM*, pages 479–491, 2015.

[23] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *SIGCOMM*, pages 407–418, 2014.

[24] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD*, pages 647–651, 2003.

[25] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44:87–95, 2014.

[26] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The Design and Implementation of Open vSwitch. In *NSDI*, pages 117–130, 2015.

[27] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust Network Measurement for Software Packet Processing. In *SIGCOMM*, pages 113–126, 2017.

[28] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo C. Luizelli, and Erez Waisbard. Constant Time Updates in Hierarchical Heavy Hitters. In *SIGCOMM*, pages 127–140, 2017.

[29] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative Network Monitoring with NetQRE. In *SIGCOMM*, pages 99–112, 2017.

[30] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *NSDI*, pages 99–111, 2013.

[31] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling Path Queries. In *NSDI*, pages 207–222, 2016.

[32] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. In *SIGCOMM*, pages 85–98, 2017.

[33] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A Better NetFlow for Data Centers. In *NSDI*, pages 311–324, 2016.

[34] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. Progme: towards programmable network measurement. *IEEE/ACM Transactions on Networking (TON)*, 19:115–128, 2011.

[35] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and Precise Triggers in Data Centers. In *SIGCOMM*, pages 129–143, 2016.

[36] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*, pages 139–152, 2015.

[37] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. *SIGCOMM Comput. Commun. Rev.*, 41:290–301, 2011.

[38] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, pages 113–126, 2012.

[39] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *NSDI*, pages 49–54, 2013.

[40] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking Beliefs in Dynamic Networks. In *NSDI*, pages 499–512, 2015.

[41] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable Symbolic Execution for Modern Networks. In *SIGCOMM*, pages 314–327, 2016.

[42] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A General Approach to Network Configuration Analysis. In *NSDI*, pages 469–483, 2015.

[43] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis Using an Abstract Representation. In *SIGCOMM*, pages 300–313, 2016.

[44] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. *ACM SIGPLAN Notices*, 51:765–780, 2016.

[45] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *SIGCOMM*, pages 155–168, 2017.

[46] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC*, pages 15–17, 2011.

[47] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *NSDI*, pages 71–85, 2014.

A Proofs for the placements algorithms

A.1 Proof for Theorem 1

Theorem. *Let a forwarding path P be the concatenation of sub-paths $Q_1, \dots, Q_n$.* **`MIRROR`** *queries on P can be correctly answered by mirroring only on the endpoints s_i and t_i of all Q_i such that no other forwarding path from s_i to t_i has the same length as Q_i.*

Proof. First, we show that if any $Q \subseteq P$ is the only path of length x from p to s, we can always distinguish mirrored packets that have been forwarded over Q by just mirroring on p and s. Let l be the length of Q, and let t_p and t_s be the TTL values of any packet mirrored from p and s. Under the assumption that the TTL is properly decreased (by one) at each forwarding hop, we can unambiguously determine if the packet has been forwarded over Q: If $t_s = t_p - l$, then Q must be the traversed sub-path because Q is the only path of length l between p and s by hypothesis.

The statement of the theorem then follows by noting that the same property applies to any sub-path Q_i, as well as to their concatenation—i.e., P. □

A.2 Proof for Theorem 2

Theorem. **`CONFINE`** *queries on a region R can be correctly answered if and only if the set of mirroring locations is the edge surrounding of R.*

The following lemma proves Theorem 2.

Lemma 1. *Given a region R, its edge surrounding $E(R)$ and any prefix d, it is guaranteed to capture all and only packets for d that exit R if and only if mirroring rules matching d are active on every edge in $E(R)$.*

Proof. We first show that all packets exiting R are captured if and only if rules are placed on all the edges in $S(R)$. Consider any packet p entering from an ingress node i in R. For p to exit R, there must be a node r in R (possibly $r = i$) that forwards p to a node o outside R. The packet will then traverse the edge (r, o) with $r \in R$ and $o \notin R$. If mirroring rules are active on all the edges in $S(R)$, then p is detected, by definition of $S(R)$. In contrast, if a mirroring rule is not active on any edge (r_m, o_1) with $r_m \in R$ and $o_1 \notin R$, then a packet p' will not be mirrored if it exits R through (r_m, o_1) and never enters R again – e.g., following a path $[r_1 \dots r_m o_1 \dots o_k]$, where $\forall i = 1, \dots, m\ r_i \in R$ and $\forall j = 1, \dots, k\ o_j \notin R$.

In addition, if a packet is captured by a rule placed on an edge (x, y) in $S(R)$, then it must have crossed a node x in R and be forwarded to a node y outside R, by definition of $S(R)$. This implies that only packets leaving R are mirrored, which yields the statement. □

A.3 Proof for Theorem 3

Theorem. *In the absence of forwarding anomalies, a* **`CONFINE`** *query on a region R can be correctly answered if and only if every mixed-egress path for R contains at least one mirroring location.*

The following lemma proves Theorem 3.

Lemma 2. *In the absence of forwarding anomalies, any packet not confined to a region R is guaranteed to be mirrored if and only if every simple path starting from a node in R, traversing a node outside R and ending in any egress point crosses at least one active mirroring rule matching the packet destination.*

Proof. We separately prove sufficiency and necessity of the condition expressed by the theorem.

Sufficiency: Proof by contradiction. Assume that some packets not confined to R are not mirrored despite mirroring rules matching the condition in the theorem statement. In the absence of forwarding anomalies, those packets are guaranteed to be delivered to an egress point. Not to be confined to R, they must follow a path $[r_1 \dots r_n o_1 \dots o_l \dots e]$, where e is an egress point, nodes $r_i \in R\ \forall i = 1, \dots, n$, and nodes $o_j \notin R\ \forall j = 1, \dots, l$. By hypothesis, an active mirroring rule must be on this path and must mirror the packets, contradicting the assumption that packets are not mirrored.

Necessity: Proof by contradiction. Assume that it is guaranteed to mirror all packets confined to R but no mirroring rule is active on a given path $P = [r_m \dots o \dots e]$ from a node $r_m \in R$ to an egress point e including a node $o \notin R$ (possibly $o = e$). Consider now any path $[r_1 \dots r_m]$, where $m \geq 1$, r_1 is an ingress point, and $r_i \in R$ $\forall i = 1, \dots, m$. This path must exist since a region is defined as a connected component (see §2). Packets forwarded on the concatenation of the previous two paths (i.e., $[r_0 \dots r_m \dots o \dots e]$) are not confined to R, as they cross $o \notin R$. However, they are not mirrored, contradicting the assumption. □

B Scheduling ILP formulations

B.1 Optimal bin-packing

Input All queries and their associated costs, an upper bound on the number of of timeslots needed.

Decision Variables Let Q be the set of input queries and S the set of all time slots in the measurement campaign. We define:

R_{qs} as the binary variable representing the decision to schedule the query $q \in Q$ in timeslot $s \in S$ when $R_{qs} = 1$;

U_s as the binary variable representing whether the timeslot $s \in S$ has any assigned query when $U_s = 1$.

Parameters

B The maximal available bandwidth in a single timeslot;

a_q The expected traffic volume generated by the mirroring rules for the query q.

Objective Function Minimize the length of the sub-schedule

$$\min \sum_{s}^{S} U_s$$

Constraints

C1 In any timeslot s, the expected traffic generated by the mirroring rules across all queries activated in s must be lesser or equal than the budget.

$$\forall s : \sum_{q}^{Q} (R_{qs} a_q) \leq U_s B$$

C2 Every query must be scheduled.

$$\forall q : \sum_{s}^{S} R_{qs} = 1$$

C3 Track used slots.

$$\forall q,\ s : U_s \geq R_{qs}$$

C4 Timeslots should be used in sequence (tie-breaking constraint).

$$\forall s,\ s',\ s < s' : U_s \leq S_{s'}$$

B.2 Maximal filling

Input A list of queries and their associated cost, a list of time slot and leftover budget. Queries are pruned such that any query whose cost is greater than the biggest leftover budget available is excluded.

Decision Variables Let Q be the set of input queries and S the set of all time slots in the measurement campaign. We define:

R_{qs} as the binary variable representing the decision to schedule the query $q \in Q$ in timeslot $s \in S$ when $R_{qs} = 1$;

M as the continuous variable representing the minimal number of slots allocated to any query.

Parameters

β_s The available leftover bandwidth in the timeslot s, thus $\beta_s \leq B$;

Ω The spreading factor, which lets the operator favor schedules where all queries have a similar number of timeslots (high value) or schedules maximizing the absolute number of allocation;

w_q The preference level of the query q. Queries with a higher preference are scheduled preferably to queries with a lower preference;

a_q The expected traffic volume generated by the mirroring rules for the query q.

Objective Function Maximize the utilization of the budget, either by maximizing the number of allocations of some queries, according to their preference level, or by spreading the budget across all queries (thus maximizing the minimal allocation).

$$\max \left[\sum_{q}^{Q} \left(\sum_{s}^{S} R_{qs} \right) w_q + M\Omega \right]$$

Constraints

C1 In any timeslot s, the expected traffic generated by the mirroring rules across all queries activated in s must be lesser or equal than the leftover budget.

$$\forall s : \sum_{q}^{Q} (R_{qs} a_q) \leq U_s \beta_s$$

C2 M should represent the minimal number of allocated slots across all queries.

$$\forall q : M \leq \sum_{s}^{S} R_{qs}$$

PLOVER: Fast, Multi-core Scalable Virtual Machine Fault-tolerance

Cheng Wang,[*] Xusheng Chen,[*] Weiwei Jia, Boxuan Li,
Haoran Qiu, Shixiong Zhao, and Heming Cui
The University of Hong Kong

Abstract

Cloud computing enables a vast deployment of online services in virtualized infrastructures, making it crucial to provide fast fault-tolerance for virtual machines (VM). Unfortunately, despite much effort, achieving fast and multi-core scalable VM fault-tolerance is still an open problem. A main reason is that the dominant primary-backup approach (e.g., REMUS) transfers an excessive amount of memory pages, all of them, updated by a service replicated on the primary VM and the backup VM. This approach makes the two VMs identical but greatly degrades the performance of services.

State machine replication (SMR) enforces the same total order of inputs for a service replicated across physical hosts. This makes *most* updated memory pages across hosts the same and they do not need to be transferred. We present Virtualized SMR (VSMR), a new approach to tackle this open problem. VSMR enforces the same order of inputs for a VM replicated across hosts. It uses commodity hardware to efficiently compute updated page hashes and to compare them across replicas. Therefore, VSMR can efficiently enforce identical VMs by transferring only divergent pages. An extensive evaluation on PLOVER, the first VSMR system, shows that PLOVER's throughput on multi-core is 2.2X to 3.8X higher than three popular primary-backup systems. Meanwhile, PLOVER consumed 9.2X less network bandwidth than both of them. PLOVER's source code and raw results are released on `github.com/hku-systems/plover`.

1 Introduction

The cloud computing paradigm enables a pervasive deployment of online services in virtualized infrastructures (e.g., Xen [26]). Meanwhile, a virtual machine (VM) is incorporating more and more virtual CPUs (vCPU) on multi-core hardware because online services process many requests concurrently. This rapid growth of cloud computing components implies that hardware failures become commonplace [18] rather than occasional. A fast and multi-core scalable VM fault-tolerance approach is highly desirable for online services.

Primary-backup (e.g., REMUS [36]), a dominant VM fault-tolerance approach, works in a physical time slot manner. In each slot, it runs a service in the primary VM to process client requests, tracks updated VM states (e.g., dirty memory pages), and buffers network outputs. When a slot ends, a `syncvm` operation is invoked to transfer dirty pages from the primary to backup. Once the transfer succeeds, network outputs are sent to clients. By doing so, primary-backup ensures *external consistency* [36]: primary and backup have the same states and a primary failure will not be observed by clients.

Unfortunately, despite much effort [13, 36, 42, 64, 82], achieving fast and multi-core scalable VM fault-tolerance remains an open problem [5, 38, 42, 82] . A main reason is that the primary-backup approach often has to transfer an excessive amount of dirty memory pages, which greatly degrades the performance of a service and occupies prohibitive network bandwidth.

For instance, if a program updates 20K dirty memory pages within a 100ms slot, transferring these pages consumes a huge network bandwidth of 6.4 Gbps. Both our evaluation (§6) and prior study [36, 40, 42, 60] show that many programs access even more dirty pages on over four CPU cores. vSphereFT-6.5 [17], a latest primary-backup product, permits up to four vCPUs per VM and only two of such VMs per physical host [5]. Therefore, to enable fault-tolerance, people often sacrifice multi-vCPU speedup and VM consolidation [34].

As a service includes multiple programs (e.g., a website deployed in one VM can include an `Nginx` web server, a Python interpreter, and `MySQL`), and a program scales better on more CPU cores and accesses more memory, this problem becomes even more challenging.

Another approach, state machine replication (SMR), appears a promising solution for this open problem. SMR [68] models a program as a deterministic state machine and replicates it on different physical hosts (or replicas). It uses a distributed consensus protocol (typically, PAXOS [56]) to enforce the same total order of program inputs across replicas, making them perform the same sequence of state transitions. SMR systems [35, 45, 50] often incur low performance overhead with popular programs on 16 CPU cores.

However, to ensure external consistency, SMR requires extra mechanisms to resolve divergent executions (i.e., multithreading nondeterminism [60]) across replicas. Extisting SMR systems provide two major mechanisms. First, EVE [50] requires program developers to manually annotate variables shared by threads, and it detects divergent variable states at runtime. Second,

[*]The first two authors contributed equally to this work.

Rex [45] and Crane [35] enforce the same order of inter-thread synchronization (e.g., locks) across replicas. If no data race occurs, determinism is ensured; otherwise, developers' diagnosis [51, 75] may be needed. Therefore, neither of the two mechanisms is fully automatic.

Our key observation is that by enforcing the same total order of inputs for a VM replicated across hosts, almost all updated memory pages across the hosts are the same and they do not need to be transferred. Intuitively, if a VM containing a key-value service is replicated across hosts and it receives the same order of requests, all these hosts should contain roughly the same data in memory. Empirically, we enforced the same total order of client requests for 8 diverse services running on two VMs, and 72% to 97% of the services' dirty pages were the same after processing these requests.

This paper presents Virtualized SMR (VSMR), a new SMR approach that can achieve fast, multi-core scalable VM fault-tolerance. VSMR enforces same total order of network inputs for a VM replicated across hosts. It then periodically invokes a `syncvm` operation to efficiently compute updated page hashes, to compare them across the replicas, and to transfer only the divergent pages.

In a conceptual level, VSMR replicates an entire guest VM as a state machine and achieves the strengths of both SMR and primary-backup. By transferring only those divergent pages, VSMR automatically and efficiently ensures external consistency. Leveraging the powerful fault-tolerance of Paxos, VSMR tackles a notorious "split-brain problem" (§2.2) in primary-backup systems.

We implemented Plover,[1] the first VSMR system in Linux. Plover uses Apus [92], a fast, RDMA-powered Paxos implementation. Plover intercepts inbound network packets in the KVM QEMU hypervisor [80] and replicates them to other VM hypervisors using Paxos. Plover's `syncvm` operation (§4) is built on top of Paxos for robustness, and it uses RDMA to efficiently compare page hashes across replicas. Plover does not modify the underlying Paxos protocol, so it is generic to work with other fast consensus protocols [58, 78].

We evaluated Plover on 12 widely used programs, including 8 servers (e.g., `SSDB` [85] and `Tomcat` [3]) and 4 dynamic language interpreters (e.g., PHP). We group these programs into 8 practical services, including DjCMS [7], a content management system (CMS) that consists of `Nginx` [73], Python, and `MySQL` [22]. We compared Plover with three well-engineered primary-backup systems QEMU-MicroCheckpoint [13] (for short, MC) , Colo [38], and STR [63]. Evaluation shows that:

1. On average, Plover's throughput is 2.2X higher than MC, STR, and Colo on 4-vCPU VMs, 3.8X higher on 16-vCPU VMs. Compared to unreplicated executions, Plover's overhead on response time is modest. Plover has reasonable CPU usage.

2. Plover consumes 9.2X less network bandwidth than both MC, STR, and Colo on average. It enables consolidating multiple fault-tolerant VMs on one host.

3. Plover is robust to various failures.

Our major contribution is VSMR, a new SMR approach, which automatically achieves much faster and more scalable VM fault-tolerance. Our other contributions include the Plover implementation and an extensive evaluation on diverse, sophisticated online services. Moreover, by efficiently enforcing the same VM across hosts, Plover can be broadly applied to other research areas. For instance, page-level false-sharing [28, 61] is a notorious performance problem in multithreading replay [40, 54, 67]. Plover can be an effective template to alleviate this problem, because most false-shared pages across the record and replay hosts should have the same contents and they do not need to be transferred.

The remaining of the paper is organized as follows. §2 introduces the background of RDMA, VM, and Paxos. §3 gives an overview on Plover's architecture and its advantages over the primary-backup approach. §4 presents Plover's runtime system. §5 describes implementation details, §6 presents evaluation results, §7 introduces related work, and §8 concludes.

2 Background

2.1 RDMA

RDMA (Remote Direct Memory Access) [2] can directly write from the userspace memory of a host to the userspace memory of a remote host, bypassing the OS and CPU on both hosts. RDMA architectures (e.g., Infiniband [2] and RoCE [11]) are commonplace within a datacenter due to their ultra low latency and decreasing costs. RDMA's ultra low latency comes from not only its OS bypassing feature, but also its dedicated network stack implemented in hardware. RDMA latency is several times smaller than software-only OS bypassing techniques (e.g., DPDK [6] and Arrakis [77]).

The advantage of RDMA latency is especially significant when transferring messages of small sizes. Benchmarks [4, 10] show that, with the same network interface card (NIC), transferring messages of less than 2KB on RDMA is about 10X∼30X faster than on TCP. If the message size becomes larger (e.g., over 8KB), RDMA latency is merely about 30% faster than TCP because network bandwidth becomes a bottleneck for both. This

[1] The Pacific golden plover is well known for her strong tolerance to the extreme weather in Alaska.

suggests that RDMA is attractive for invoking consensus on inputs and sending hashes of memory pages, and it is less beneficial for transferring pages. PLOVER uses RDMA for invoking PAXOS consensus, exchanging page hashes across replicas, and transferring divergent pages.

2.2 Virtual Machine and Its Fault-tolerance

VMs [26, 52, 91] are widely used in clouds and datacenters due to their low performance overhead [42], platform independence, performance isolation [47], etc. For instance, KVM [52] is an accelerator that uses the hardware virtualization features of various CPUs, while QEMU [80] emulates the hardware for VMs. PLOVER uses KVM-QEMU for three main reasons. First, KVM-QEMU incurs little performance overhead compared to bare-metal. Second, QEMU works in userspace and is suitable for RDMA-based PAXOS to intercept inputs (RDMA currently only supports userspace memory). PLOVER uses QEMU's `tap_send()` API to intercept network inputs. Third, the QEMU virtual threads that act as vCPUs are spawned from the QEMU main process, which enables PLOVER to monitor programs running in a guest VM non-intrusively [87] without modifying the guest.

Moreover, VM platform independence enables consolidation [34]: people can migrate many VMs [32, 72] to a small number of physical hosts to save energy and ease management. However, consolidation also implies that many VMs are prone to hardware failures. Therefore, a fast, scalable, and network bandwidth friendly VM fault-tolerance approach is highly desirable.

Existing VM fault-tolerance systems [13, 17, 36, 38, 63, 64, 82] are mainly based on the primary-backup approach. To maintain external consistency, the primary must transfer the dirty memory modified by a program within one time slot to the backup before releasing outputs, the so called "output commit problem" [86]. Therefore, the major performance bottleneck of this approach is the time taken to transfer dirty pages, because local memory access speed can be 10X$\sim$100X faster [14] than network speed. As real-world programs become increasingly scalable on multi-core and access more memory per second, this bottleneck becomes even more significant. An evaluation [42] shows that this transfer time can be much bigger than a `syncvm` time slot, greatly degrading the performance of services.

Synchronization Traffic Reduction (STR) [63] is a heuristic for reducing the number of transferred pages. It runs both primary and secondary VMs in parallel to process the same network inputs in the same order. STR uses a 25ms `syncvm` interval and only transfers divergent pages in each `syncvm` operation. However, both our evaluation and STR's show that this heuristic is ineffective because of the static `syncvm` interval (§6.2).

COLO is a primary-backup system deployed in Huawei [20]. It runs the same service on both primary and backup, compares per-connection network outputs, and does a `syncvm` if there is any network output divergence. COLO can safely skip the `syncvm` operation if network outputs remain identical. Nevertheless, both our evaluation and COLO's show that its performance severely degrades when the number of client connections is large.

vSphereFT used to take a record-replay approach [29, 83] for uni-vCPU, but it switches to the REMUS approach since vSphereFT 6.0 [9, 17]. If fault-tolerance is enabled, vSphereFT permits at most four vCPUs per VM and only two of such VMs per host [5]. This affects multi-vCPU speedup and VM consolidation.

Since primary-backup has only two replicas, when network partition occurs, neither the primary nor backup can determine whether the other one fails forever or is temporarily partitioned. Therefore, they both may serve client requests, breaking external consistency. This is the notorious "split-brain problem" [23, 24, 83].

2.3 PAXOS and SMR Systems

PAXOS [55, 56, 68] is a major protocol to enforce the same, totally ordered inputs across replicas. For efficiency, typical PAXOS implementations [68, 74] take the Multi-Paxos approach [55]: it elects a dedicated leader in each view to invoke consensus on new inputs, and other replicas work as witnesses to agree on inputs. In PAXOS, the value of each agreed input is flexible, and PLOVER takes advantage of this flexibility. PAXOS can be used to maintain different roles consistently for different replicas [58, 71], and replicas with different roles can interpret the same agreed input value differently according to the (consistent) roles. E.g., the leader of NOPaxos [58] executes inputs; its witnesses agree on inputs and interpret inputs as no-operation (NOP).

To maintain roles for replicas consistently, PAXOS replicas send periodical heartbeats [68, 74] to other replicas and track the number of heartbeat failures with a threshold. If a replica finds that its threshold is reached, it suspects the replica on the other end failed and it invokes a new consensus (e.g., leader election); otherwise, a replica can safely intercept inputs or logical operations on its own safely. During leader election, the node with the most up-to-date state wins [74, 78].

Three recent SMR protocols, NOPaxos [58], APUS [92], and DARE [78] incur a low consensus latency of tens of μs. PLOVER uses APUS for three main reasons: (1) it provides a flexible `paxos_op(void *val)` API to propose a consensus request with `val` as the proposed value; (2) its consensus protocol includes a durable storage (DARE works purely in memory); and (3) it is open source.

3 Overview

3.1 Deployment Suggestion

PLOVER's deployment follows typical SMR systems: three replicas are connected with RDMA networks, and each replica runs a PLOVER VM instance containing a set of programs. We suggest each replica have 16+ CPU cores. By running three replicas, PLOVER can tolerate hardware failures or network partitions of one replica. This fault-tolerance guarantee is sufficient because: (1) a VM can already tolerate various failures in guest OS, and (2) tolerating one failure is a common guarantee in VM fault-tolerance systems [17, 36].

We suggest more CPU cores because PLOVER uses spare cores to compute dirty page hashes. Our evaluation used 24-core hosts and PLOVER performance was already reasonable. In addition, RDMA becomes prevalent [69, 78]. RDMA is just a requirement for current PLOVER implementation, not a requirement for VSMR. One can implement VSMR using other fast PAXOS protocols (e.g., NOPaxos [58]) and using other OS bypass techniques (e.g., Arrakis [77]) to send page hashes.

3.2 PLOVER Architecture

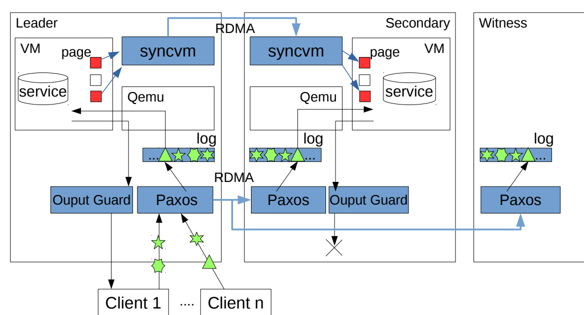


Figure 1: PLOVER *Architecture.* Key components are in blue, inputs are in green, divergent dirty pages are in red.

We designed PLOVER to be simple and generic for various PAXOS implementations. To this end, PLOVER has two unique features compared to regular SMR systems.

First, unlike regular SMR systems which maintain two replica roles (leader and witness), PLOVER invokes PAXOS to consistently maintain three replica roles: leader, secondary, and witness. In PLOVER's underlying PAXOS level, both PLOVER's secondary and witness are "PAXOS witnesses" which simply agree on consensus requests. The only difference is about interpreting `syncvm` in the upper PLOVER system level: the PLOVER leader and secondary involve the `syncvm`, and the PLOVER witness interprets `syncvm` as NOP. We made this design choice because transferring divergent pages to only the secondary is efficient.

Second, to minimize service downtime during the leader's failures, rather than letting the remaining nodes compete to be the new leader, PLOVER elevates its secondary to be the leader because the secondary's state is more up-to-date than the witness's. PLOVER has the same safety guarantee as PAXOS by ensuring there is one unique leader in each view and all the replicas are consistent with their roles (§4.5). To preserve the fault tolerance guarantee, the new leader will do a VM migration to the witness, elevate the witness to be the secondary, and then begin to serve client requests.

Figure 1 shows PLOVER's architecture with four key components: the PAXOS input coordinator (PAXOS), the consensus log (*log*), the output buffering guard (*guard*), and the `syncvm` component. The PAXOS coordinators reside in all three replicas to maintain a consensus log with the same order of SMR operations, including input requests, `syncvm`, and role changes (§4.2).

When PLOVER starts, PAXOS elects one replica as the leader, which is dedicated to receive and make consensus on client requests. When the leader receives a new network input, it invokes PAXOS to replicate this input on PLOVER's replicas. §6.3 shows that, by enforcing the same total order of realistic workload inputs for different VM replicas for 8 services, 72% ∼97% of the programs' memory are already the same and do not need to be transferred.

The leader periodically invokes consensuses on `syncvm` operations to synchronize the VM states of the VMs. PLOVER uses an adaptive algorithm to determine the intervals between two `syncvm` operations based on current workloads, which effectively reduces transferred states and improves performance (§4.3). On successful consensus on a `syncvm` operation, the `syncvm` components of the leader and secondary interpret it with three steps: (1) they exchange dirty page bitmaps and compute hashes of each dirty physical pages concurrently; (2) the leader receives hashes from the secondary and compare hashes; (3) the leader transfers only the divergent pages. §4.4 describes our `syncvm` protocol in detail.

The guards on both leader and secondary buffer network outputs since the last `syncvm` operation. When a new `syncvm` succeeds, the leader's guard releases outputs to clients, while the secondary discards outputs.

PLOVER ensures external consistency. Suppose the leader fails in the n_{th} slot (i.e., PLOVER has finished $n-1$ `syncvm` operations), the secondary becomes the new leader, and the old leader and the new leader have the same states in the last $n-1$ slots. Since the old leader's output in the n_{th} slot has not been released by PLOVER, clients will not observe any inconsistency even if the new leader's state in the n_{th} slot differs from the old leader's. Thus, the new leader can take over without perturbing clients.

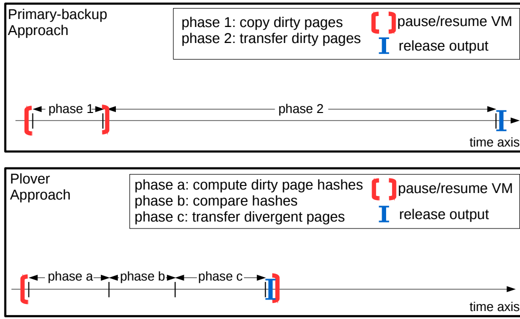


Figure 2: ***Comparing*** `syncvm` ***operation in VSMR and MC.***

3.3 Comparing PLOVER and primary-backup

We design PLOVER to gain the same fault-tolerance strength as regular SMR. By using PAXOS to maintain the roles of replicas, PLOVER can consistently maintain a leader. By running three PAXOS replicas, PLOVER is also able to consistently detect an outdated leader caused by transient network partitions. In contrast, primary-backup is known unable to handle network partition, the so called "split-brain problem" (§2.2). Because hardware failures may cause transient network partitions (e.g., NIC or network switch errors), PAXOS's strong fault-tolerance is increasingly useful.

To illustrate why PLOVER can be faster than a typical primary-backup approach, Figure 2 shows the leader or primary's workflow in PLOVER and in MC [13], a recent REMUS-based implementation developed in QEMU [80]. Within a primary-backup `syncvm` operation, the time taken in MC's primary can be divided into two major phases: (1) t_{copy}, time taken for copying dirty pages to another memory region; and (2) $t_{transfer}$, time taken for transferring dirty pages. The primary resumes its guest VM after phase (1), but it must release outputs after phase (2) for external consistency. $t_{transfer}$ is often much longer than t_{copy} and becomes the bottleneck (§6.2).

The time taken in a PLOVER leader's `syncvm` operation can be divided into three major phases: (a) $t_{compute}$, time taken to compute hashes for local dirty pages; (b) $t_{compare}$, time taken to compare hashes between leader and secondary; and (c) $t_{divergent}$, time taken to transfer only divergent pages. PLOVER resumes its guest VM and releases outputs after transferring the divergent pages. PLOVER resumes the guest VM after the transfer because it saves the page copy time by using RDMA to directly write divergent pages to the secondary.

Compared to primary-backup, PLOVER's phase (a) can be fast by leveraging CPU cores, and phase (b) can be fast by leveraging RDMA. Phase (c) can be fast if most dirty pages between the PLOVER leader and secondary are the same. Our evaluation shows that PLOVER's $t_{divergent}$ is up to 12.8X faster than MC's $t_{transfer}$.

API	Argument	API Semantic
`paxos_second`	secondary ID	Propose the secondary
`paxos_input`	input packets	Propose client requests
`paxos_syncvm`	`syncvm`	Propose a `syncvm` operation

Table 1: ***PLOVER's consensus operations, all built on top of APUS's*** `paxos_op(void *val)` ***API (§2.3) .***

4 The PLOVER Runtime System

This section introduces PLOVER's runtime System. Table 1 shows all the three types of consensus operation APIs that PLOVER's leader invokes.

4.1 Terminology Setup

A PLOVER replica maintains a <`role`,`vid`,`log`, n_{err} > tuple on its local QEMU hypervisor. `role` is a replica's role (leader, secondary, or witness) that has been agreed by PAXOS; `vid` is the current PAXOS view ID [68]; `log` is the current PAXOS consensus log (§3.2), and n_{err} is the current number of communication failures recorded in PAXOS (e.g., a PAXOS heartbeat failure will increment n_{err} by 1). `vid`, `log`, and n_{err} are all exposed from the underlying PAXOS implementation, and PLOVER only updates n_{err} if a `syncvm` has an error. In short, PLOVER runs on top of PAXOS without modifying its implementation.

4.2 SMR Operation Types

As an SMR system, all PLOVER operations run on top of the underlying PAXOS protocol. PLOVER has three SMR operations in total: `paxos_second`, `paxos_input`, and `paxos_syncvm` (Table 1).

The `paxos_second` API is invoked by the PLOVER leader to assign one replica as the secondary, ensuring that the new secondary is consistently agreed among PLOVER replicas. This API is invoked when a new PLOVER leader is elected or the secondary is suspected to fail. PLOVER's leader randomly proposes a replica in its current PAXOS group as the secondary. This operation complies with PAXOS safety guarantee even if the proposed secondary fails, because the leader's `syncvm` operations can detect the new secondary's failure by incrementing n_{err} (§4.4).

The `paxos_input` operations are invoked by the leader when inbound network packets arrive at local hypervisor. Both PLOVER secondary and witness act as "PAXOS witnesses" to agree on the proposed packet, achieving a standard PAXOS consensus.

The `paxos_syncvm` is used to invoke a `syncvm` operation in PLOVER. When the primary finds that the service running in local VM has finished processing inputs and become idle (§4.3), it invokes a consensus on `syncvm` by invoking a `paxos_syncvm` operation. Invoking a `syncvm` with consensus is beneficial: it makes the leader and secondary receive exactly the same sequence of client requests between every two consecutive `syncvm`

operations, greatly reducing memory divergence (§6.3).

4.3 Efficiently Determining Slot Boundary

Similar to primary-backup for ensuring external consistency [86], PLOVER leader must buffer all outbound packets before a `syncvm` succeeds, including client responses and TCP ACKs. Client programs will stop sending new packets when their TCP congestion windows are met, even server programs have finished processing requests and become idle. This leads to unnecessary time slots. In practical workloads with concurrent connections, arrival times of requests are often unpredictable, thus a static `syncvm` time slot configuration (e.g., 25ms in REMUS and 100ms in MC) can often cause an idle service and unnecessary time slots.

To avoid unnecessary time slots, PLOVER develops an adaptive-slot algorithm by inserting `syncvm` operations when its leader determines idle status of programs running in guest VM. PLOVER leverages QEMU's threading hierarchy to spawn an internal thread that checks the CPU usage of the guest VM to determine whether it is idle. §5.1 describes implementation details. This simple, non-intrusive algorithm helps PLOVER quickly proceed its slots, and our evaluation shows that this algorithm is effective in improving PLOVER's performance (§6.2).

4.4 Protocol for `syncvm`

PLOVER's `syncvm` contains three phases (§3.3). The first phase is `compute`. On executing the `paxos_syncvm` operation, the leader pauses its VM immediately, while the secondary does the pause when its programs become idle (§4.3). When both VMs are paused, leader and secondary exchange their dirty page bitmap and compute a union of the two bitmaps. Then, leader and secondary concurrently compute page hashes according to the union.

The second phase is `compare`. The secondary sends its hash list to the leader, and the leader does a comparison to identify all divergent pages.

The third phase is `transfer`. The leader uses RDMA to transfer all divergent pages to secondary and append a special EOF at the end. The secondary saves the pages in a static buffer, sends an ACK to the leader, and applies divergent pages to its guest VM in an atomic manner. This is crucial for PLOVER's correctness because if the secondary starts applying pages while receiving, and the leader fails in the middle, the secondary will end in a corrupted state. On receiving the ACK, the leader releases outputs since the last `syncvm` and resumes its guest VM.

All the three phases carry the sender's `vid` and the receiver checks `vid` as a standard PAXOS way [35, 68]. If any communication error happens during a `syncvm`, a local replica increments n_{err} by 1. If this replica is the leader, it re-invokes a `syncvm` consensus (§4.2). PAXOS will be involved once n_{err} reaches its re-election threshold. Although updating n_{err} in both PLOVER and in the underlying PAXOS may have data races, n_{err} is just a statistic variable and there is no a correctness issue.

4.5 Handling Replica Failures

PLOVER automatically tolerates one replica failure. If the secondary fails, the leader will invoke a standard VM migration to bring the witness's states up-to-date and elevate the witness to be the secondary. If the witness fails, no PLOVER actions are needed because the leader can continue to serve client requests and ensure fault tolerance.

If the leader is suspected to fail, a new leader will be elected. Because the secondary's state is more up-to-date, PLOVER ensures if a secondary is working normally, it will always be the new leader. To do so, PLOVER doesn't let the witness become the leader in the leader election. After the secondary becomes the leader, it will do a VM migration from itself to the witness, elevate the witness to be the secondary, and start to serve client requests.

4.6 Correctness

PLOVER is designed to handle the same failure model as regular SMR, where network messages may be lost but will not be corrupted, network may be partitioned, and hosts may fail. As an SMR system with three replicas, PLOVER can tolerate the failure of one replica.

PLOVER guarantees external consistency: if a client receives a reply for its requests, the execution states generating this reply will not be lost. Prior work [36, 38] shows that this guarantee is sufficient for VM fault-tolerance in a client-server model.

We give a proof sketch of PLOVER's external consistency guarantee in three steps. First, all replies are sent from PLOVER's leader. PLOVER's underlying PAXOS protocol ensures one strongly consistent leader among the replicas. Moreover, only the leader invokes `syncvm` operations and network outputs will not be released until a `syncvm` finishes.

Second, PLOVER does not affect the correctness of its underlying PAXOS. We made only two modifications to the underlying PAXOS protocol: always elevating the secondary to be the new leader and increasing n_{err} on a `syncvm` error (§4.1). These two modifications do not hurt PAXOS's correctness because it guarantees there is one unique leader in each view, and n_{err} is just a counter of observed communication errors on a local host.

Third, before sending out a reply, the leader has finished a `syncvm` and successfully replicated the states that generate this reply to the secondary. No matter which replica fails, the states will not be lost. Therefore, PLOVER ensures external consistency.

We also carefully designed PLOVER for reasonable

liveness. If the leader is alive, its `syncvm` operation has a timeout-and-retry mechanism and its program-idle determination (§4.3) has a bounded waiting time.

5 Implementation Details

Much of PLOVER implementation code was inherited from well-engineered VM systems [13, 38, 52], including replicating file system [38]. Our implementation found and fixed two new bugs that crashed MC [13]: one bug was an integer overflow on the number of dirty pages, the other was an inconsistent states between the PCI device and bus on restarting replicas. QEMU developers confirmed both our bug reports.

5.1 Determining Server Program Idle Status

When clients connect with services running in a VM fault-tolerance system (e.g., REMUS and PLOVER) using TCP, the system buffers network outputs and causes the clients' TCP windows to become full and to stop sending requests. This will result in an idle service and a wasted time slot. Therefore, a mechanism is needed to determine when the service is idle, so that a `syncvm` is invoked.

To efficiently find the idle status of a service, PLOVER creates a simple, non-intrusive algorithm without modifying guest OS. This algorithm uses the threading hierarchy of QEMU: all QEMU virtual threads (threads that emulate vCPUs) are spawned from the QEMU hypervisor process (§2.2). PLOVER creates an internal thread in the process to call `clock()`, which gets the total CPU clock of a process and its children. If PLOVER finds that the increment rate of this clock is as small as an vacant VM for a threshold (100μs), it finds the service idle.

This eliminates wasted time slots in PLOVER and lets services run almost in full speed. Moreover, because both the PLOVER leader and secondary finish processing current requests, their memory should be mostly the same. This simple algorithm is already effective for reducing page divergence (§6.3) and achieved reasonable performance overhead (§6.2) in our evaluation, and it can be further extended to handle straggler requests.

5.2 Computing Dirty Page Hashes Concurrently

We leveraged multi-core hardware and implemented a multi-threaded dirty page hash computing mechanism. The mechanism detects the number of CPU cores on local host creates same number of threads to compute hashes of dirty physical pages since the last PLOVER `syncvm` operation. We used Google's CityHash [43], because it is fast. Our evaluation shows that computing hashes has reasonable CPU footprint (§6.4) because it takes only about 6.3μs for each page.

5.3 Fast Consensus in Hypervisor

PLOVER uses APUS [92] to achieve consensus on network inputs among replicas. A naive approach for implementing this is to let APUS intercept network packets and synchronously achieve consensus in QEMU's inbound network device (e.g., TAP device). However, this approach causes severe performance degradation. QEMU's network is implemented in an event driven model. On receiving a network packet, the event handler needs to acquire a global lock and feed the packet into the VM. The whole process takes less than 1μs. On the other hand, APUS takes over 10μs to reach consensus. As a result, this naive approach would hold the global lock for a long period and block the handling of other events, causing great performance degradation to the VM.

To address this problem, we implemented a non-blocking consensus mechanism in QEMU. On receiving a network packet, rather than directly feed it into the guest VM, the event handler only appends the packet to a buffer. PLOVER asynchronously reads packets from the buffer, invokes APUS to achieve consensus, and leverages QEMU's event driven loop to feed the packet into the VM.

6 Evaluation

Our evaluation hosts were nine Dell R430 servers with Linux 3.16.0, 2.6 GHz Intel Xeon CPU with 24 hyper-threading cores, 64GB memory, and 1TB SSD. All NICs are Mellanox ConnectX-3 Pro 40Gbps connected with Infiniband [2]. To mitigate LAN/WAN network variance, all client benchmarks and VMs were run in these hosts. Running clients in WAN will further mask PLOVER overhead compared to unreplicated executions.

We evaluated PLOVER on 12 widely used programs, including 8 server programs (`Redis` [81], `SSDB` [85], `MediaTomb` [21], `Nginx` [73], `MySQL` [22], `Tomcat` [3], `PgSql` [79], and `lighttpd` [59]) and 4 dynamic language interpreters (`Node.js`, PHP, Python, and JSP). To be close to real-world deployments, we group these programs into 8 practical services, including DjCMS [7], a large, sophisticated content management system (CMS) consisting of `Nginx`, Python, and `MySQL`.

We used popular workloads that make these services run at their peak throughputs and then collected results. Prior study [76, 88] shows that hardware errors occur more frequently when services have higher load, thus the fault-tolerance of PLOVER is more crucial. Table 2 shows our workloads. For each workload, we spawned different number of clients to saturate the services and collected the curve of throughputs for unreplicated executions.

For `Redis` and `SSDB`, each request contains a batch of 1K operations of 50% SET and 50% GET; for the other six services, each request contains one operation. We found sending operations in batches for `Redis` and `SSDB` made them reach peak throughput. For instance, when each request for `Redis` contains only one SET or GET operation, `Redis`'s throughput is only 43K oper-

ation/s for 64 connections; when each request is a 1K-operation batch, its throughput reaches a peak value of 481K operation/s for 64 connections.

Service	Workload
`Redis`	50% SET, 50% GET requests arriving in batches.
`SSDB`	50% SET, 50% GET requests arriving in batches.
`MediaT`	Concurrent requests on transcoding a 50MB video.
DjCMS	Concurrent requests on a dashboard page [8].
`PgSql`	PgBench [79] with TPC-B benchmark.
`Tomcat`	Concurrent requests on a shopping store page [15].
`lighttpd`	Concurrent requests using PHP to watermark images [1].
`Node.js`	Concurrent requests on a messenger bot [12].

Table 2: ***Eight services and workloads used in experiments.***

We compared PLOVER with four fault-tolerance systems: CRANE [35], an open-source SMR system among recent ones [35, 45, 50]; QEMU-MicroCheckpoint [13] (for short, MC), a REMUS-based primary-backup system carried in QEMU [80]; Synchronization Traffic Reduction (STR) [63], a primary-backup system designed to reduce the number of transfered pages; and COLO [38], a primary-backup system deployed in Huawei [20]. MC has an RDMA implementation [14], but it is being actively developed and not runnable on our hosts. We did not use REMUS because it was built before 2008 and did not run on our hosts. This section focuses on six questions:

§6.1: Can PLOVER correctly enforce deterministic executions by transferring only divergent pages?

§6.2: How fast is PLOVER compared to MC, STR, and COLO? How does it scale to multi-core?

§6.3: How effective is each PLOVER technique on reducing divergence of dirty pages?

§6.4: What is PLOVER's CPU footprint and how well does it support VM consolidation?

§6.5: Can PLOVER efficiently handle replica failures?

§6.6: What did we learn from VSMR and its implementation PLOVER? What are PLOVER's limitations?

6.1 Verifying Correctness

To check whether PLOVER can capture all divergent memory pages, we took Racey [48], a nondeterminism stress testing benchmark. Racey generates many data race accesses by using multiple threads to access an in-memory array concurrently without acquiring any locks, and it computes an output based on the array content.

We wrote a shell script to repetitively launch the Racey program in PLOVER leader VM for 3K times and appended its output to a file in local VM. We compared the files between PLOVER's leader and secondary and found the files had the same content. Thus, PLOVER indeed captured and transfered all divergent pages.

VMware's documentation [9, 17] states that vSphereFT-6.5 works similar to MC. Since vSphereFT is not open source and has restrictions on publishing evaluation results [16], we compared PLOVER with MC instead.

6.2 Performance and Scalability on Multi-core

Figure 3 shows PLOVER, MC, STR, and COLO's throughput on 8 services with different number of clients. MC used 100ms-slot (MC's default) and STR used 25ms-slot (STR's own default). All experiments ran on 4-vCPU per VM (unless specified) because COLO [38] and REMUS [36] evaluated up to 4 vCPUs per VM. On average, PLOVER's throughput is 2.2X higher than MC, STR, and COLO.

As the number of clients increases, PLOVER's throughput overhead becomes less obvious. The overhead mainly comes from the `syncvm` operations, which is determined by the `syncvm` frequency and the time spent on each `syncvm`. When the load on the service increases, the VM takes more time to be idle, so the `syncvm` frequency becomes smaller. On the other hand, the time spent on each `syncvm` remains almost the same because PLOVER only transfers divergent pages. Therefore the `syncvm` overhead becomes smaller when the number of client increases.

`PgSql` is the only service for which PLOVER is slower than COLO. COLO compares per-connection outputs between its primary and backup and skips `syncvm` if outputs did not diverge. `PgSql` ran SQL transaction workloads and its outputs were mostly the same. Except for `PgSql`, PLOVER was several times faster than COLO.

To analyze COLO, we also looked into `SSDB`, which had concurrent SET/GET requests. We found that COLO's output divergence was frequent when data dependencies exist among connections (i.e., GET requests frequently got different responses when SET and GET requests on the same key arrived at `SSDB` concurrently). When any output in any connection had an output divergence, COLO did a `syncvm`. COLO evaluation shows that it greatly slowed down when the number of client connections was large. PLOVER is not sensitive to outputs.

Intuitively, STR should perform better than MC because it only transfers divergent dirty pages in `syncvm`. However, our evaluation found that sometimes STR's throughput is lower than MC (e.g., 64 clients in `Redis`). This comes from two aspects. First, STR uses a static `syncvm` interval and this causes many divergent dirty pages to be transferred. Second, compared to MC, STR requires extra time to compute and compare dirty page hashes.

All eight services' unreplicated executions reach their peak throughput on 64 clients except for `PgSql`; `PgSql` reaches its peak throughput on 32 clients. For the remaining of the paper, we use the peak throughput points of unreplicated executions of each service as our sample points.

Figure 4 shows the response time of the four systems

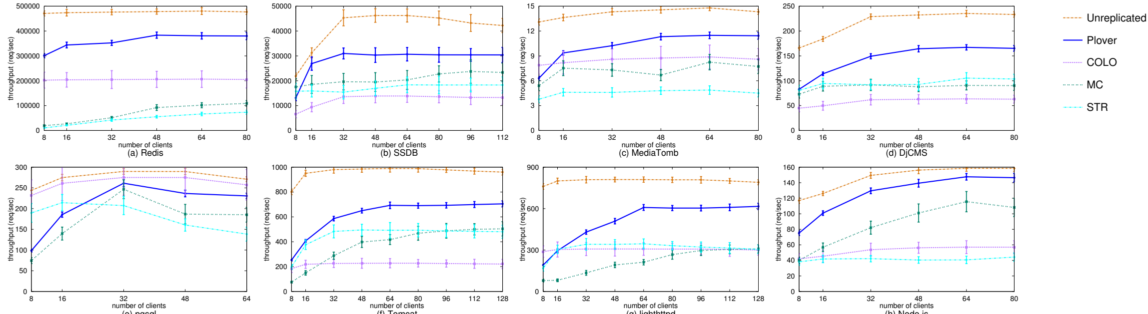


Figure 3: ***Throughput comparison (4 vCPUs per VM).*** **The error bars represent 95% confidence intervals about the mean. For the remaining figures and tables, we use the peak throughput points of unreplicated executions as our sample points.**

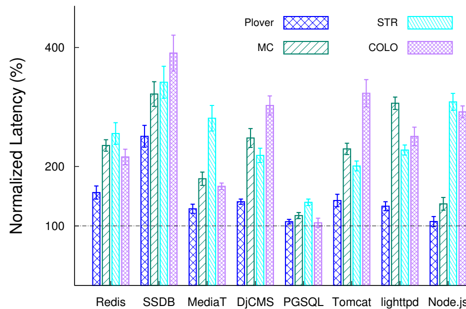


Figure 4: ***Response times normalized to unreplicated executions (4 vCPUs per VM).*** **100% means no overhead.**

normalized to unreplicated executions. For six services (excluding `Redis` and `SSDB`), PLOVER's overhead of response time follows the same trend as the overhead of throughput because each client connection in these six services sends requests one by one. For `Redis` and `SSDB`, because the requests arrive in batches in order to saturate the two services, all four systems incur high overhead on response time. Specifically, PLOVER incurred the highest latency overhead for `SSDB`, because its same dirty page rate was only 77% (Table 3).

Figure 5 explains why PLOVER's performance was higher. PLOVER consumes 9.2X less bandwidth than MC, STR, and COLO on average. This reduction makes PLOVER the first VM fault-tolerance system that supports consolidating multiple VMs on a host (§6.4).

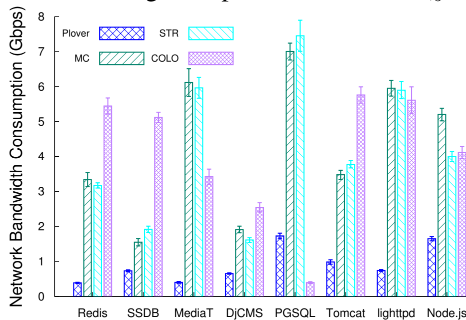


Figure 5: PLOVER ***network bandwidth consumption compared with STR, MC and*** **COLO** ***(four vCPUs per VM).***

To understand PLOVER's performance, Table 3 shows its micro events. For all evaluated services, we observed that 72%∼97% pages between leader and secondary were the same. This greatly reduced page transferring time, a major performance bottleneck in primary-backup systems such as MC. The time between every two `syncvm` operations largely varied, which reflects that PLOVER can automatically detect service idle time (§5.1) for diverse workloads.

Table 4 shows that MC-25ms and MC-100ms have similar performance: a larger `syncvm` time slot accumulates more dirty pages, and thus much longer copy time and transfer time (§3.3). Combining Table 3 and Table 4 explains why PLOVER was much faster than MC: PLOVER only needs to transfer 3% ∼ 28% of dirty pages.

The results of STR-25ms and STR-100ms are similar. In our experiments, the throughput difference between these two systems was 18% on average. This is because STR uses a static `syncvm` interval, in which primary and secondary process different number of client requests. As a result, both STR-25ms and STR-100ms have low same dirty page rate and need to transfer most of the dirty pages. Therefore, we only focus on evaluating MC-100ms and STR-25ms (their own default settings) in the following sections.

To evaluate the effectiveness of RDMA in PLOVER's implementation, we changed PLOVER's dirty page bitmap and divergent page transfer mechanisms from RDMA to TCP (PLOVER-TCP). We ran the 8 services with PLOVER-TCP and found that, compared to PLOVER, PLOVER-TCP's overall throughput dropped by 2.1% ∼ 9.8%. We found that PLOVER-TCP increased the time spent in the two transfer mechanisms by 35.1% ∼ 74.2%. Because neither of the two mechanisms is PLOVER's performance bottleneck, PLOVER's high performance mainly stems from greatly reducing the pages that need to be transferred rather than RDMA.

We also evaluated PLOVER scalability on up to 16 vCPUs per VM. Figure 6 shows the scalability results on four services, normalized to PLOVER throughput on four vCPUs. The throughputs of the other four services were not scalable to multi-core (e.g., `PgSql` is I/O bound and its throughput increased by only 14.7% when we changed the number of vCPUs per VM from 4 to 16),

Service	Compute	Compare	Trans	Interval	Page	Same
Redis	3.5ms	1.9ms	2.8ms	153ms	13.5k	93%
SSDB	2.3ms	1.4ms	6.2ms	180ms	9.1k	77%
MediaT	7.9ms	4.0ms	17.1ms	914ms	29.2k	86%
DjCMS	0.9ms	1.3ms	3.3ms	90ms	3.6k	74%
PgSql	2.8ms	1.5ms	8.3ms	93ms	11.1k	76%
Tomcat	1.1ms	0.6ms	3.6ms	78ms	4.3k	72%
lighttpd	9.4ms	5.0ms	2.8ms	86ms	33.9k	97%
Node.js	9.6ms	5.5ms	28.8ms	375ms	37.8k	74%

Table 3: PLOVER *performance analysis for each* `syncvm` *operation (on average)*. "Compute" means the time of computing hashes for dirty pages; "Compare" means the time of comparing hashes between leader and secondary; "Trans" means the time of transferring divergent pages; "Interval" means the time between two `syncvm` detected by PLOVER (§4.3); "Page" means the number of dirty pages in each `syncvm`; "Same" means the same rate of dirty pages.

	MC-25ms			MC-100ms		
Program	Page	Copy	Transfer	Page	Copy	Transfer
Redis	6.1k	6.6ms	30.2ms	11.0k	11.9ms	35.1ms
SSDB	2.7k	2.9ms	7.8ms	4.8k	5.2ms	20.0ms
mediaT	4.6k	5.1ms	20.5ms	3.8k	4.2ms	16.5ms
DjCMS	2.8k	3.1ms	9.0ms	3.8k	4.1ms	13.2ms
PgSql	7.9k	8.5ms	39.0ms	8.2k	8.9ms	40.9ms
Tomcat	6.5k	6.5ms	15.6ms	12.2k	13.2ms	39.8ms
lighttpd	33.3k	23.9ms	53.5ms	33.9k	11.6ms	55.7ms
Node.js	11.3k	11.6ms	36.7ms	21.3k	14.9ms	42.5ms

Table 4: *MC performance analysis for each* `syncvm` *operation (on average) with 25ms and 100ms time slot*. "Page" means the number of dirty pages in each `syncvm`; "Copy" means the time for copying dirty pages (§3.3); "Transfer" means the time for transferring ditry pages.

so the four services do not need the 16-vCPU speedup. Overall, PLOVER scaled well for all four services, and its throughput was 3.8X higher than MC, STR, and COLO on 16-vCPU VMs. When the number of virtual CPUs increased from 1 to 16, the throughput for COLO, STR, and MC reached a bottleneck at 4 cores and even dropped for SSDB and MediaTomb. Prior study [36, 40, 42] points out a main reason of this huge drop: the number of dirty pages a primary-backup approach has to transfer will increase greatly when more vCPUs are added into one VM.

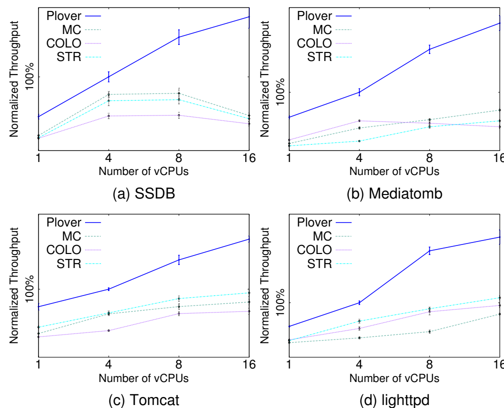


Figure 6: *Throughput scalability on the number of vCPUs per VM, normalized to* PLOVER*'s 4-vCPU throughput.* The more vCPUs per VM, the faster PLOVER than STR, MC and COLO.

6.3 Effectiveness of PLOVER Reduction Techniques

PLOVER's high performance is mainly brought by two techniques: same total order of inputs and efficiently determining service idle time (§4.3). To assess their effectiveness, we used three plans.

First (Plan1), we implemented a per-TCP-connection input forwarding mechanism between leader and secondary to order each TCP connection separately and used a 25ms `syncvm` time slot. Second (Plan2), we enforced a total order of inputs for all connections between leader and secondary, and used a 25ms `syncvm` time slot. Third (Plan3), we ran the full PLOVER.

For all three plans, we measured Same Dirty Page Rate (SDPR): the percentage of same dirty physical pages between two replicas. The difference between Plan1 and Plan2 shows the effectiveness of total ordering of network inputs between leader and secondary. The difference between Plan2 and Plan3 (PLOVER) shows the effectiveness of determining service idle time. When PLOVER is configured with a static `syncvm` interval (25ms), it has an average of 5.1% higher SDPR than Plan2. This shows that using PAXOS instead of STR to order network inputs incurs only a small cost.

Figure 7 shows that, the SDPR for 8 services differed by 25.5% on average between Plan1 and Plan2, and the difference between Plan2 and Plan3 was 29.2% on average. Both PLOVER's two techniques were quite effective on improving SDPR and the performance of services.

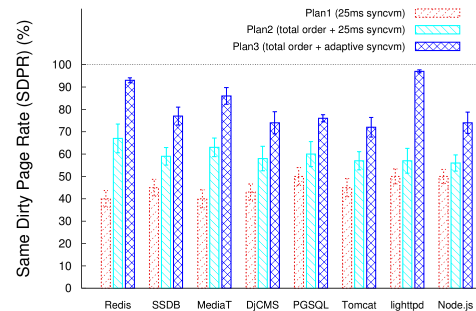


Figure 7: *Effectiveness of* PLOVER *techniques on reducing divergent pages.*

6.4 CPU Footprint and Consolidation

Figure 8 shows PLOVER's CPU footprint on the 8 services compared with unreplicated execution in KVM, MC and COLO. STR's CPU footprint is not included in the figure because it is similar to PLOVER's. Both PLOVER and COLO let their leaders and secondaries execute clients' requests concurrently. MC's secondary does not execute clients' requests but is busy applying updated states. Different from COLO and MC, PLOVER has a witness which consumed 7% $\sim$ 15% CPU to agree on network inputs without executing them.

Except for Redis, PLOVER's leader and secondary incurred 2.7% $\sim$ 9.2% and 5.3% $\sim$ 18.3% more CPU than

unreplicated executions, including computing hashes, comparing hashes and transferring divergent pages. PLOVER's CPU footprint was not significant for two reasons. First, computing hash for each page only took 6.3μs. Second, by transferring only divergent pages, PLOVER saved much CPU on transferring pages. For `Redis`, all three systems incurred obvious CPU footprint because `Redis` is single threaded, so its unreplicated execution only used 1 out of the 4 vCPUs.

We also evaluated PLOVER's performance on VM consolidation. We deployed one to five PLOVER leader VMs (each with 4 vCPUs) on a 24-core host, ran `PgSql` in each VM, and spawned the same number of clients for each VM. We found the total throughputs of all VMs in the host increased from 230 (one VM) to 1089 requests/s (five VMs) and the network bandwidth consumption increased from 1.8 Gbps to 10.1 Gbps. These results suggest that PLOVER is friendly to consolidating multiple fault-tolerant VMs on the same host due to its greatly reduced network bandwidth consumption compared to MC and COLO. Neither MC [13] nor COLO [38] evaluated consolidation. vSphereFT-6.5 [5] currently supports up to two 4-vCPU VMs on each physical host.

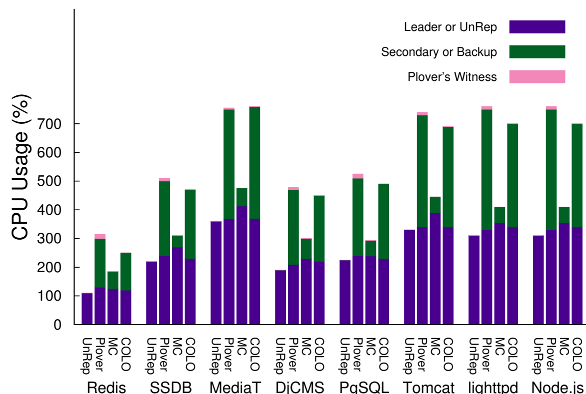


Figure 8: ***CPU footprint on 4-vCPU VMs*****. "UnRep" means unreplicated execution. PLOVER has 3 replicas; MC and COLO have 2. PLOVER and COLO have similar footprint.**

6.5 Handling Hardware Failures

We measured the performance of PLOVER when various failures happened. We killed the leader, secondary, and witness in each experiment and monitored the real-time throughput of `Redis`. When the witness was killed, we did not observe performance impacts for `Redis`.

Figure 9 shows `Redis`'s throughput fluctuation when we killed the leader at the 3rd second and then added a new replica after a few seconds. The APUS leader election protocol [92] employed by PLOVER took a 100ms timeout to detect the leader's failure and 16.3μs to elect the secondary as the new leader. Then the new leader did a full VM migration to make the witness's guest VM up-to-date, which took about 2.8s. We also partitioned the leader out and then added it back after a second, and we found the new leader was elected almost as quickly as the leader's failure case without having a split-brain issue. Unlike existing SMR systems [35, 74] which need complex mechanisms to find the new leader's IP address, clients were not perturbed during a PLOVER leader election because VSMR replicates an entire guest VM (including its IP address) as a state machine.

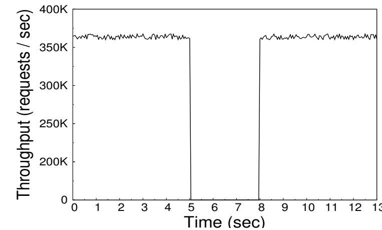


Figure 9: ***`Redis`'s performance on handling PLOVER's leader failure and adding a new replica*****.**

6.6 Lessons Learned

We found VM a promising abstraction to enforce same executions among SMR replicas. This owes to three main reasons. First, the VM abstraction can efficiently and systematically capture state changes in the guest OS, including both userspace and kernel memory. We also found VM useful on synchronizing systems nondeterminism (e.g., enforcing same physical times and ASLR layouts) across our replicas. Second, a VM carries a rich set of management primitives (e.g., migration), which makes SMR recovery easy to implement.

Third, a VM itself has several transparency features that SMR needs. For instance, the same VM replicated on different physical hosts can maintain the same IP and MAC addresses, making client connections transparently switch to the new PLOVER leader if current leader fails. In contrast, traditional SMR implementations (e.g., Raft [74]) require complex mechanisms to find the new leader. In this regard, VSMR makes SMR simpler.

PLOVER has two limitations. First, it requires the leader VM and the secondary VM to occupy the same amount of computation resources, so that they can finish processing current requests roughly at the same time and do `syncvm` efficiently. We deemed this requirement reasonable due to three reasons: (1) it is much easier to achieve in VM deployments than on bare-metal, because VMs have performance isolation and they will not overuse resources; (2) PLOVER has greatly reduced network bandwidth consumption, a major resource that may cause performance contention among VMs on the same host; and (3) requiring primary and backup to run on roughly the same amount of computation resources is a common requirement in primary-backup systems [17].

Second, as an SMR approach, VSMR requires three replicas and thus it consumes more CPU than primary-backup systems. Our evaluation shows that PLOVER's CPU consumption is compatible to COLO, about twice as

big as unreplicated executions. Nevertheless, PLOVER's robust fault-tolerance, modest performance overhead, and low network bandwidth consumption could make its extra CPU usage worthwhile.

7 Related Work

VM-based Fault-tolerance. Existing VM-based fault-tolerance systems [13, 36, 42, 62–64, 82] typically take the primary-backup approach: they propagate incremental updates from the primary VM to the backup VM. Primary-backup is more scalable than the log-replay [83] approach on multi-core because the latter needs to record exact interleavings of shared memory access. However, all typical VM fault-tolerance systems (REMUS, COLO, MC and vSphereFT) evaluated up to 4 vCPUs per VM. As most programs scale to more and more cores and access increasingly larger memory working set, transferring these dirty pages becomes a notorious problem [36, 38, 42] for the primary-backup approach, greatly degrading program performance and hijacking excessive network bandwidth.

Four recent papers aim to alleviate the open problem. First, COLO [38] lets primary and backup compare per-TCP-connection outputs and avoid dirty page propagation if no outputs diverge. COLO has effectively scaled the Remus-based approach to up to four vCPUs per VM. As shown in both COLO's and our evaluation, when the number of client connections is large or when data dependency among connections exists, COLO does many more `syncvm` operations than REMUS. PLOVER is not sensitive to output divergence.

Second, Gerofi et al. [42] shows that using copy-on-write during the dirty memory copying (t_{copy} in §3.3), primary-backup can resume VMs faster than REMUS; this work also shows that using a 10Gbps RDMA NIC can transfer dirty page faster than using a 1Gbps Ethernet NIC. Another latest work [82] also shows that RDMA can mitigate t_{copy}. These two works [42, 82] are complementary to PLOVER because PLOVER focuses on greatly reducing the amount of transferred dirty pages.

Third, Adaptive-Remus [37] shows that REMUS can monitor its output buffer and do a `syncvm` once noticing outputs. This work improves REMUS's performance by 29% when the number of client connections was small. However, with many connections, it will invoke a `syncvm` for almost every network output and incur prohibitive performance overhead.

Fourth, Tardigrade [62] uses lightweight VM (LVMs) to decrease the memory footprint on the primary to reduce checkpoint costs. On the other hand, PLOVER focuses on transferring only the divergent pages between primary and secondary to alleviate the checkpoint overhead. Besides, Tardigrade typically runs a single-process application, while PLOVER runs multiple processes (programs) in a guest VM.

State Machine Replication (SMR). Fault-tolerance is an essential technique in distributed systems [27, 29, 68]. SMR [68] is a powerful fault-tolerance technique: it typically uses PAXOS [55, 56, 68, 71, 89]) to enforce a total order of inputs for the replicated service, tolerating various failures. Many PAXOS implementation protocols [30, 31, 35, 68] exist. Consensus is widely used in datacenters [19, 49, 94] and worldwide Internet [33, 65]. Much work is done to improve specific aspects, including commutativity [66, 71], understandability [56, 74], and verification [44, 93].

To make SMR work with modern parallel programs, extra mechanisms are needed to ensure same program executions across replicas. Existing SMR systems propose a few fast mechanisms, including annotating global variables in program code [50] and enforcing same order of inter-thread synchronization [35, 45]. These mechanisms have shown reasonable performance on real-world programs, but they may require developer intervention (e.g., incorrect annotation or data races). Moreover, these mechanisms only enforce best-effort determinisms on userspace, not in kernel. PLOVER implements the new VSMR approach to realize an automatic, faster, and more scalable SMR system.

Multi-core Replay. Deterministic replay [25, 39–41, 46, 53, 54, 70, 75, 84, 90] aims to replay the exact recorded executions. Scribe tracks page ownership to enforce deterministic memory access [54]. Respec [57] uses online replay to keep multiple replicas of a multithreaded program in sync. In these record-replay systems, a false-sharing problem exists: recording becomes expensive even if multiple threads access different portions of same page. As most false-shared pages should have same contents, PLOVER may mitigate this problem.

8 Conclusion

We have presented VSMR, a novel SMR approach that makes VM fault-tolerance much faster and more scalable on multi-core. We have described PLOVER, the first VSMR system implementation and its evaluation on a wide range of real-world server programs and services. PLOVER runs several times faster than three popular primary-backup systems and it saves much bandwidth. PLOVER has the potential to greatly improve the reliability of real-world online services, and it can be applied to other research areas (e.g., multi-core replay).

Acknowledgments

We thank Jay Lorch (our shepherd) and anonymous reviewers for their many helpful comments. This paper is funded in part by a research grant from the Huawei Innovation Research Program (HIRP) 2017, HK RGC ECS (No. 27200916), HK RGC GRF (No. 17207117), and a Croucher innovation award.

References

[1] Adding watermarks to images using alpha channels. http://php.net/manual/en/image.examples-watermark.php.

[2] An Introduction to the InfiniBand Architecture. http://buyya.com/superstorage/chap42.pdf.

[3] Apache tomcat. http://tomcat.apache.org/.

[4] Comparison of 40G RDMA and Traditional Ethernet Technologies. https://www.nas.nasa.gov/assets/pdf/papers/40_Gig_Whitepaper_11-2013.pdf.

[5] Configuration Maximums (vSphere 6.5). https://www.vmware.com/pdf/vsphere6/r65/vsphere-65-configuration-maximums.pdf.

[6] Data Plane Development Kit (DPDK). http://dpdk.org/.

[7] django cms - enterprise content management with django. https://www.django-cms.org/en/.

[8] Django fluent dashboard. https://github.com/django-fluent/django-fluent-dashboard.

[9] Fault Tolerance Performance in vSphere 6. https://blogs.vmware.com/performance/2016/01/vsphere6-fault-tolerance-perf.html.

[10] Implementing TCP Sockets over RDMA. https://www.openfabrics.org/images/eventpresos/workshops2014/IBUG/presos/Thursday/PDF/09_Sockets-over-rdma.pdf.

[11] Mellanox Products: RDMA over Converged Ethernet (RoCE). http://www.mellanox.com/page/products_dyn?product_family=79.

[12] Pokdex messenger bot for pokmon go. https://github.com/zwacky/pokedex-go.

[13] QEMU MicroCheckpoint. https://wiki.qemu.org/Features/MicroCheckpointing.

[14] RDMA migration and rdma fault tolerance for QEMU. http://www.linux-kvm.org/images/0/09/Kvm-forum-2013-rdma.pdf.

[15] Simple shopping store. https://github.com/SaiUpadhyayula/SimpleShoppingStore.

[16] VMware End User License Agreements. http://www.vmware.com/download/eula.html.

[17] VMware vSphere 6 Fault Tolerance: Architecture and Performance. http://www.vmware.com/files/pdf/techpaper/VMware-vSphere6-FT-arch-perf.pdf.

[18] Which Hardware Fails the Most and Why. http://www.storagecraft.com/blog/hardware-failure/.

[19] Why the data center needs an operating system. https://cs.stanford.edu/~matei/papers/2011/hotcloud_datacenter_os.pdf.

[20] Huawei FusionSphere. https://www.youtube.com/watch?v=yvsVuLAOhCo, 2014.

[21] MediaTomb - Free UPnP MediaServer. http://mediatomb.cc/, 2014.

[22] MySQL Database. http://www.mysql.com/, 2014.

[23] Intermediate Course In Operating System: High Availability. www.cs.cornell.edu/ken/book/New%20514%20slide%20set/12-HighAvailability.ppt, 2015.

[24] The OTHER way of recovering from VMware ESXi Split Brain. https://www.pei.com/2017/02/way-recovering-vmware-esxi-split-brain/, 2017.

[25] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, Oct. 2009.

[26] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[27] K. P. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, 1985.

[28] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, Sedms'93, 1993.

[29] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Dec. 1995.

[30] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[31] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, Aug. 2007.

[32] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, 2005.

[33] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Oct. 2012.

[34] A. Corradi, M. Fanelli, and L. Foschini. Vm consolidation: A real case based on openstack cloud. *Future Gener. Comput. Syst.*, Mar. 2014.

[35] H. Cui, R. Gu, C. Liu, and J. Yang. Paxos made transparent. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[36] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[37] M. P. da Silva, R. R. Obelheiro, and G. P. Koslovski. Adaptive remus: adaptive checkpointing for xen-based virtual machine replication. *International Journal of Parallel, Emergent and Distributed Systems*, 32(4):348–367, 2017.

[38] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. Colo: Coarse-grained lock-stepping virtual machines for non-stop service. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[39] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 211–224, Dec. 2002.

[40] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the 4th International Conference on Virtual Execution Environments (VEE '08)*, pages 121–130, Mar. 2008.

[41] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, Apr. 2007.

[42] B. Gerofi and Y. Ishikawa. Rdma based replication of multiprocessor virtual machines over high-performance interconnects. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, 2011.

[43] `https://github.com/google/cityhash`.

[44] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 265–278, Oct. 2011.

[45] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 2014 ACM European Conference on Computer Systems (EUROSYS '14)*, page 11. ACM, 2014.

[46] 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 *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 193–208, Dec. 2008.

[47] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, 2006.

[48] M. D. Hill and M. Xu. Racey: A stress test for deterministic execution. `http://www.cs.wisc.edu/~markhill/racey.html`, 2009.

[49] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[50] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the Tenth Symposium on Operating Systems Design and Implementation (OSDI '12)*, volume 12, pages 237–250, 2012.

[51] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Oct. 2015.

[52] `http://www.linux-kvm.org/`.

[53] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pages 219–228, May 2000.

[54] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '10)*, pages 155–166, June 2010.

[55] L. Lamport. Paxos made simple. `http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf`.

[56] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[57] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Fifteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 77–90, Mar. 2010.

[58] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Fast replication with nopaxos: Replacing consensus with network ordering. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, Nov. 2016.

[59] `https://www.lighttpd.net/`.

[60] T. Liu, C. Curtsinger, and E. D. Berger. DTHREADS: efficient deterministic multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 327–336, Oct. 2011.

[61] T. Liu, C. Tian, Z. Hu, and E. D. Berger. Predator: Predictive false sharing detection. *SIGPLAN Not.*, 49(8), Feb. 2014.

[62] J. R. Lorch, A. Baumann, L. Glendenning, D. T. Meyer, and A. Warfield. Tardigrade: Leveraging lightweight virtual machines to easily and efficiently construct fault-tolerant services. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, 2015.

[63] M. Lu and T.-c. Chiueh. Fast memory state synchronization for virtualization-based fault tolerance. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 534–543. IEEE, 2009.

[64] M. Lu and T.-c. Chiueh. Speculative memory state transfer for active-active fault tolerance. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, 2012.

[65] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[66] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, 2014.

[67] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum. Towards practical default-on multi-core record/replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[68] D. Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[69] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2013.

[70] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Fourteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 73–84, Mar. 2009.

[71] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Nov. 2013.

[72] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[73] Nginx web server. `https://nginx.org/`, 2012.

[74] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, June 2014.

[75] 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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 177–192, Oct. 2009.

[76] S. Pertet and P. Narasimhan. Causes of failure in web applications (cmu-pdl-05-109). *Parallel Data Laboratory*, page 48, 2005.

[77] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the Eleventh Symposium on Operating Systems Design and Implementation (OSDI '14)*, Oct. 2014.

[78] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, 2015.

[79] Postgresql. `https://www.postgresql.org`, 2012.

[80] `http://www.qemu.org`.

[81] `http://redis.io/`.

[82] V. A. Sartakov and R. Kapitza. Multi-site synchronous vm replication for persistent systems with asymmetric read/write latencies.

[83] D. J. Scales, M. Nelson, and G. Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev.*, Dec. 2010.

[84] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 29–44, June 2004.

[85] `ssdb.io/`.

[86] R. E. Strom, D. F. Bacon, and S. Yemini. *Volatile logging in n-fault-tolerant distributed systems*. IBM Thomas J. Watson Research Division, 1987.

[87] S. Suneja, C. Isci, V. Bala, E. de Lara, and T. Mummert. Non-intrusive, out-of-band and out-of-the-box systems monitoring in the cloud. *SIGMETRICS Perform. Eval. Rev.*, June 2014.

[88] S. Technologies. Transient error protection. 2005.

[89] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):42:1–42:36, 2015.

[90] `http://www.vmware.com/solutions/vla/`.

[91] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002.

[92] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: Fast and scalable Paxos on RDMA. In *Proceedings of the Eighth ACM Symposium on Cloud Computing (Santa Clara, CA, USA*, 2017.

[93] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228, Apr. 2009.

[94] M. Zaharia, B. Hindman, A. Konwinski, A. Ghodsi, A. D. Joesph, R. Katz, S. Shenker, and I. Stoica. The datacenter needs an operating system. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, 2011.

Odin: Microsoft's Scalable Fault-Tolerant CDN Measurement System

Matt Calder,[†⋆] *Manuel Schröder,*[†] *Ryan Gao,*[†] *Ryan Stewart,*[†] *Jitendra Padhye*[†]
Ratul Mahajan,[#] *Ganesh Ananthanarayanan,*[†] *Ethan Katz-Bassett*[§]
Microsoft[†] USC[⋆] Intentionet[#] Columbia University[§]

Abstract

Content delivery networks (CDNs) are critical for delivering high performance Internet services. Using worldwide deployments of front-ends, CDNs can direct users to the front-end that provides them with the best latency and availability. The key challenges arise from the heterogeneous connectivity of clients and the dynamic nature of the Internet that influences latency and availability. Without continuous insight on performance between users, front-ends, and external networks, CDNs will not be able to attain their full potential performance.

We describe Odin, Microsoft's Internet measurement platform for its first-party and third-party customers. Odin is designed to handle Microsoft's large user base and need for large-scale measurements from users around the world. Odin integrates with Microsoft's varied suite of web-client and thick-client applications, all while being mindful of the regulatory and privacy concerns of enterprise customers. Odin has been operational for 2 years. We present the first detailed study of an Internet measurement platform of this scale and complexity.

1 Introduction

Content delivery networks (CDNs) are a key part of the Internet ecosystem. The primary function of a CDN is to deliver highly-available content at high performance. To accomplish this, CDNs deploy Points of Presence (PoPs) around the world that interconnect with other Autonomous Systems (ASes) to provide short, high quality paths between content and end users.

While a CDN's goal is to deliver the best performance to all users in a cost-effective manner, the dynamic, heterogeneous, and distributed nature of the Internet makes this difficult. CDNs serve content to users all over the world, across tens of thousands of ASes, using various forms of Internet access and connection quality. User performance is impacted by Internet routing changes, outages, and congestion, all of which can be outside the control of the CDN. Without constant insight into user performance, a CDN can suffer from low availability and poor performance. To gain insight into user performance, CDNs need large-scale measurements for critical CDN operations such as traffic management [1,2,3,4,5], Internet path performance debugging [6,7], and deployment modeling [8].

Microsoft operates a CDN with over 100 PoPs around the world to host applications critical to Microsoft's business such as Office, Skype, Bing, Xbox, and Windows Update. This work presents our experience designing a system to meet the measurement needs of Microsoft's global CDN. We first describe the key requirements needed to support Microsoft's CDN operations. Existing approaches to collecting measurements were unsuitable for at least one of two reasons:

- **Unrepresentative performance.** Existing approaches lack coverage of Microsoft users or use measurement techniques that do not reflect user performance.
- **Insensitive to Internet events.** Existing approaches fail to offer high measurement volume, explicit outage notification, and comparative measurements to satisfy key Microsoft CDN use cases.

Next we present the design of Odin, our scalable, fault-tolerant CDN measurement system. Odin issues active measurements from popular Microsoft applications to provide high coverage of Internet paths from Microsoft users. It measures to configurable *endpoints*, which are hostnames or IP addresses of remote target destinations and can be in Microsoft or external networks. Measurement allocation is controlled by a distributed web service, enabling many network experiments to run simultaneously, tailoring measurements on a per-use-case basis as necessary. Odin is able to collect measurements even in the presence of Microsoft network failures, by exploiting the high availability and path diversity offered by third party CDNs. Last, we demonstrate that Odin enables important Microsoft CDN use cases, including improving performance.

There are two key insights that make our design distinct and effective. Firstly, first-party CDNs have an enormous advantage over third-party CDNs in gathering rich measurement data from their own clients. Secondly, integration with external networks provides a valuable opportunity for rich path coverage to assist with network debugging and for enabling fault-tolerance.

2 Background

This section provides background about content delivery networks and Microsoft's deployment.

2.1 Content Delivery Networks

Architecture. A content delivery network (CDN) has geographically distributed server clusters (known as *front-ends*, edges, or proxies), each serving nearby users to shorten paths and improve performance [3, 9, 10, 11] (see Figure 1). Many front-ends are in CDN "points of presence" (PoPs), physical interconnection points where the CDN peers with other ISPs. CDNs typically deploy PoPs at major metro areas. Some CDNs also deploy front-ends in end-user networks or in datacenters. A front-end serves cached content immediately and fetches other content from a *back-end*. Back-ends can be complex web applications. Some CDNs operate *backbone* networks to interconnect PoPs and back-ends.

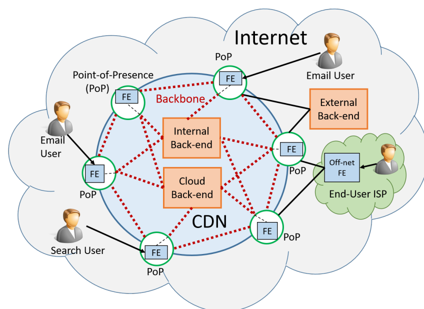


Figure 1: High-level architecture of many CDNs.

Microsoft's CDN is a "hybrid cloud" CDN, i.e., it is used for both its own first-party content as well as for other large third-party customers such as streaming services and online newspapers.

CDN services. Two CDN services are relevant to our paper. Front-ends *cache* static content such as images, JavaScript, CSS, and video. Front-ends also serve as *reverse proxies*, terminating users' TCP connections and multiplexing the requests to the appropriate back-ends via pre-established warm TCP connections [12, 13]. The back-ends forward their responses back to the same front-ends, which relay the responses to the users. Reverse proxies accelerate websites because shorter round-trip times between clients and front-ends enable faster congestion window growth and TCP loss recovery [9].

Client redirection. This work considers the two most common *redirection* mechanisms CDNs use to direct a client request to a front-end for latency sensitive traffic: anycast and DNS. With both mechanisms, when a user desires CDN-hosted content, it issues a request to its *local DNS resolver (LDNS)* for the hostname of the content. The LDNS forwards the request to the CDN's authoritative DNS resolver, and the authoritative resolver returns a record with an IP address of a front-end that can serve the content. With *anycast*, this IP address is announced by multiple PoPs, and BGP routes a request to a PoP based on BGP's notion of best path. The front-end collocated with that PoP then serves the request. With *DNS*-based redirection, the CDN's authoritative resolver returns an IP address for the particular front-end the CDN wants to serve the user from. Because the request to the authoritative resolver generally includes the LDNS (but not user's) IP address, CDN performance benefits from knowledge of which users the LDNS serves (§8.1.1).

2.2 Microsoft's network

Microsoft provides high performance and availability to its customers using a global network with 100+ PoPs, many datacenters, and a Microsoft-operated backbone network interconnecting them. Microsoft operates two types of datacenters. One set is Microsoft's Azure public cloud compute platform [14] which currently has 36 *regions*. The second consists of legacy datacenters, predating Azure. Third-party cloud tenants only run in the Azure datacenters, whereas first-party services operated by Microsoft run in both types. Figure 1 shows Azure regions as "Cloud Back-ends" and private datacenters as "Internal Back-ends".

Redirection of first-party and third-party clients Microsoft currently runs two independent CDNs. A first-party *anycast* CDN runs Microsoft services such as Bing, Office, and XBox [15, 16]. It has more than 100 front-end locations around the world, collocated with all PoPs and several Microsoft public and private datacenters. The second CDN is an Azure traffic management service offered to Azure customers with applications deployed in multiple regions. Whereas Microsoft's first party CDN uses anycast to steer clients, its Azure service uses DNS to direct users to the lowest-latency region. After receiving the DNS response, users connect directly to an Azure region.

2.3 Comparison to other CDNs

Microsoft's architecture closely mirrors other CDNs, especially hybrid-cloud CDNs from Google and Amazon.
End-user applications. All three have web, mobile, and desktop application deployments with large global user bases. Google's include the Chrome Browser, Android OS, Search, YouTube, and Gmail. Amazon's include the Store, Audible, and Prime Video. Microsoft's include Office, Windows OS, Skype, and XBox.
CDN and cloud services. Like Microsoft, Amazon and Google run multiple types of CDNs. Google runs a first-party CDN [6, 7, 9], a third-party CDN [17], and application load balancing across Google Cloud regions [18]. Amazon's equivalent services are CloudFront [19] and Route 53 [20]. Amazon Web Services [21] and Google Cloud Platform [22] are similar to Microsoft Azure [14]. Amazon [10] and Google [23] also run backbone networks.

Because of these similarities, we believe our goals, requirements, and design are applicable to networks beyond Microsoft.

3 Goals and Requirements

We need measurements to support Microsoft's CDN operations and experimentation, leading to the following goals and resulting requirements.

Goal-1: Representative performance reflecting what users could achieve on current and alternate routes.

Requirement: High coverage of paths between Microsoft's users and Microsoft is critical for traffic engineering, alerts on performance degradation, and "what-if" experimentation on CDN configurations, to avoid limited or biased insight into the performance of our network. In particular, our measurements should cover paths to /24 prefixes that combine to account for 90% of the traffic from Microsoft. In addition, they should cover paths to 99% of designated "high revenue" /24 prefixes, which primarily are enterprise customers.

Requirement: Coverage of paths between Microsoft users and external networks to help detect whether a problem is localized to Microsoft and to assess the performance impact of expanding Microsoft's footprint to new sites. External networks may be any CDN, cloud provider, or virtual private server hosting service.

Requirement: Measurements reflect user-perceived performance, correlating with application metrics and reflecting failure scenarios experienced by production traffic, to enable decisions that improve user experience.

Goal-2: Sensitive and quick detection of Internet events.

Requirement: High measurement volume in order to quickly detect events across a large number of users and cloud endpoints, even if the events impact only a small number. Without high measurements counts, events can be missed entirely, or data quality can be too poor to confidently make measurement-driven traffic engineering choices. A reasonable level of sensitivity is the ability to detect an availability incident that doubles the baseline failure rate, e.g., from 0.1% to 0.2%. Figure 12 in Appendix A shows, if we assume measurements fail independently according to a base failure rate, detecting this change would require at least 700 measurements, and detecting a change from 0.01% to 0.02% would require at least 7000 measurements. For confidentiality reasons, we cannot describe our baseline failure rates, but we consider several thousand measurements within a five minute window from clients served by an ISP within one metropolitan region sufficient for our needs.

Requirement: Explicit outage signals, in order to detect events that impact small groups of clients. Historical trends are too noisy to detect the gray failures that make up the majority of cloud provider incidents [24].

Requirement: Fault tolerance in data collection, to collect operation-critical measurements in the presence of network failures between the client and collector.

Requirement: Comparative measurements in same user session for experimentation, providing accurate "apples-to-apples" comparisons when performing an A/B test and minimizing the chance of changing clients or network conditions coloring the comparison between test and control measurements.

Goal-3: Compatible with operational realities of existing systems and applications.

Requirement: Measurements of client-LDNS associations, which are needed to operate both anycast and DNS-redirection CDNs effectively (§2.1,7.1.1,7.2.1).

Requirement: Minimal updates to user-facing production systems, given that network configuration changes are a common cause of online service outages [25].

Requirement: Application compliance across varying requirements. Each Microsoft application independently determines the level of compliance certifications (FISMA, SOC 1-3, ISO 27018, etc.), physical and logical security, and user privacy protections. Application requirements determine the endpoints that can be measured, set of front-ends that can process the measurements, requirements for data scrubbing and aggregation (e.g., IP blocks), and duration of data retention. These strict security policies stem from Microsoft's enterprise customers. Any cloud provider or CDN that serves enterprises, such as Akamai [26], also need to meet these compliance requirements.

4 Limitations of Existing Solutions

This section describes how existing approaches fail to meet our requirements, summarized in Table 1.

1) Third-party measurements platforms provide insufficient measurement coverage of Microsoft users. Non-commercial measurement platforms such as Planetlab, MLab, Caida ARK, and RIPE Atlas have insufficient coverage, with only a few hundred to few thousand vantage points. The largest, RIPE Atlas, has vantage points in 3,589 IPv4 ASes [27], less than 10% of the number of ASes seen by Microsoft's CDN on a standard weekday.

Commercial measurement platforms also lack sufficient coverage. Platforms including Dynatrace [28], ThousandEyes [29], and Catchpoint [30] offer measurements and alerting from cloud-based agents in tier 1 and "middle-mile" (tier 2 and tier 3) ISPs. Cedexis uses a different approach, providing customers with layer 7 measurements collected from users of Cedexis partner websites [31]. However, none of the platforms provides measurements from more than 45% of Microsoft client

Goals	Requirements	Third-party measurement platforms	Layer 3 measurements from CDN infrastructure	Layer 3, DNS from users	Server-side measurements of client connections	Odin
Representative Performance	Coverage of paths between Microsoft users and Microsoft				✓	✓
	Coverage of paths between Microsoft users and external networks				✓	✓
	Measurements reflect user-perceived performance	✓			✓	✓
Sensitive to Internet Events	High measurement volume		✓	✓	✓	✓
	Explicit outage signal	✓	✓	✓		✓
	Fault tolerance	✓	✓	✓		✓
	Comparative measurements in same user session for experimentation		✓	✓	✓	✓
Compatible with Operational Realities	Measurements of client-LDNS associations	✓		✓	✓	✓
	Minimal updates to user-facing production systems	✓	✓	✓		✓
	Application compliance	✓	✓		✓	✓

Table 1: Goals of Odin and requirements to meet our operational CDN needs. No existing approach satisfies all the requirements.

/24 networks. On top of missing over half the networks, the platform with the best coverage provides 10^+ measurements a day from less than 12% of the networks and 100^+ measurements a day from only 0.5% of them, not enough to meet Microsoft's operational need for sensitivity to Internet events.

2) Layer 3 measurements from CDN infrastructure cannot provide representative coverage of the performance of Microsoft users. A CDN can issue measurements such as traceroutes and pings from its front-ends or datacenters to hosts across the Internet. For example, Entact measures the performance along different routes by issuing pings from servers in datacenters to responsive addresses in prefixes across the Internet [1]. One measurement technique used by Akamai is to traceroute from CDN servers to LDNSes to discover routers along the path, then ping those routers as a proxy for CDN to LDNS or end-user latency [32].

However, these measurements cannot provide a good understanding of user performance. Many destinations do not respond to these probes, so Entact was unable to find enough responsive addresses in the networks responsible for 74% of MSN traffic. Similarly, previous work has shown that 45% of LDNS do not respond to ICMP ping or to DNS queries from random hosts [33], and 40% of end users do not respond to ICMP probes [34]. Routers are more responsive than LDNS, with 85% responding to ping [35], but measurements to routers may not reflect a client's application performance because ICMP packets may be deprioritized or rate-limited [36]. All of the above fail to exercise critical layer 7 behaviors including SSL/TLS and HTTP redirection.

3) Layer 3 and DNS measurement from clients may not reflect user-perceived performance and do not provide sufficient coverage. Many systems perform layer 3 measurements from end user devices [37, 38, 39, 40, 41].[1] These measurements are generally dropped by the strict network security policies of enterprise networks. Further, these measurements generally cannot be generated from in-browser JavaScript and instead require installing an application, keeping them from providing measurements from Microsoft's many web users.

4) Server-side measurements of client connections can satisfy some but not all of our use cases. Google [2, 6, 7, 42], Facebook [3], Microsoft [43], and other content providers and CDNs collect TCP- and application-layer statistics on client connections made to their servers [44]. To measure between users and alternate PoPs or paths, CDNs use DNS or routing to direct a small fraction of traffic or client requests to alternate servers or paths. These measurements are useful for performance comparisons, and DNS redirection could steer some of the measurements to measurement servers hosted in external cloud providers. However, if a user cannot reach a server, the outage will not register in server-side measurements, and so these measurements cannot be used to measure fine-grained availability. There are also several practical challenges with only using server-side measurements. While Table 1 shows that technically server-side measurements can be collected on external networks, there are a number of engineering and operational trade-offs that make client-side measurements a better solution for large content providers. The first is that measuring to external networks would mean hosting alternate front-ends on an external provider which immediately raises serious compliance and production concerns. The second issue is that doing A/B network testing with production traffic is considered too high risk with an enterprise customer base.

[1] Ono [37] and Netalyzr [39] also measure throughput.

5 Design Decisions

To meet our goals (§3) and overcome the limitations of other approaches (§4), Odin uses user-side, application-layer measurements of client connections, combining the explicit outage signaling and fault tolerance of user-side measurements (as with layer 3 measurements from users in §4) with the representative performance and coverage achieved by measuring client connections (as with server-side measurements in §4).

Client-side active measurement from Microsoft users. Odin embeds a measurement client into some Microsoft thick clients and web applications. It directs measurement clients to fetch web objects.

This approach helps achieve a number of our requirements. Odin issues measurements from Microsoft users, achieving coverage important to Microsoft's businesses and (by issuing measurements at a rate similar to the use of Microsoft's applications) sensitivity to Internet events, even events that impact only a small fraction of users or connections. By embedding our measurement client into thick clients, Odin can issue measurements even from users unable to reach a Microsoft web server.

Application layer measurements. Odin clients perform DNS resolutions and fetch web objects, measuring availability and timing of these application-layer actions and reporting the results to Odin. The clients can use `http` and `https`, allowing integration with Microsoft applications that require `https`. Unlike ping and traceroute, the measurements are compatible with enterprise networks that host many Microsoft services and users.

These measurements capture the application-layer user performance that we care about, exercising mechanisms across the network stack that can impact performance and availability, including TLS/SSL, web caching, TCP settings, and browser choice. `http` and `https` measurements also provide status code errors that are useful for debugging. They also suffice to uncover user-LDNS associations [45], a key need for both our anycast and DNS redirection CDNs (§7).

External services and smarter clients. We design the clients to conduct measurements and report results even when they cannot reach Microsoft services, as outage reports are some of the most valuable measurements and measurement-dependent operations must continue to function. To build this fault tolerance, clients that cannot fetch measurement configuration or report results fall back to using third-party CDNs for these operations. We use the third-party CDNs to proxy requests to Microsoft and to host static measurement configuration.

Flexible measurement orchestration and aggregation. We built a measurement orchestration system for Odin that supports parallel experiments with different configurations, helping meet a variety of requirements. To accommodate the unique business constraints and compliance requirements of each application that Odin measures to or from, the system provides control over which endpoints an application's users may be given to measure and which servers they upload reports to. When appropriate, experiments can measure to servers in external (non-Microsoft) networks, and clients conduct multiple measurements in a session to allow direct comparisons. By having clients fetch instructions on which active measurements to run, new experiments generally do not require changes to operational services or to clients, reducing operational risk. We also allow for flexibility in aggregation of the measurements (e.g., in 5 minute buckets) for faster upload to our real-time alerting system.

6 System Design

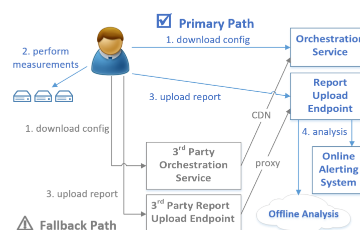


Figure 2: Odin Architecture Overview: CDN clients download measurement config, perform measurements, and upload results. If first-party network sites are unreachable, third-party sites can cache config and relay upload requests.

Figure 2 outlines the Odin measurement process. A number of Microsoft applications embed the Odin client (§6.1). Odin clients in thick applications support a range of measurements. This paper focuses on measuring latency and availability, our highest priorities, supported by thick and web applications.

Step 1: The client uses a background process to fetch a *measurement configuration* from the Orchestration Service (§6.2). The configuration defines the type of measurements and the targets (measurement endpoints).

Step 2: The client issues the measurements. To measure latency and availability, endpoints host a small image on a web server for clients to download. Many Microsoft applications require `https` requests, so measurement endpoints have valid certificates. The endpoints can be in Microsoft front-ends, Microsoft data centers, or third-party cloud/collocation facilities.

Step 3: When the client completes its measurements, it uploads the measurement results to a *Report Upload Endpoint* (§6.3). The Report Upload Endpoint forwards the measurements to Odin's two analysis pipelines.

Step 4: The real-time pipeline performs alerting and network diagnostics, and the offline pipeline enriches measurements with metadata for big data analysis (§6.4).

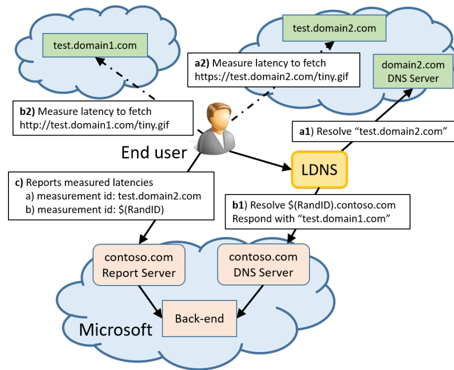


Figure 3: Odin supports two measurement types for latency. Measurement *a* measures the test domain directly. Measurement *b* contacts an Odin authoritative DNS first, which responds with the endpoint to measure. This gives Odin client-LDNS association for a measurement.

6.1 Client

Measurements can vary along 3 dimensions: `http` or `https`, direct or DNS-based, and warm or cold. Figure 3 illustrates direct and DNS-based measurements. The first type has the client performing DNS resolution of `test.domain2.com` in *a1* and fetching the image and recording the latency in *a2*. The measurement to `<randid>.contoso.com` is an example of the second type, which we refer to as a *DNS-based measurement* and which we use to measure web fetch latency *and* client-LDNS association. The domain (`contoso.com`) is one that we control. We design the clients to recognize the `<randid>` scheme and substitute in a unique, transient, random identifier **$(RandID)**.[2] The client then issues a DNS request via the user's LDNS for **$(RandID)**`.contoso.com` (step *b1*). The DNS request goes to our authoritative DNS server, which returns a record for the endpoint Odin wants to measure (`test.domain1.com`) and logs its response associated with the **$(RandID)**. The client then fetches `http://test.domain1.com/tiny.gif`. In step *c*, the client reports its measurements, reporting the ID for the second measurement as **$(RandID)**. The measurement back-end uses **$(RandID)** to join the client's IP address with the DNS log, learning the user-LDNS association.

The Orchestration Service can ask clients to perform "cold" and/or "warm" measurements. A cold measurement initiates a new TCP connection to fetch the image. A warm measurement fetches the image twice and reports the second result, which will benefit from DNS caching and from a warm TCP connection.[3]

[2] Generating the **$(RandID)** at the client rather than at the Orchestration Service lets caches serve the measurement configuration.

[3] The client prevents browser caching by appending a random parameter to the image request (e.g. `tiny.gif?abcde12345`).

Web client vs. thick client measurements. Web clients default to measuring latency using the request start and end times in JavaScript, which is known to be imprecise [46]. If the browser supports the W3C resource-timing API [46], then the client reports that more precise measurement instead, along with a flag that signals that it used the more precise option. If the image fetch fails, the client reports the HTTP error code if one occurred, otherwise it reports a general failure error code. A limitation of in-browser measurements is that low-level networking errors are not exposed to JavaScript. For example, we cannot distinguish between a DNS resolution failure and a TCP connection timeout. Thick clients issue measurements through an Odin application SDK. Unlike web clients, the SDK can report specific low-level networking errors which are valuable in debugging.

6.2 Orchestration Service

The Orchestration Service coordinates and dispatches measurements. It is a RESTful API service that Odin clients invoke to learn which measurements to perform. The service returns a small JSON object specifying the measurements. In the rare case of major incidents with Odin or target Microsoft services, the Orchestration Service has the option to instruct the client to issue no measurements to avoid aggravating the issues.

```
NumMeasurements: 3,
MeasurementEndpoints: [
   {type:1, weight:10, endpoint:"m1.contoso.com"},
   {type:1, weight:20, endpoint:"m2.microsoft.com"},
   {type:2, weight:30, endpoint:"m3.azure.com"},
   {type:3, weight:10, endpoint:"m4.azure.com"},
   {type:2, weight:30, endpoint:"m5.azure.com"},
   {type:1. weight:15, endpoint:"m6.microsoft.com"}],
ReportEndpoints: ["r1.azure.com","r2.othercdn.com"]
```

Listing 1: Example measurement configuration that is served by the orchestration service to the client.

Listing 1 shows an example configuration that specifies three measurements to be run against three out of six potential endpoints. The ability to specify more endpoints than measurements simplifies configurations that need to "spray" measurements to destinations with different probabilities, as is common in CDN A/B testing [16]. The client performs a weighted random selection of three endpoints.

The other component of orchestration is the customized authoritative DNS server for DNS-based measurements (§6.1). When a client requests DNS resolution for a domain such as `12345abcdef.test.contoso.com`, the DNS server responds with a random record to a target endpoint, with the random choice weighted to achieve a desired measurement distribution.

Even a unique hostname used for client-LDNS mapping can generate multiple DNS queries. Our measurements reveal that 75% of unique hostnames result in multiple LDNS requests, and 70% result in requests from multiple LDNS IP addresses. If our authoritative DNS returned different responses for a single hostname, we would be unable to determine from logs which target endpoint the client actually measured. To overcome this issue, we use consistent hashing to always returns the same response for the same DNS query.

The Orchestration Service allocates measurements to clients based on *experiments*. An experiment has an Orchestration Service configuration that specifies the endpoints to be measured, which applications' users will participate, and which report endpoints to use based on compliance requirements of the applications. Experiment owners configure endpoint measurement allocation percentages, and the Orchestration Service converts them into weights in the configuration. The Orchestration Service runs multiple experiments, and experiments may be added or removed at any time.

The Orchestration Service allows Odin to tailor configurations to meet different measurement needs and use cases. For example, the service can generate specialized configuration for clients depending on their geography, connection type, AS, or IP prefix. When experimenting with CDN settings, we tailor Odin configurations to exercise the experimental settings from clients in a particular metropolitan area and ASN. When debugging a performance issue, we tailor configurations to target measurements to an endpoint experiencing problems.

If the Orchestration Service is unavailable, proxies in third-party networks may be used instead. The proxies may act as reverse proxies for the first-party system. Alternatively, if the first-party system is unavailable, a fallback to a cached default configuration can be returned to clients.

6.3 Reporting

Listing 1 shows that the measurement configuration returned by the Orchestration Service also specifies the *primary* and *backup* `ReportEndpoints` for the client to upload measurement results. ReportEndpoints are hosted across the 100+ front-ends of Microsoft's first-party CDN. When a ReportEndpoint receives client measurement results, it forwards them to two Microsoft data pipelines, as shown in Figure 2. If for some reason the Microsoft CDN is unavailable, the client will fall back to using proxies hosted in third-party networks. The proxies forward to a set of front-ends that are not part of the primary set of front-ends.

Fault-tolerant measurement reporting is necessary to support our requirement of an explicit outage signal, since we cannot measure the true availability of Microsoft's first-party CDN if we also report measurements there. Odin's fault-tolerant approach for fetching measurement configuration and uploading results will succeed if the backup reporting channel(s) use a path that avoids the failure and fail if both the primary and backup paths encounter the failure. As long as the client can reach a backup, and the backup can reach at least one of the Odin servers at Microsoft, Odin receives the result, tolerating all but widespread failures that are detectable with traditional approaches and are often outside of Microsoft's control to fix. From operational experience, Odin's handling of faults provides a high level of resilience for our measurement data. We now discuss Odin's behavior in the face of three fault scenarios. We do not consider this an exhaustive treatment of all possible fault scenarios.

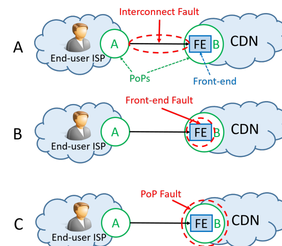


Figure 4: Three types of Internet faults that may occur when fetching measurement configuration or uploading reports.

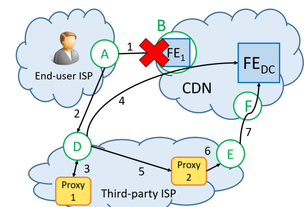


Figure 5: Topology of backup paths when FE_1 is unreachable. FE_1 is a front-end collocated with a PoP while FE_{DC} is a front-end in a datacenter.

Interconnection faults impact an individual link(s) between an end-user ISP and the CDN, caused by issues such as peering misconfiguration or congestion. Connectivity to other ISPs is not impacted. Figure 4(A) shows an interconnection fault between PoPs A and B. Figure 5 shows that, when these faults occur, the client will send a backup request using path $2,3$ to Proxy 1. The proxy then forwards the request back to the CDN by path $3,4$, through D, to a datacenter front-end FE_{DC} instead of FE_1.

Front-end system faults are failures of a front-end due to software or hardware problems, as shown in Figure 4(B). Because the backup proxies connect to a distinct set of front-ends (hosted in datacenters), we gain resilience to front-end system faults, as seen in Figure 5.

PoP-Level faults impact multiple ISPs and a large volume of traffic exchanged at that facility. These faults may be caused by a border router or middle-box misconfiguration or a DDoS attack. In our experience, these faults are rare and short-lived, and so we did not design Odin to be resilient to them. Figure 5 shows that Proxy 1's path to FE_{DC} goes through the same PoP as the client's

path to FE_1, whereas Proxy 2 avoids it. We present a preliminary evaluation of this scenario in Section 8.2.

6.4 Analysis Pipeline

Measurement reports get sent to two analysis pipelines.

Offline Pipeline. The measurement reports include a report ID, metadata about the client (version number of client software, ID of Microsoft application it was embedded in, whether it used the W3C Resource Timing API), and the measurement results, each of which includes a measurement ID, the measurement latency (or failure information), and whether it was a cold or warm measurement. The offline pipeline enriches measurements with metadata including the client's LDNS and the user's location (metropolitan region), ASN, and network connection type. This enriched data is the basis for most operational applications (§7).

Real-time Alerting Pipeline. Many of the target endpoints that Odin monitors are business-critical so must react quickly to fix high latency or unavailability. To ensure fast data transfer to the back-end real-time data analytics cluster, each reporting endpoint reduces data volume by aggregating measurements within short time windows. It annotates each measurement with the client's metropolitan region and ASN, using in-memory lookup tables. Within each window, it groups all measurements to a particular endpoint from clients in a particular ⟨metropolitan region, ASN⟩, then reports fixed percentiles of latency from that set of measurements, as well as the total number of measurements and the fraction of measurements that failed.

7 Supporting CDN Operations with Odin

We use Odin to support key operational concerns of CDNs – performance and availability, plus CDN expansion/evolution and how it impacts the other concerns. The two CDNs we support have sizes of over a hundred sites (which is more than most CDNs) and few dozen sites (which is common for CDNs [47]).

7.1 Directing users to the CDN front-ends

Low latency web services correlate with higher user satisfaction and service revenue. A central proposition of a CDN is that distributed servers can serve users over short distances, lowering latency, but deploying the servers alone does not suffice to achieve that goal.

Odin continuously monitors client performance for both of Microsoft's CDNs. Previous work demonstrated the value of comparing performance of our CDN to another to guard against latency regressions [15]; of comparing performance from one client to multiple CDN servers [16], and of comparing the performance from a CDN to multiple clients in the same city [6]. Odin provides measurements for all these analyses, which can uncover performance problems stemming from circuitous routing in either direction or from poor server selection. This section describes how we use Odin to create high-performance redirection maps for DNS redirection (§7.1.1) and to identify cases in which Internet routing selects poor performing anycast routes (§7.1.2).

7.1.1 Generating low latency DNS redirection maps

Azure's traffic manager service (§2.2) directs a user to a particular Azure region [14] by returning a DNS record for that region. When determining which DNS record to return, the traffic manager knows which LDNS issued the request but not which user.[4] We refer to an instance of the DNS redirection policy as a *map* (from LDNS to IP addresses of Azure regions).

To achieve low latency for users, we need to understand which use each LDNS and their performance to the various regions. Microsoft constructs maps using Odin data as the primary data source, as follows:

(1) Data Aggregation. The offline pipeline annotates each DNS-based measurement with the LDNS the client used (§6.4). We use this associate to group the measurements directed by each LDNS to each Azure region and calculate the median latency to each region from each LDNS. (In practice, before finding the median latency, we aggregate all LDNS within the same /26 IP prefix, which we found balances precision because of IP localization and statistical power from measurement aggregation.)

(2) Filtering. Next, we filter out LDNS-region pairs which do not have enough measurements. Our minimum threshold was chosen using statistical power analysis. If we filter the region that was lowest latency for the LDNS in the currently-deployed map, we do not update the map for the LDNS, to prevent us from making the best decision from a set of bad choices.

(3) Ranking Results. For each LDNS, we rank the regions by latency. At query resolution time, the traffic management authoritative DNS responds to an LDNS with the lowest latency region that is currently online.

(4) Applying the Overrides. The final step is to apply the per-LDNS changes to the currently deployed map, resulting in the new map. The map generation process takes care of prefix slicing, merging, and aggregation to produce a map with a small memory footprint.

7.1.2 Identifying and patching poor anycast routing

Microsoft's first-party CDN uses anycast (§2.2). Anycast inherits from BGP an obliviousness to network performance and so can direct user requests to suboptimal front-ends. We identify incidents of poor anycast routing in Microsoft's anycast CDN by using Odin to measure

[4] Except for the few LDNS that are ECS-enabled [48, 49].

performance of anycast and unicast alternatives from the same user. Our previous study used this methodology for a one-off analysis using measurements from a small fraction of Bing users [16]. Odin now continuously measures at a large scale and automatically generates daily results. As with our earlier work, we find that anycast works well for most–but not all–requests. The traditional approach to resolving poor anycast routing is to reconfigure route announcements and/or work with ISPs to improve their routing towards the anycast address.

While Microsoft pursues this traditional approach, announcements can be difficult to tune, and other ISPs may not be responsive, and so we also patch instances of poor anycast performance using a hybrid scheme that we proposed (but did not implement) in our previous work [16]. The intuition is that both DNS redirection and anycast work well most of the time, but each performs poorly for a small fraction of users. DNS redirection cannot achieve good performance if an LDNS serves widely distributed clients [32], and anycast performs poorly in cases of indirect Internet routing [16]. Since the underlying causes do not strongly correlate, most clients that have poor anycast performance can have good DNS redirection performance. We use Odin measurements to identify these clients, and a prominent Microsoft application now returns unicast addresses to the small fraction of LDNS that serve clients with poor anycast routing.

7.2 Monitoring and improving service availability

The first concern of most user-facing web services is to maintain high availability for users, but it can be challenging to quickly detect outages, especially those that impact only a small volume of requests.

Odin's design allows it to monitor availability with high coverage and fast detection. By issuing measurements from the combined user base of a number of services, it can detect issues sooner than any individual service. By having a single client session issue measurements to multiple endpoints, sometimes including an endpoint outside of Microsoft's network, it can understand the scope of outages and differentiate client-side problems from issues with a client contacting a particular service or server. By providing resilient data collection even in the face of disruptions, Odin gathers these valuable measurements even from clients who cannot reach Microsoft services. Anycast introduces challenges to maintaining high availability. This section discusses how Odin helps address them.

7.2.1 Preventing anycast overload

Monitoring a front-end's ability to control its load. Previous work from our collaborators demonstrated how Microsoft's anycast CDN prevents overload [50]. The approach works by configuring multiple anycast IP addresses and organizing them into a series of "rings" of front-ends. All front-ends are part of the largest ring, and then each subsequent ring contains only a subset of the front-ends in the previous one, generally those with higher capacity. The innermost ring contains only high capacity data centers. Each front-end also hosts an authoritative nameserver. If a front-end becomes overloaded, its authoritative nameserver "sheds" load by directing a fraction of DNS queries to a CNAME for the next ring. These local shedding decisions work well if anycast routes a client's LDNS's queries and the client's HTTP requests to the same front-end, in which case the authoritative nameserver can shed the client's requests.

The previous work used measurements from Odin to evaluate how well HTTP and DNS queries correlate for each front-end [50], a measure of how controllable its traffic is. Odin now continuously measures the correlations and controllability of each front-end, based on its measurements of client-to-LDNS associations.

Designing rings with sufficient controllability. We use Odin data on per front-end controllability to design anycast rings that can properly deal with load. The data feeds a traffic forecasting model that is part of our daily operation. The model predicts per front-end peak load, broken down by application, given a set of rings.

Two scenarios can compromise a front-end's ability to relieve its overload. First, the above approach sheds load at DNS resolution time, so it does not move existing connections. This property is an advantage in that it does not sever existing connections, but it means that it cannot shed the load of applications with long-lived TCP connections. Second, if a front-end receives many HTTP queries from users whose DNS queries are not served from the front-end, it can potentially be overwhelmed by new connections that it does not control, even if it is shedding all DNS requests to a different ring.

We use Odin measurements in a process we call *ring tuning* to proactively guard against these two situations. For the first, we use measurements to identify a high-correlation set of front-ends to use as the outermost anycast ring for applications with long-lived connections. The high-correlation allows a front-end that is approaching overload to quickly shed any new connections, both from the long-lived application and other applications it hosts on other anycast addresses. To guard against the second situation, we use measurements to design rings that avoid instances of uncontrollable load, and we configure nameservers at certain front-ends to shed all requests from certain LDNS to inner rings, to protect another front-end that does not control its own fate.

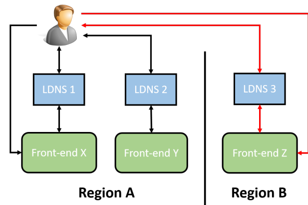


Figure 6: In a regional anycast scenario, if a user's LDNS is served by a front-end in the user's region, the user's performance is unaffected. If the user's LDNS is served by a front-end in a different region, then the user will be served from the distant region, likely degrading performance.

7.2.2 Monitoring the impact of anycast route changes on availability

Long-lived TCP connections present another challenge to anycast: an Internet route change can shift ongoing TCP connections from one anycast front-end to another, severing the connections [51, 52, 53, 54]. Odin measurements address this concern in two ways. First, by having clients fetch an object from both an anycast address and a unicast address, we can monitor for problems with anycast availability. Second, we use Odin to monitor the availability of different candidate anycast rings in order to identify subsets of front-ends with stable routing.

7.3 Using measurements to plan CDN evolution

7.3.1 Comparing global vs regional anycast

In a basic anycast design, all front-ends share the same global IP address. However, this address presents a single point of failure, and a routing configuration mistake at Microsoft or one of our peers has the potential to blackhole a large portion of our customer traffic. An alternate approach is to use multiple regional anycast addresses, each announced by only a subset of front-ends. Such an approach reduces the "blast radius" of a potential mistake, but it can also change the performance of the CDN. A user's request can only end up at one of the front-ends that announces the anycast address given to its LDNS, which might prevent the request from using the best performing front-end . . . or prevent Internet routing from delivering requests to distant front-ends.

Figure 6 shows the three scenarios that can occur when partitioning a global anycast into regions. A user's LDNS may be served by the same front-end as the user or by a different one. If different, the front-ends may be assigned to the same or different regions. If they are assigned to different regions, then the user will be directed away from its global front-end to a different one, likely degrading performance.

In a use case similar to anycast ring tuning, we used Odin to collect data, then used a graph partitioning algorithm to construct regions that minimize the likelihood that a user and their LDNS are served by front-ends in different regions. We construct a graph where vertices represent front-ends and edges between vertices are weighted proportional to the traffic volume where one endpoint serves the DNS query and the other serves the HTTP response. We use an off-the-shelf graph partitioning heuristic package to define 5 regions, each with approximately the same number of front-ends, that minimizes the number of clients directed to distant regions. We compare the performance of regional versus global anycast in Section 8.3.

Country	P75 Imp.	P95 Imp.	Country	P75 Imp.	P95 Imp.
Spain	30.68%	10.79%	Switzerland	10.67%	22.18%
Italy	29.92%	17.95%	Netherlands	7.22%	24.94%
Japan	28.14%	32.02%	France	6.60%	18.14%
Australia	20.05%	16.82%	Norway	5.61%	14.93%
Canada	19.17%	5.10%	U.K.	4.44%	12.39%
Sweden	14.14%	24.02%	Germany	2.82%	5.49%
U.S.A.	14.04%	8.81%	Finland	1.56%	12.97%
South Africa	13.97%	6.33%	Brazil	0.68%	6.18%
India	13.97%	6.08%			

Table 2: The performance improvement in the 75th and 95th percentile from a 2 month roll-out using the Odin-based mapping technique over May and June 2017.

8 Evaluation and Production Results

Odin has been running as a production service for 2 years. It has been incorporated into a handful of Microsoft applications, measuring around 120 endpoints.

8.1 Odin improves service performance

8.1.1 Odin's DNS redirection maps reduce latency

In May 2017, the Azure traffic manager began directing production traffic using maps generated as described in Section 7.1.1, replacing a proprietary approach that combined geolocation databases with pings from CDN infrastructure. We evaluated the performance of the two maps by running an Odin experiment that had each client issue two measurements, one according to each map. Table 2 shows the latency change at the 75th and 95th percentile for the countries with the most measurements. Finland and Brazil saw negligible latency increases (1.56%, 0.68%) at the 75th percentile, but all other high traffic countries saw reductions at both percentiles, with a number improving by 20% or more.

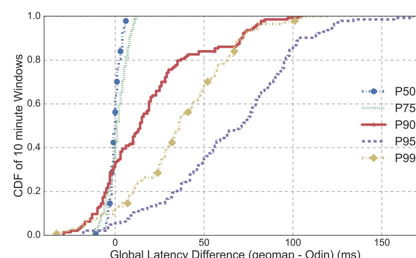


Figure 7: Difference in global performance over one day between a Odin map and a map generated from LDNS locations (geomap). Values less than 0 show the fraction of time that the geomap performed better.

Comparison with alternative DNS mapping techniques. A simple approach to generating a redirection map for a CDN is to use the locations of LDNSes. To test the performance difference between this and the Odin approach, we generated a map using a proprietary Microsoft geolocation database that aggregates locations from many sources. For every IPv4 address, we find the geographically closest front-end and choose that for the map entry. We aggregate neighboring IP addresses with the same map entry and convert this into a set of DNS records. We then configured Odin to measure both this map and the current Odin-based map for 24 hours on Sept. 21, 2017. We bucketed measurements into 10-minute bins. For each bin, we calculated the latency differences at different percentiles. Figure 7 depicts a CDF over all 10 minute bins. Most of the time there is no real difference at the median. The difference is also small at the 75th percentile, although Odin is better around 65% of the time. The real improvement of using Odin comes at the 90th, 95th, and 99th percentile. At P95, Odin's map is better by 65ms half the time.

Dispelling mistaken conventional wisdom. Prior work on CDN performance sometimes exhibited misconceptions about DNS redirection, because operational practices were not transparent to the research community. We distill some takeaways from our work that contradict prior claims and elucidate realities of modern CDNs.

- *For many CDNs, measurements of user connections suffice as the key input to map generation,* whereas previous work often describes mapping as a complex process requiring many different types of Internet measurements [4], including measurements from infrastructure to the Internet [6, 55]. This reality is especially true for CDNs that host popular first-party services, as the CDN has measurement flexibility and control over first party services.
- *The geographic or network location of an LDNS does not impact the quality of redirection*, even though redirection occurs at the granularity of an LDNS. Previous work claimed that redirection decisions were based on the location of or measurements to the LDNS [55], or that good decisions depending on users being near their LDNS [45, 56, 57]. In reality, techniques for measuring associations between users and LDNS have been known for years [45], allowing decisions based on the performance of the users of an LDNS to various front-ends, which provides good performance as long as the users of an LDNS experience good performance from the same front-end as each other.
- *Most redirection still must occur on a per LDNS basis,* even though EDNS client-subnet (ECS) enables user prefix-granularity decisions [32, 48, 55, 58]. Our measurements reveal that, outside of large public resolvers, almost no LDNS operators have adopted ECS.

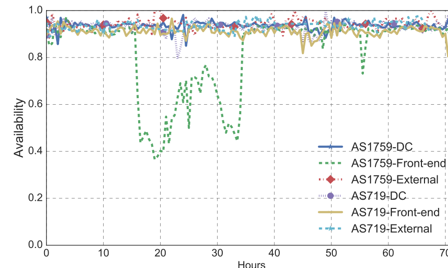


Figure 8: Debugging 2017 availability drop between Helsinki front-end and AS1759 users in Finland.

8.1.2 Odin patches anycast performance

Due to space constraints, we summarize the results from our earlier work [16]. Anycast directed 60% of requests to the optimal front-end, but it also directed 20% of requests to front-ends that were more than 25ms worse than the optimal one. Today we use Odin measurements to derive unicast "patches" for many of those clients.

8.2 Using Odin to identify outages

An outage example. Figure 8 visualizes Odin measurements showing an availability drop for Finnish users in AS1759 during a 24 hour period in 2017. The availability issue was between users in that ISP and a Microsoft front-end in Helsinki. Because Odin measures from many Microsoft users to many endpoints in Microsoft and external networks, it provides information that assists with fault localization. First, we can examine measurements from multiple client ISPs in the same region towards the same endpoint. For readability, we limit the figure to one other ISP, AS719, which the figure shows did not experience an availability drop to the front-end. So, the front-end is still able to serve some user populations as expected. Second, the figure indicates that AS1759 maintains high availability to a different endpoint in Microsoft's network, a nearby datacenter. So, there is no global connectivity issue between Microsoft and AS1759. Last, the figure indicates that availability remains high between clients in AS1759 and an external network. The rich data from Odin allows us to localize the issue to being between clients in AS1759 and our Helsinki front-end.

Reporting in the presence of failures. Odin successfully reports measurements despite failures between end-users and Microsoft. Figure 9 shows the fraction of results reported via backup paths for representative countries in different regions, normalized by the minimum fraction across countries (for confidentiality). During our evaluation period, there were no significant outages so the figure captures transient failures that occur during normal business operations. All countries show a strong

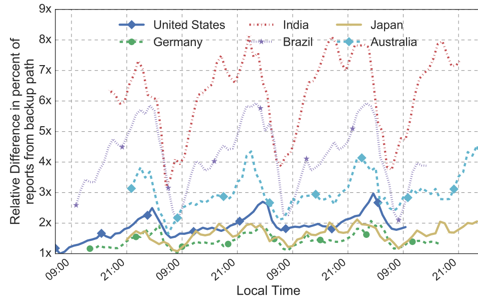


Figure 9: Relative difference per hour in percentage of reports received through the backup path across four weekdays.

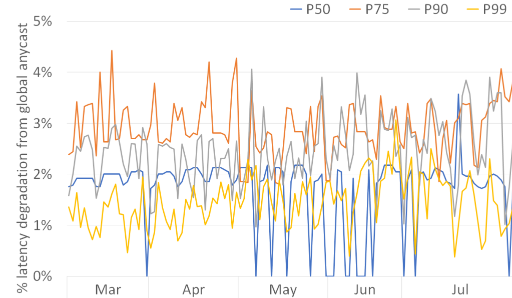


Figure 10: Latency degradation of 5-region vs. global anycast.

diurnal pattern with peaks around midnight and valleys around 8 a.m. local time. Interestingly, the peaks of highest failover reports occur well outside of working hours, when Microsoft's traffic volume is low. This is consistent with previous work which found that search performance degraded outside business hours, because of an increase in traffic from lower quality home broadband networks relative to traffic from well-provisioned businesses [43].

The percentage of reports uploaded through third-parties varies significantly across countries. For example, at peak, Brazil have 3x and 4x the percentage of backup reports as compared to Germany and Japan. Another distinguishing characteristic across countries is the large difference in range between peaks and valleys. India ranges from $\approx 3\times$ to $\approx 8\times$ the baseline, Australia from $\approx 2\times$ to $\approx 4\times$, and Japan from $\approx 1\times$ to $\approx 2\times$

Backup path scenarios. Backup proxies forward report uploads to datacenter front-ends instead of front-ends collocated with PoPs (§6). To illustrate why this is necessary, we allocated a small fraction of Odin measurements to use an alternate configuration in which the third-party proxies instead forward traffic to an anycast ring consisting of front-ends at the same PoPs as the primary `ReportEndpoints`. The third party CDN has roughly the same number of front-end sites as Microsoft. Out of 2.7 billion measurements collected globally over several days in January 2018, around 50% were forwarded to the same front-end by both the third-party proxies and the primary reporting pathway, meaning that the reports could be lost in cases of front-end faults.

Fault-tolerance for PoP-level failures. Figure 4(C) shows an entire PoP failing. It is likely that the nearest front-end and nearest backup proxy to the end-user are also relatively close to each other. When the proxy forwards the request, it will likely ingress at the same failing PoP, even though the destination is different.

To route around PoP-level faults, we want the end-user to send the backup request to a topologically distant proxy, such as Proxy 2 in Figure 5. The proxy will forward the request through nearby CDN PoP F and avoid the failure. To test this, we configured two proxy instances in a popular cloud provider, on the East and West Coasts of the United States. These proxies forward requests to the set of front-ends collocated at Microsoft PoPs. We configured a load balancing service to send all requests to the East Coast instance by default, but with an exception to direct all requests from East Coast clients to the West Coast proxy. After collecting data globally for several days, we observed that only 3% of backup requests enter Microsoft's network at the same PoP as the primary, as opposed to the 50% above. This prototype is not scalable with so few proxy instances, but demonstrates an approach to mitigate PoP-level faults that we will develop in a future release.

8.3 Using Odin to evaluate CDN configuration

This section uses Odin to evaluate the performance impact of regional rings as compared to a global anycast ring (§7.3.1). The cross-region issue illustrated in Figure 6 still has the potential to introduce poor anycast performance, even though our graph partitioning attempts to minimize it. To measure the impact to end users, we configure an Odin experiment that compares the latency of the regional anycast ring with our standard "all front-ends" ring. Figure 10 shows that performance change at the median is relatively small – just about 2%. The 75th percentile consistently shows the highest percentage of degradation over time, fluctuating around 3%. While the median and 75th percentiles are stable over the five months, both 90th and 99th percentiles begin to trend higher in the starting in May, suggesting that static region performance may change over time at higher percentiles.

8.4 Evaluating Odin coverage

In this section we examine Odin's coverage of Microsoft's users as part of our requirement to cover paths between Microsoft users, Microsoft, and external networks. We will examine four applications which we have integrated with Odin. We have categorized them by their user base: General, Consumer, and Enterprise.

We first look at how much of Microsoft's user base is covered by individual and combined applications. Figure 11 shows Consumer1, Consumer2, and Enterprise1 have similar percent coverage of Microsoft end users by

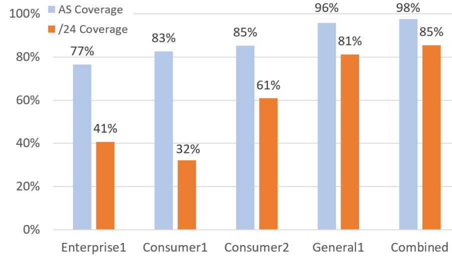


Figure 11: Percentages of ASes and /24s with measurements based on 4 properties with different user populations.

AS. The benefit of multiple applications is more apparent when looking at /24 coverage. We see that all four applications combined cover 85% of /24s whereas individually all except for General1 cover much less. We also examined the overlap between application coverage and found that the four properties only see around 42% pairwise overlap in /24 coverage, meaning that individual applications contribute a substantial amount of user diversity to Odin. General1 is the highest distinct contributor by providing about 18% of distinct /24s observed.

Breaking down the coverage by "high revenue" (e.g. Reverse Proxy for Enterprise scenarios), "medium revenue" (e.g. Consumer email, ad-supported content) and "low revenue" (commodity caching workloads), we observe a higher /24 prefix coverage with Odin for "high revenue" (95%) compared to "medium" (91%) and "low" (90%). This suggests that the missing coverage of Odin is in the tail of relatively low-value traffic.

9 Related Work

There has been a considerable prior work on improving and evaluating general CDN performance [4, 56, 59, 60, 61, 62, 63]. Prior work has also explored cooperation between ISPs and CDNs. Specifically, the efficacy of ISPs releasing distance maps to CDNs to enable more accurate client to server mappings [59], or ISPs hosting CDN servers on demand [60]. WISE [8] is a tool that predicts the deployment implications of new infrastructure in CDNs by using machine learning. WhyHigh [6] and LatLong [7] focus on automating troubleshooting for large content providers like Google, using active measurements and passive latency data, respectively.

Entact [1], EdgeFabric [3], Espresso [2] measure the quality of egress paths from a CDN to the Internet. Entact describes a technique to measure alternative paths to Internet prefixes by injecting specific routes (/32) at border routers to force egress traffic through a particular location. These paths are utilized by a collection of "pinger" machines deployed in DCs to target IPs likely to be responsive within a prefix. EdgeFabric and Espresso direct a small percent of user traffic through alternative egress paths to measure alternate path performance.

Fathom [41], Netalyzr [39], Ono [37], Via [64], Dasu [38] are thick client applications that run measurements from end user machines; BISmark [40] measures from the home routers. Akamai collects client-side measurements using their Media Analytics Plugin [65] and peer-assisted NetSession [32, 66] platform. From commercial measurement platforms, Cedexis is the closest in nature to Odin. Cedexis partners with popular websites with large user bases such as LinkedIn and Tumblr that embed Cedexis' measurement JavaScript beacon into their page. Cedexis customers register their own endpoints to be measured by a portion of end-users of Cedexis' partners. In this way, a customer collects measurements to their endpoints from a relatively large user base. Conviva is a commercial platform which uses extensive client-side measurements from video players to optimize video delivery for content publishers [67, 68].

Akamai published a study on DNS-based redirection [32] showing that enabling ECS [48] greatly improved the performance of user. Alzoubi et al. [51, 53] have examined properties of anycast CDNs. Follow up work focuses on handling anycast TCP session disruption due to BGP path changes [52]. Our work is complementary and orthogonal to our colleagues' work, FastRoute [50], that load balances within an anycast CDN.

Odin uses a user-to-LDNS association technique similar to [34, 45] whereas Akamai uses their NetSession download manager software to obtain client-to-LDNS mappings [32]. Measuring latency using JavaScript beacons is a well established technique [16, 69].

10 Conclusion

CDNs are critical to the performance of large-scale Internet services. Microsoft operates two CDNs, one with 100+ endpoints that uses anycast and one for Azure-based services that uses DNS-redirection. This paper describes Odin, our measurement system that supports Microsoft's CDN operations. These operations span a wide variety of use cases across first- and third-party customers, with clients spread out worldwide. Odin has helped improve the performance of major services like Bing search and guided capacity planning of Microsoft's CDN. We believe that the key design choices we made in building and operating Odin at scale address the deficiencies of many prior Internet measurement platforms.

Acknowledgements

We thank the anonymous NSDI reviewers for a constructive set of reviews. We thank our shepherd, Michael Kaminsky, for providing insightful recommendations and thorough comments on multiple drafts. The work was partially supported by NSF awards CNS-1564242, CNS-1413978, and CNS-1351100.

References

[1] Z. Zhang, M. Zhang, A. G. Greenberg, Y. C. Hu, R. Mahajan, and B. Christian, "Optimizing Cost and Performance in Online Service Provider Networks," in *NSDI*, pp. 33–48, 2010.

[2] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, *et al.*, "Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering," in *SIGCOMM*, pp. 432–445, ACM, 2017.

[3] B. Schlinker, H. Kim, T. Cui, E. Katz-Bassett, H. V. Madhyastha, I. Cunha, J. Quinn, S. Hasan, P. Lapukhov, and H. Zeng, "Engineering Egress with Edge Fabric," in *SIGCOMM*, 2017.

[4] V. Valancius, B. Ravi, N. Feamster, and A. C. Snoeren, "Quantifying the benefits of joint content and network routing," in *SIGMETRICS*, pp. 243–254, ACM, 2013.

[5] H. H. Liu, R. Viswanathan, M. Calder, A. Akella, R. Mahajan, J. Padhye, and M. Zhang, "Efficiently Delivering Online Services over Integrated Infrastructure," in *NSDI*, pp. 77–90, 2016.

[6] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao, "Moving Beyond End-to-End Path Information to Optimize CDN Performance," in *IMC*, 2009.

[7] Y. Zhu, B. Helsley, J. Rexford, A. Siganporia, and S. Srinivasan, "LatLong: Diagnosing wide-area latency changes for CDNs," in *Transactions on Network and Service Management*, 2012.

[8] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering What-If Deployment and Configuration Questions with WISE," in *SIGCOMM*, pp. 99–110, ACM, 2008.

[9] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing Web Latency: The Virtue of Gentle Aggression," *SIGCOMM*, vol. 43, no. 4, pp. 159–170, 2013.

[10] J. Hamilton, "AWS re:Invent 2016: Amazon Global Network Overview." https://www.youtube.com/watch?v=uj7Ting6Ckk.

[11] "Netflix Open Connect." https://media.netflix.com/en/company-blog/how-netflix-works-with-isps-around-the-globe-to-deliver-a-great-viewing-experience.

[12] Y. Chen, S. Jain, V. K. Adhikari, and Z.-L. Zhang, "Characterizing Roles of Front-end Servers in End-to-End Performance of Dynamic Content Distribution," in *IMC*, pp. 559–568, ACM, 2011.

[13] A. Pathak, Y. A. Wang, C. Huang, A. Greenberg, Y. C. Hu, R. Kern, J. Li, and K. W. Ross, "Measuring and Evaluating TCP Splitting for Cloud Services," in *PAM*, pp. 41–50, Springer, 2010.

[14] "Azure regions." https://azure.microsoft.com/en-us/regions/.

[15] A. Flavel, P. Mani, D. A. Maltz, N. Holt, J. Liu, Y. Chen, and O. Surmachev, "Fastroute: A Scalable Load-aware Anycast Routing Architecture for Modern CDNs," in *NSDI*, vol. 27, p. 19, 2015.

[16] M. Calder, A. Flavel, E. Katz-Bassett, R. Mahajan, and J. Padhye, "Analyzing the Performance of an Anycast CDN," in *IMC*, pp. 531–537, ACM, 2015.

[17] "Google Cloud CDN." https://cloud.google.com/cdn/.

[18] "Google Cloud Load Balancer." https://cloud.google.com/load-balancing/.

[19] "Amazon CloudFront." https://aws.amazon.com/cloudfront/.

[20] "Amazon AWS Route53." https://aws.amazon.com/route53/.

[21] "Amazon Web Services." https://aws.amazon.com/.

[22] "Google Cloud Platform." https://cloud.google.com/.

[23] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a Globally-deployed Software Defined WAN," in *SIGCOMM*, pp. 3–14, ACM, 2013.

[24] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, "Gray Failure: The Achilles' Heel of Cloud-Scale Systems," in *HotOS*, 2017.

[25] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure," in *SIGCOMM*, pp. 58–72, ACM, 2016.

[26] "Akamai Compliance Management." https://www.akamai.com/uk/en/multimedia/documents/product-brief/akamai-for-compliance-management-feature-sheet.pdf.

[27] "Ripe atlas network coverage." https://atlas.ripe.net/results/maps/network-coverage/.

[28] "Dynatrace." https://www.dynatrace.com/capabilities/synthetic-monitoring/.

[29] "Thousandeyes." https://www.thousandeyes.com/.

[30] "Catchpoint." https://www.catchpoint.com.

[31] "Cedexis." https://www.cedexis.com/.

[32] F. Chen, R. K. Sitaraman, and M. Torres, "End-user Mapping: Next Generation Request Routing for Content Delivery," in *SIGCOMM*, vol. 45, pp. 167–181, ACM, 2015.

[33] C. Huang, N. Holt, A. Wang, A. G. Greenberg, J. Li, and K. W. Ross, "A DNS Reflection Method for Global Traffic Management.," in *USENIX ATC*, 2010.

[34] C. Huang, D. A. Maltz, J. Li, and A. Greenberg, "Public DNS System and Global Traffic Management," in *INFOCOM*, pp. 2615–2623, IEEE, 2011.

[35] M. H. Gunes and K. Sarac, "Analyzing Router Responsiveness to Active Measurement Probes," in *PAM*, pp. 23–32, Springer, 2009.

[36] R. A. Steenbergen, “A Practical Guide to (correctly) Troubleshooting with Traceroute,” *NANOG 37*, pp. 1–49, 2009.

[37] D. Choffnes and F. E. Bustamante, “Taming the Torrent: A Practical Approach to Reducing Cross-ISP Traffic in Peer-to-Peer Systems,” in *SIGCOMM*, 2008.

[38] 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 Internet’s Edge,” in *NSDI*, pp. 487–499, 2013.

[39] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, “Netalyzr: Illuminating the Edge Network,” in *IMC*, pp. 246–259, ACM, 2010.

[40] S. Sundaresan, S. Burnett, N. Feamster, and W. De Donato, “BISmark: A Testbed for Deploying Measurements and Applications in Broadband Access Networks.,” in *USENIX ATC*, pp. 383–394, 2014.

[41] M. Dhawan, J. Samuel, R. Teixeira, C. Kreibich, M. Allman, N. Weaver, and V. Paxson, “Fathom: A Browser-based Network Measurement Platform,” in *IMC*, pp. 73–86, ACM, 2012.

[42] T. Flach, P. Papageorge, A. Terzis, L. Pedrosa, Y. Cheng, T. Karim, E. Katz-Bassett, and R. Govindan, “An Internet-wide Analysis of Traffic Policing,” in *SIGCOMM*, pp. 468–482, ACM, 2016.

[43] Y. Chen, R. Mahajan, B. Sridharan, and Z.-L. Zhang, “A Provider-side View of Web Search Response Time,” in *SIGCOMM*, pp. 243–254, ACM, 2013.

[44] M. Andrews, B. Shepherd, A. Srinivasan, P. Winkler, and F. Zane, “Clustering and Server Selection Using Passive Monitoring,” in *INFOCOM*, vol. 3, pp. 1717–1725, IEEE, 2002.

[45] Z. M. Mao, C. D. Cranor, F. Douglis, M. Rabinovich, O. Spatscheck, and J. Wang, “A Precise and Efficient Evaluation of the Proximity Between Web Clients and Their Local DNS Servers.,” in *USENIX ATC*, pp. 229–242, 2002.

[46] A. Jain, J. Mann, Z. Wang, and A. Quach, “W3C Resource Timing Working Draft.” https://www.w3.org/TR/resource-timing-1/, July 2017.

[47] “USC CDN Coverage.” http://usc-nsl.github.io/cdn-coverage.

[48] C. Contavalli, W. van der Gaast, S. Leach, and E. Lewis, “RFC7871: Client Subnet in DNS Queries.” https://tools.ietf.org/html/rfc7871.

[49] M. Calder, X. Fan, Z. Hu, E. Katz-Bassett, J. Heidemann, and R. Govindan, “Mapping the Expansion of Google’s Serving Infrastructure,” in *IMC*, pp. 313–326, ACM, 2013.

[50] A. Flavel, P. Mani, D. Maltz, N. Holt, J. Liu, Y. Chen, and O. Surmachev, “FastRoute: A Scalable Load-Aware Anycast Routing Architecture for Modern CDNs,” in *NSDI ’15*, 2015.

[51] H. A. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. Van der Merwe, “Anycast CDNs Revisited,” in *WWW*, 2008.

[52] Z. Al-Qudah, S. Lee, M. Rabinovich, O. Spatscheck, and J. Van der Merwe, “Anycast-aware Transport for Content Delivery Networks,” in *WWW*, 2009.

[53] H. A. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. Van Der Merwe, “A Practical Architecture for an Anycast CDN,” *ACM Transactions on the Web (TWEB)*, 2011.

[54] L. Wei and J. Heidemann, “Does Anycast Hang up on You?,” in *TMA*, IEEE, 2017.

[55] G. Gürsun, “Routing-aware Partitioning of the Internet Address Space for Server Ranking in CDNs,” *Computer Communications*, vol. 106, pp. 86–99, 2017.

[56] J. S. Otto, M. A. Sánchez, J. P. Rula, and F. E. Bustamante, “Content Delivery and the Natural Evolution of DNS,” in *IMC*, 2012.

[57] J. S. Otto, M. A. Sánchez, J. P. Rula, T. Stein, and F. E. Bustamante, “namehelp: Intelligent Client-side DNS Resolution,” in *SIGCOMM*, pp. 287–288, ACM, 2012.

[58] “A Faster Internet.” http://www.afasterinternet.com/participants.htm.

[59] I. Poese, B. Frank, B. Ager, G. Smaragdakis, S. Uhlig, and A. Feldmann, “Improving Content Delivery with PaDIS,” *Internet Computing*, vol. 16, no. 3, pp. 46–52, 2012.

[60] B. Frank, I. Poese, Y. Lin, G. Smaragdakis, A. Feldmann, B. Maggs, J. Rake, S. Uhlig, and R. Weber, “Pushing CDN-ISP Collaboration to the Limit,” *CCR*, vol. 43, no. 3, pp. 34–44, 2013.

[61] B. Ager, W. Mühlbauer, G. Smaragdakis, and S. Uhlig, “Web Content Cartography,” in *IMC*, 2011.

[62] E. Nygren, R. K. Sitaraman, and J. Sun, “The Akamai Network: A Platform for High-performance Internet Applications,” in *SIGOPS*, pp. 2–19, ACM, 2010.

[63] M. J. Freedman, E. Freudenthal, and D. Mazieres, “Democratizing Content Publication with Coral,” in *NSDI*, vol. 4, pp. 18–18, 2004.

[64] J. Jiang, R. Das, G. Ananthanarayanan, P. Chou, V. Padmanabhan, V. Sekar, E. Dominique, M. Goliszewski, D. Kukoleca, R. Vafin, and H. Zhang, “Via: Improving internet telephony call quality using predictive relay selection,” in *SIGCOMM*, 2016.

[65] S. S. Krishnan and R. K. Sitaraman, “Video Stream Quality Impacts Viewer Behavior: Inferring Causality using Quasi-experimental Designs,” *Transactions on Networking*, vol. 21, no. 6, pp. 2001–2014, 2013.

[66] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec, “Peer-assisted Content Distribution in Akamai NetSession,” in *IMC*, pp. 31–42, ACM, 2013.

[67] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang, "C3: Internet-Scale Control Plane for Video Quality Optimization," in *NSDI*, vol. 15, pp. 131–144, 2015.

[68] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang, "CFA: A Practical Prediction System for Video QoE Optimization," in *NSDI*, pp. 137–150, 2016.

[69] Adnan Ahmed, Zubair Shafiq, Harkeerat Bedi, Amir Khakpour, "Peering vs. Transit: Performance Comparison of Peering and Transit Interconnections," in *ICNP*, 2017.

Appendices

A Measurement Counts

Let the number of measurements be n and the true failure rate be p. Analytically, the observed failure rate $\hat{p}$ is distributed as $Bin(n, p)/n$, so the average error is

$$E\left[|\hat{p} - p|\right] = E\left[\left|\frac{Bin(n,p)}{n} - p\right|\right].$$

Figure 12, however, is generated computationally via Monte Carlo simulations. For example, to find the value described in the caption, we simulated a large number (10^7) of draws from the binomial distribution

$$\hat{p} \sim Bin(n = 200, p = 0.01)/200,$$

then found the average value of $|\hat{p} - p| \approx 54\%$.

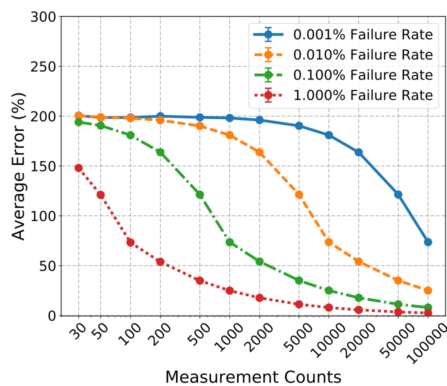


Figure 12: The average error of observed failure rate, as a function of number of measurements and true failure rate. For example, if the true failure rate of a service is 1.0% (red dotted line), then a sample of 200 measurements would yield an average error of about 50%, i.e., $1.0 \pm 0.5\%$.

Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure

Qiao Zhang[1], Guo Yu[2], Chuanxiong Guo[3], Yingnong Dang[4], Nick Swanson[4], Xinsheng Yang[4], Randolph Yao[4], Murali Chintalapati[4], Arvind Krishnamurthy[1], and Thomas Anderson[1]

[1]University of Washington, [2]Cornell University, [3]Toutiao (Bytedance), [4]Microsoft

Abstract

In Infrastructure as a Service (IaaS), virtual machines (VMs) use virtual hard disks (VHDs) provided by a remote storage service via the network. Due to separation of VMs and their VHDs, a new type of failure, called VHD failure, which may be caused by various components in the IaaS stack, becomes the dominating factor that reduces VM availability. The current state-of-the-art approaches fall short in localizing VHD failures because they only look at individual components.

In this paper, we designed and implemented a system called Deepview for VHD failure localization. Deepview composes a global picture of the system by connecting all the components together, using individual VHD failure events. It then uses a novel algorithm which integrates Lasso regression and hypothesis testing for accurate and timely failure localization.

We have deployed Deepview at Microsoft Azure, one of the largest IaaS providers. Deepview reduced the number of unclassified VHD failure events from tens of thousands to several hundreds. It unveiled new patterns including unplanned top-of-rack switch (ToR) reboots and storage gray failures. Deepview reduced the time-to-detection for incidents to under 10 minutes. Deepview further helped us quantify the implications of some key architectural decisions for the first time, including ToR switches as a single-point-of-failure and the compute-storage separation.

1 Introduction

Infrastructure-as-a-Service (IaaS) is one of the largest cloud services today. Customers rent virtual machines (VMs) hosted in large-scale datacenters, instead of managing their own physical servers. VMs are hosted in compute clusters, and mount OS and data VHDs (virtual hard disks) from remote storage clusters via datacenter networks. Resources can be scaled up and down elastically since compute and storage are separated by design.

Achieving high availability is arguably the most important goal for IaaS. Recently, large-scale system design [18, 33, 27], failure detection and mitigation techniques [23, 43, 25, 7, 6, 29, 36], and better engineering practices [10] have been applied to improve cloud system availability. Yet, attaining the gold standard of five-nines (99.999%) VM availability remains a challenge [32, 12].

At Microsoft Azure, there are on the order of thousands of VM down events daily. The biggest category of down events (52%) is what we call VHD failures. Due to compute-storage separation, when a VM cannot access its remote VHDs, the hypervisor has to crash the VM, resulting in a VHD failure. Those VHD failures are caused by various failures in the IaaS stack and constitute the biggest obstacle towards attaining five-nines availability for our IaaS [1].

Compute-storage separation brings unique challenges to locating VHD failures. First, it is hard to find the failing component in a timely fashion, among a large number of interconnected components across compute, storage, and network. The current practice of monitoring individual components is not sufficient. The complex dependencies and interactions among components in our IaaS mean that a single root cause can have multiple symptoms at different places. A network or storage failure may ripple through many other components and affect many VMs and applications. It becomes hard to distinguish causes from effects, resulting in a lengthy troubleshooting process as the incidents get ping-ponged among different teams.

Second, many component failures in the IaaS stack are gray in nature and hard to detect [27]. For failures such as intermittent packet drops and storage performance degradation, some VHD requests that pass through the component can fail but not others. The failure signals in these cases are weak and sporadic in time and space, making fast and accurate detection difficult.

[1]Azure has 34 regions and attains 99.9979% uptime in 2016. [41]

To address these challenges, we have designed and deployed a system called Deepview. Deepview takes a global view: it gathers VHD failure events as well as the VHD paths between the VMs and their storage as inputs, and constructs a model that connects the compute, storage, and network components together. We further introduce an algorithm which integrates Lasso regression [39] and hypothesis testing [14] for principled and accurate failure localization.

We implement the Deepview system for near-real-time VHD failure localization on top of a high-performance log analytics engine. To meet the near-real-time requirement, we add streaming support to the engine. Our implementation can run the Deepview failure localization algorithm in seconds, at the scale of thousands of compute and storage clusters, tens of thousands of network switches, and millions of servers and VMs.

Now in deployment at Azure, Deepview helped us identify many new VHD failure root causes which were previously unknown such as gray storage cluster failure and unplanned Top of Rack switch (ToR) reboot. With Deepview, unclassified VHD failure events dropped from several thousands per day to less than 500, and the Time to Detection (TTD) for incidents was reduced from tens of minutes and sometimes hours to under 10 minutes.

Contributions. We identified VHD failures as the biggest obstacle to five-nines VM availability for our IaaS cloud, and proposed a system to quickly detect and localize them. In particular, we

- Introduce a global-view-based algorithm that accurately localizes VHD failures, even for gray failures.
- Build and deploy a near-real-time system that localizes VHD failures in a timely manner.
- Quantify the implications of key IaaS architectural design decisions, including ToR as a single-point-of-failure and compute-storage separation (Section 7).

2 Background and Motivation

In this section, we first provide background on Azure's IaaS architecture. We explain how compute-storage separation can result in a new type of failure—VHD failures. Then, we introduce the state-of-the-art industry practice in localizing VHD failures and explain how it is slow and inaccurate. Finally, we motivate the approach Deepview takes and explain the challenges for putting the system into production uses.

2.1 Compute-Storage Separation in IaaS

Figure 1 shows Azure's IaaS architecture. A similar architecture seems to be used at Amazon for instances backed by the elastic block store [1]. Every VM has one OS VHD and one or more data VHDs attached. One key design decision is to separate compute and storage physically—**VMs and their VHDs are provisioned from different physical clusters**.

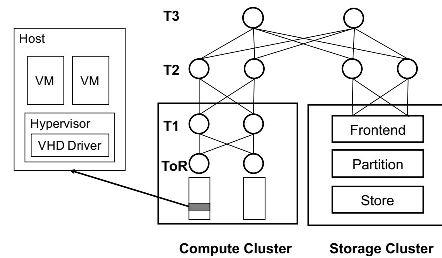


Figure 1: Azure's IaaS architecture. A region has tens to hundreds of compute/storage clusters. Each Tier2 (T2) switch connects some subset of clusters, while Tier3 (T3) switches connect the T2 switches. T3 switches are connected by inter-region network (not drawn).

The main benefit of this separation is to keep customer data available when their VMs become unavailable, e.g., due to a localized power failure. As a result, VM migration becomes easy as we only need to create a new VM (possibly on a different host or cluster) and attach the same VHDs.

In our datacenters, VHDs are provisioned and served from a highly available, distributed storage service [13, 21]. Azure's storage service is deployed in self-contained units of clusters with their own Clos-like network [5, 24, 13], software load balancers, frontend machines and disk/SSD-equipped servers. Similarly, VMs are hosted on physical servers grouped in what we call compute clusters. Each metro region typically has tens to hundreds of compute clusters and storage clusters, interconnected by a datacenter network.

Another benefit is load-balancing. A VM in a compute cluster can remotely mount VHDs from many different storage clusters. A compute cluster therefore uses VHDs from multiple storage clusters, and a storage cluster can serve many VMs from different compute clusters. As we will see later in section 3, this many-to-many relationship is leveraged by Deepview.

VHD Access is Remote. Compute-storage separation requires all VMs to access their VHDs over the network. When a VM accesses its disks, it is unaware that they are remotely mounted. The VHD driver in the host hypervisor provides the needed disk virtualization layer. The driver intercepts VM disk accesses, and turns them into VHD remote procedure call (RPC) requests to the remote storage service. The VHD requests and responses traverse over multiple system components (e.g., the VHD driver and the remote storage stack) and through multiple network hops (e.g., ToR/T1/T2/T3 switches).

VHD Failure	SW Failure	HW Failure	Unknown
52%	41%	6%	1%

Table 1: Breakdown of the causes of VM downtime. VHD failures cause the majority of VM downtime.

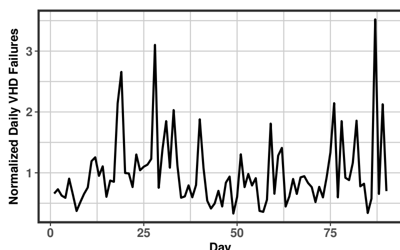


Figure 2: Daily VHD failures normalized the by 3-month average. Every day had at least one failure. On the worst day there were 3.5x more failures than average.

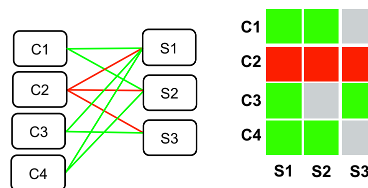
2.2 A New Type of Failure: VHD Failure

Compute-storage separation causes a new type of failure. In our datacenter, whenever VHD accesses are too slow to respond (the default timeout is 2 minutes), the hypervisor crashes the guest OS. In order to protect data integrity, when VHDs do not respond, the guest OS must be paused. But the pause cannot be indefinite—an unresponsive VM can cause customers to fail their own application level SLAs. After some wait, one reasonable option is to surface the underlying VHD access failure to the customer by crashing the guest OS. We call this **VHD failure caused by Compute-Storage Separation** or VHD failure for short.

VHD failure is the biggest cause of unplanned VM downtime. We analyzed an entire year's of IaaS VM down events, including their durations and causes (an internal team finds root causes for VM down events). Table 1 shows that 52% of VM downtime is due to VHD Failures, 41% due to Software Failures (data-plane software and host OS), and 6% due to Hardware Failures, and 1% due to unknown causes.

Figure 2 further shows the daily number of VHD failures normalized by the 3-month average across tens of regions. VHD failures happen daily. Occassionally, they are particularly numerous. The worst day over the 3-month period saw a 3.5x spike in volume.

To minimize the impact of VHD failures and improve VM availability, the most direct approach is to quickly localize and mitigate these failures. Next, we explain the prior VHD failure handling approach and its drawbacks.



(a) Bipartite model (b) Matrix view

Figure 3: The bipartite model and the corresponding matrix view of a downtime event.

2.3 State-of-the-Art: Component View

Our datacenter operators prioritize by the impact of each incident. A large rise in VHD failure events would automatically trigger incident tickets and set off an investigation.

The site reliability engineers (SREs) look at system components individually and locally, to see if any local component anomaly coincides in time with the VHD failure incident. The Compute team might look for missed heartbeats to see if the impacted physical machines have failed. The Storage team might look at performance counters to see if the storage system is experiencing an overload. The Network team might look at network latency and link packet discard rates to determine if some network devices/links could be at fault. Once the failure location is confirmed, the responsible team often has standard procedures for quick mitigation.

Prior to Deepview, failure localization was slow. It was common that we needed tens of minutes, sometimes more than one hour, to localize and mitigate big incidents, and hours to tens of hours to detect and localize gray failures. When a big incident happened, often more than one component had an anomaly because a single root cause could cascade to other services. For example, one big network incident caused as many as 363 related incidents from different services! As a result, the incident ticket could get ping-ponged among the teams.

Further, localization for gray failures [27] was often inaccurate and slow. For example, while we know ToR uplink packet discards can cause VHD requests to fail, it was unclear how severe the discard rate has to be. Setting a threshold to catch those failures became an art: too low generated too many false positives, while too high delayed diagnosis or missed the issue.

3 Our Approach: Global View

Our key insight is that rather than looking at the components individually and locally, we should take a global view. The intuition can be illustrated by the bipartite model in Figure 3a. In this model, we put compute clusters on the left side and storage clusters on the right. We

draw an edge from a compute cluster to a storage cluster if it has VMs that mount VHDs from the storage cluster. We also assign an edge weight equal to the fraction of VMs that have experienced VHD failures.

For a compute cluster issue such as an unplanned ToR reboot that causes all VMs under the ToR to crash regardless of what storage clusters they use, we see the edges (highlighted in red) from the impacted compute cluster with high VHD failure rates, as in Figure 3a. When a storage cluster fails, causing all VMs using that storage cluster to experience VHD failures, we see edges with high VHD failure rates coming to the impacted storage cluster.

If we put the compute clusters along the y-axis and the storage clusters along the x-axis, we get a matrix view as shown in Figure 3b. In this matrix view, a horizontal pattern points the incident to the computer cluster, while a vertical pattern points to the storage cluster.

Challenges. Though the bipartite model looks intuitive and promising, there are several challenges to use that insight in a production setting. First, since the bipartite model cannot be easily extended to model the multi-tier network layers, we cannot use it to diagnose failures in the network. Second, while we can use some voting/scoring heuristics to automate the visual pattern recognition, they work well only when the failures are fail-stop. For gray failures [27], fewer VMs would crash so the VHD failure signals are often weaker, and the VHD failure patterns are less clear cut. Third, when big incidents happen, many customers may feel the impact, making timely failure localization imperative. Our system must therefore operate in near-real-time.

Problem Statement. Our goal is to localize VHD failures for both fail-stop and gray failures to component failures in compute, storage or network, at the finest granularity possible (clusters, ToRs and network tiers), all within a TTD target of 15 minutes, in line with our availability objectives.

4 Deepview Algorithm

In this section, we explain how the Deepview algorithm solves the first two challenges—handling network and gray failures. We first describe our new model, a generalization of the bipartite model to include network devices. Then, we introduce our inference algorithm with two main techniques: 1) **Lasso regression** [39] to select a small subset of components as candidates to blame; 2) **hypothesis testing** [14] as a principled and interpretable way to handle strong and weak signals and decide on the components to flag to operators. There are other failure localization algorithms that can be adapted for our problem. We compare Deepview with them in Section 6.2,

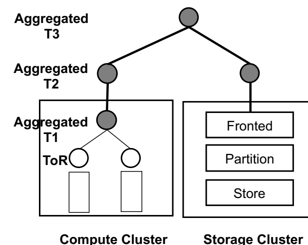


Figure 4: Transforming the Clos network to a tree. Not shown: each aggregated T2 switch connects to many compute/storage clusters and each aggregated T3 switch connects to many aggregated T2 switches.

and show that our approach has better recall and precision.

4.1 Model

In Section 3, we introduced a bipartite model that takes a global view of compute and storage clusters. Here we generalize the model to include network devices.

In this new model, we have three types of components: compute clusters, network devices and storage clusters. Figure 1 shows that compute clusters and storage clusters are interconnected by a number of Tier-2 (T2 for short) and Tier-3 (T3) switches in a Clos topology. ToR and Tier-1 (T1) switches are within the clusters, and are part of the clusters. To model the network, we replace each edge in the bipartite model with a path through the network that connects a compute cluster to a storage cluster. Here we describe the model at the level of clusters (which we call Cluster View in Section 6). We have also extended the model to the granularity of ToRs inside compute clusters (ToR View). For this work, we keep the storage cluster as a blackbox due to its complexity. As future work, we plan to apply our approach to the host level and the storage clusters internals.

4.1.1 Simplify the Clos Network to a Tree

One complication in modeling the network is that each compute/storage cluster pair is connected by many paths. Due to Equal-Cost Multi-path (ECMP) routing [24, 38], we do not know precisely which path a VHD request takes, and therefore, we do not know which path to blame when the request fails.

Our solution is to transform the Clos topology (Figure 1) to a tree topology (Figure 4) so that there is a unique shortest path between each cluster pair. We start from the bottom and go up for each cluster and aggregate the network devices by tiers, and then use shortest path routing to find the lowest overlap between each cluster pair.

The detailed procedure is as follows. First, we start with ToR switches in a cluster and find the T1 switches

they connect to. Then, we group those T1 switches as an aggregated T1 group for that cluster. Similarly, we can find the connected T2 switches for those T1 switches and group them as an aggregated T2 group for that cluster. We repeat this procedure to find the aggregated T3 groups. At the end of the aggregation, we have determined the aggregated T1, T2, T3 groups for each cluster in a region. The next step is to find the shortest path for each compute-storage pair . If the aggregated T2 groups of a cluster pair overlap, the midpoint is that overlapped aggregated T2 group; if their aggregated T2 groups do not overlap but their aggregated T3 groups do, the midpoint is the T3 group.

Due to the simplification, we cannot pinpoint to a specific network device, but only to within a network tier. In practice, Deepview is mainly used to decide which SRE teams to notify when VMs crash. Upon notification, network teams have other tools (e.g. Traceroute) to further narrow down to a device for mitigation.

4.1.2 From Paths to Components

Next, we use our observations of VHD failure occurrences to pinpoint which component has failed. We assume that components fail independently, which is a practical and reasonable approximation of the real world. For example, a compute cluster failure is unlikely to be correlated with a storage cluster failure. We can write down a simple probabilistic equation for a path consisting of compute, storage and network components:

$$\mathbb{P}(\text{path i is fine}) = \prod_{j \in \text{path}(i)} \mathbb{P}(\text{component j is fine}) \quad (1)$$

We approximate $1 - \mathbb{P}(\text{path i is fine})$ using the rate of VHD failures observed for that path:

$$\frac{n_i - e_i}{n_i} \approx \prod_{j \in \text{path}(i)} p_j \quad (2)$$

where n_i is the total number of VMs, e_i is the number of VMs that have VHD failures for a given time window, and p_j is the probability that component j is fine. We get a system of equations by writing down (2) for every path. Next, we infer the values of p_j for all components.

We know there is noise in our measurement, so we cannot directly solve the system of equations and would need to explicitly model the noise. Specifically, after taking log on both sides of equation (2) and adding a noise term ε_i, we get a set of linear models:

$$y_i = \sum_{j=1}^{N} \beta_j x_{ij} + \varepsilon_i, \qquad \varepsilon_i \overset{i.i.d.}{\sim} N(0, \sigma^2) \quad (3)$$

where $y_i = \log\left(\frac{n_i - e_i}{n_i}\right)$, $\beta_j = \log p_j$, and the binary variable $x_{ij} = 1$ iff i-th path goes through the j-th component.

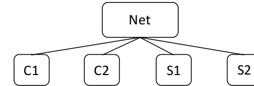


Figure 5: Example where multiple solutions may exist.

Interpretation of β_j: Once we get estimates for β_j, the probability that component j is fine can be computed from β_j because $p_j := \exp(\beta_j)$. If β_j is close to 0, we can clear component j from blame. Otherwise, if β_j is unusually negative, we have strong evidence to blame component j (see Section 4.3). We would ensure $\beta \leq 0$.

Next, we answer the following two questions: (1) how to get fast, accurate, and interpretable estimates for β_j; (2) given the estimates, how to decide which component to blame in a principled and interpretable manner?

4.2 Prefer Simpler Explanation

In practice, the number of unknown variables (β's) can be larger than the number of equations. We illustrate this in a simple example shown in Figure 5. We can list 4 equations with 5 free variables (the βs):

$$\begin{aligned} y_1 &= \beta_{c1} + \beta_{net} + \beta_{s1} + \varepsilon_1 \\ y_2 &= \beta_{c1} + \beta_{net} + \beta_{s2} + \varepsilon_2 \\ y_3 &= \beta_{c2} + \beta_{net} + \beta_{s1} + \varepsilon_3 \\ y_4 &= \beta_{c2} + \beta_{net} + \beta_{s2} + \varepsilon_4. \end{aligned} \quad (4)$$

Suppose all four paths saw equal probability of VHD failures. The blame can be pushed to the compute clusters C1 and C2, or the storage clusters S1 and S2, or the network, or a mix of those. Traditional least-square regression cannot give a solution in this case. But our experience tells us that multiple simultaneous failures are rare for a short window of time (e.g., 1 hour) because individual incidents are rare and failures are (mostly) independent. How do we encode this domain knowledge into our model to help us identify the most likely solution?

To prefer a small number of failures is mathematically equivalent to prefer the estimates $\beta = (\beta_1, \cdots, \beta_N)$ to be sparse (mostly zeros). We express this preference by imposing a constraint on model parameters β. By asking the sum of absolute values of β, i.e., $\|\beta\|_1$ to be small, we can force most of the components of β to zero, leaving only a small number of components of β remaining. This technique of adding a L1-norm constraint is known as Lasso [39], a computationally efficient technique widely used when sparse solutions are needed. We also ensure $\beta \leq 0$ to get valid probabilities. The estimate procedure that encodes all our beliefs in our model is thus the following convex program,

$$\hat{\beta} = \underset{\beta \in \mathbb{R}^N, \beta \leq \mathbf{0}}{\arg\min} \ \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda \|\beta\|_1. \quad (5)$$

Simplicity vs. Goodness-of-fit via λ: This loss function tries to strike a balance between goodness-of-fit in the first term (i.e., how well the model explains the observation) and sparsity in the second term (i.e., fewer failing components are more likely). The regularization parameter λ is the knob. Larger λ prefers fewer components to be blamed at potentially worse goodness-of-fit. The optimal value of λ is set by an automatic (data-adaptive) cross-validation procedure [26].

4.3 Decide Who To Blame

While big incidents are relatively easy to localize with a fixed threshold, it is much harder to find a threshold that can discriminate gray failures from normal components when there are random measurement errors. The estimated failure probabilities for gray failures can be very close to zero (see Section 6.1). The challenge then becomes how big an estimated failure probability is for a gray failure versus just measurement error. Setting such a threshold manually requires laborious data-fitting and is often based on some vague notions of anomaly. In practice, it can be difficult and fragile.

Can we use data to find the decision threshold in a principled and automatic way? Intuitively, the larger the magnitude a (negative) Lasso estimate has, the higher its estimated failure probability, and correspondingly more likely the component has failed. We had a painful experience manually tuning the threshold, but the process gave us some experience in distinguishing true failures (big incidents and gray failures) from measurement noise. We found that if a component's Lasso estimate is much worse than the average, then it is likely a real failure and should be flagged. The further from average, the more confident we are that the component has failed.

This decision can be automated in a hypothesis testing framework. Consider the following one-sided test:

$$H_0(j) : \beta_j = \bar{\beta} \quad \text{v.s.} \quad H_A(j) : \beta_j < \bar{\beta} \tag{6}$$

The null hypothesis $H_0(j)$ says the true probability that component j is fine is no different from the grand average of all components. We then use the data to tell us if we can reject $H_0(j)$ or not. If the data allow us to reject $H_0(j)$ in favor of the alternative hypothesis $H_A(j)$, then we can blame component j. Otherwise, we do not blame component j. The hypothesis test has three steps.

Step 1: Compute Test Statistic. Given Lasso estimates for components in a region, we find the mean $\bar{\hat{\beta}}$ and standard deviation $\sigma_{\hat{\beta}}$. Then we compute a modified Z-score for each component j,

$$z_j = \frac{\hat{\beta}_j - \bar{\hat{\beta}}}{\sigma_{\hat{\beta}} / \sqrt{N}}. \tag{7}$$

Under the assumptions that the measurement error is Gaussian, and other caveats,[2] we approximate the distribution of z_j as a Gaussian distribution with mean zero (under $H_0(j)$) and certain variance.

Step 2: Compute p-value. We then compute the p-value [14] for each component j. The p-value is the probability of seeing a failure probability for component j as extreme as currently observed simply by chance assuming that it is no different from the average. If the p-value is really small, then we do not believe the failure probability for component j is just about average. See the Appendix for more discussion on p-value.

Step 3: Make a Decision. Finally, we apply a standard threshold of 1% on p-value.[3] It expresses our tolerance for false positive rate. For example, if the p-value for component j is less 1%, we blame the component with at most 1% false positive rate. Otherwise, we have insufficient evidence to blame component j.

Avoid the Pitfalls in Multiple Testing. We test every component in a region and flag them based on p-values. For every test, we may falsely blame a normal component with a small chance. But with a large number of components in every region, we are bound to commit an actual false positive if not careful. This is called the multiple testing problem. We use the Benjamini-Hochberg procedure [9] to control the False Discovery Rate. See Appendix for details.

5 Deepview Design and Implementation

We have two main system requirements:

- **Near-real-time (NRT) processing**: VHD failures result in customer VM downtime, so failure localization must be speedy and accurate. We have the requirement that the time-to-detection (TTD) be within 15 minutes.
- **Speedy iteration**: VHD failures are the biggest obstacle to higher VM availability, so there is an immediate need by the operations team for better diagnosis. Our system is designed for quick iteration.

Our system requires two types of input data: non-real-time structural data and real-time event data. The former include the compute and storage clusters information, all the VMs and their VHD storage account information and related context, the paths for all the compute-storage pairs, and the network topology. Taking periodic snapshots of those every few hours suffices for our purposes.

[2] Testing on Lasso estimates is an active research area. We fit a Lasso model to obtain a set of nonzero variables, and refit these variables with least squares. See [46].

[3] Another common threshold is 5%, but it generates too many false positives for testing multiple hypotheses in our setting. See Appendix.

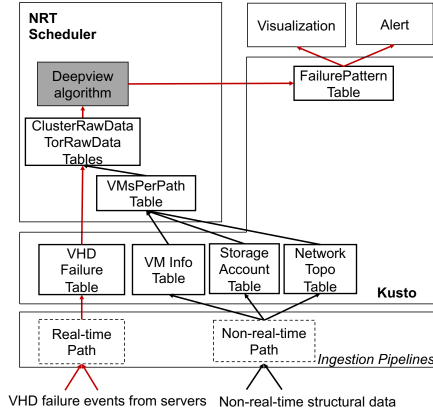


Figure 6: Deepview system architecture. Data schema is given in Table 2.

The latter are the VHD failure signals from servers. To meet near-real-time requirements, our algorithm needs to see VHD failure signals within minutes, ideally through a streaming system.

We need to scale to thousands of compute and storage clusters, tens of thousands of VHD failures per day, tens of thousands of network switches, hundreds of thousands paths, and millions of VMs.

The non-real-time information is either already in our in-house log analytics engine called Azure Kusto [3, 2], or can be generated and ingested into Kusto. Kusto stores data as tables but the tables are append-only, and it supports a SQL-like declarative query language. Kusto is backed by reliable persistent storage from a distributed storage service, using memory and SSD for a read-only cache. By default, it builds indices for all columns to improve query speed.

VHD failure events are generated by hypervisors. They are collected by a real-time pipeline. Since most of our data is already in Kusto, and Kusto provides highly expressive declarative language and fast data analysis, we ingest the VHD failure events into Kusto and build Deepview system on top of it.

System Architecture: The resulting system architecture is shown in Figure 6. It has four components. The real-time path and non-real-time path are for the input data ingestion for the Deepview algorithm. The Kusto platform provides both data analysis and storage for input, intermediate, and output data. The visualization and alert are tools for the consumption of Deepview results. The NRT scheduler is what we build on top of Kusto to support stream processing for the Deepview algorithm.

5.1 Stream Processing

We build our own stream processing system on top of Kusto because most of our data are already there; a few additions to satisfy our needs. We do not claim novelty compared to existing research and commercial streaming systems [44, 8, 4].

Table Name	Schema
VHDFailure	(ts, vm_id, vhd, str_account)
VMInfo	(ts, vm_id, comp_cluster, tor)
StorageAccount	(ts, str_account, str_cluster)
NetworkTopo	(ts, cluster, tor_list, t1_list, t2_list, t3_list)
VMsPerPath	(tstart, tend, num_vms)
ClusterRawData	(tstart, tend, comp_cluster, str_cluster, num_vms, num_failed_vms)
TorRawData	(tstart, tend, comp_cluser, tor, str_cluster, num_vms, num_failed_vms)
FailurePattern	(tstart, tend, region, type, loc, pval, visual_url)

Table 2: Kusto schemas for the Deepview data.

To support stream processing on top of Kusto tables, we use two abstractions:

- A computation directed-acyclic-graph (DAG) declared as a set of SQL-like queries with their output tables.
- A scheduler that runs each query at a given frequency.

We store the DAG and its scheduling policy as tables, since tables are Kusto's only supported data structure.

Computation DAG. The computation DAG consists of a set of queries that read from input tables and produce one or multiple output tables. The queries are the "edges" and the input/output tables are the "nodes". To maintain the DAG in Kusto, we give each query a name and store the query definition and the query output table name in yet another table.

NRT Scheduler. To provide a streaming window abstraction, we use a schedule to describe when each query in the DAG should be executed. The schedule describes how often it should run and how many times to retry. To meet availability requirements, we use a one-hour sliding window that moves forward every 5 minutes.

5.2 Algorithm Implementation

The algorithm implementation has three parts: first, construct the model—instantiate the design matrix x_{ij} and observation y_i based on the Deepview raw data tables, then run Lasso regression to infer β, and finally carry out hypothesis testing to pinpoint the failures.

Sparse Matrix and Region Filtering. The scale of our data poses some challenges for algorithm running time

and memory footprint. Constructing a full design matrix requires filling in entries for every path and every component with either zero or one. This can be slow and has high memory usage. However, x_{ij} are mostly zeros since each path has at most tens of components, so we only need to store the non-zero entries. Another simple technique is to only get data from Kusto for regions with non-zero VHD failure occurrences. Since simultaneous failures are rare, region filtering can avoid running the algorithm for some regions without hurting accuracy.

Coordinate Descent. Lasso regression has no closed form solution. Coordinate descent [22] is one of the fastest algorithms to solve the Lasso regression. We minimize the loss function as in Equation 5 with respect to each coordinate β_j while holding all others constant. We cycle among the coordinates until all coefficients stabilize. In practice, with warm start, we found that coordinate descent almost always converges in only a few rounds.

Cross-Validation with Warm Start to Set λ. We set the regularization parameter λ for Lasso using a data-adaptive method, i.e., cross-validation [26]. We use 5-fold cross validation where we split the data by paths into 5 partitions, and use any four of them to fit β for a given choice of λ and then compute the mean squared error (MSE) on the holdout partition using the fitted β. The optimal λ is the one that minimizes the average MSE. We speed up cross validation using a warm start technique [22]. Recall that a larger λ meant fewer non-zero β_j. We start with the smallest λ that turns off all β_j, and then we gradually decrease λ. Since β tends to change only slightly for a small change in λ, we reduce the number of rounds for coordinate descent by reusing $\beta(\lambda_{k-1})$ as the initial values for $\beta(\lambda_k)$.

6 Evaluation

We have deployed Deepview in production at Azure. Here, we first evaluate how well Deepview localizes VHD failures using production case studies. Then, we compare Deepview's accuracy with other algorithms. Next, we analyze various techniques proposed for Deepview and ask how useful each is. Finally, we evaluate how Deepview's runtime efficiency.

6.1 Deepview Case Studies

In this subsection, we ask how effective Deepview is at detecting and diagnosing incidents in production use.

6.1.1 Statistics

We examined the Deepview results for one month. The number of VHD failures generated per day can be up to tens of thousands. For this month, Deepview detected 100 patterns, and reduced the number of unclassified VHD failure events to less than 500 per day. We also tried to associate the detected patterns with incident tickets: 70 of the patterns were directly associated with incident tickets. The other 30 patterns were not associated with tickets. These 30 patterns turned out to be generated by weak VHD failure signals. They were all real underlying component failures that escaped the previous alerting system, either because of their smaller impact (e.g., unplanned ToR reboot) or their gray failure nature (e.g., gray storage failure).

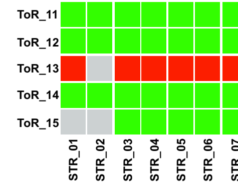


Figure 7: Deepview pattern for an unplanned ToR reboot.

Next we examine some of the representative patterns we found and discuss the insight we learn from them.

6.1.2 Unplanned ToR Reboot

From time to time, ToRs undergo scheduled downtime for firmware upgrade or other maintenance operations. Impacted customers are notified in advance, with their VMs safely migrated to other places. However, occasionally, a ToR may experience an unplanned reboot due to a hardware or software bug. Since each server connects to only one ToR, the VMs under the ToR will not be able to access their VHDs. We get VHD failures as a result. To detect unplanned ToR reboots, Deepview first estimates the failure probability and p-value for the ToR, and then checks the following conditions for confirmation: all the VMs under the ToR get VHD failures, the ToR OS boot time matches the failure time detected by Deepview, and the neighboring ToRs are working fine.

Figure 7 shows one such unplanned ToR reboot detected by Deepview in a small region.[4] It shows a portion of the Deepview UI, which we call ToR view. It clearly shows a horizontal pattern. The ToR switches in the compute cluster are listed on the y-axis and the storage clusters are listed on the x-axis. Each cell in the figure shows the status of the ToR and storage cluster pair. Gray means the VMs under the ToR do not use the corresponding storage cluster; green means the VMs do not have VHD failures; red means the VMs are experiencing VHD failures.

Deepview blamed the right ToR among 288 components in the region (ToRs, T1/T2/T3 switch groups and

[4]Readers may wonder how VHD failure events can be identified when the ToRs are single point of failure. They are in fact stored locally in the servers and are retrieved once network connectivity is restored (typically within 10 minutes for software failures).

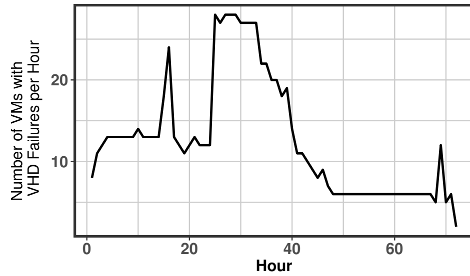


Figure 8: The number of VMs with VHD failures per hour during a storage cluster gray failure.

compute/storage clusters). Deepview estimated the failure probability for the failed ToR to be 100% with a p-value of 1.84E−64, which is much less than 0.01.

Deepview therefore makes it possible to study how often ToRs cause downtime. We discuss this in detail in Section 7.1.

6.1.3 Storage Cluster Gray Failure

Our storage cluster runs a full storage stack including load balancer/frontend, meta-data management, storage layer, etc. VHD failures can happen due to a variety of failure modes in the storage stack. When storage cluster failures are non-fail-stop, the VHD signals can be weak and noisy. For example, the load balancer could discard VHD requests to shed load, and in other cases, software bugs could cause some VHDs to become unavailable, impacting only a subset of VMs.

We next discuss such a storage gray failure case. A new storage cluster was brought online, but with a misconfiguration that allowed a test feature in the caching subsystem to be enabled. This bug mistakenly put some VHDs in negative cache (denoting deletion), rendering them "invisible" and unavailable for VM access.

Based on the VHD failure events at hour 0 in Figure 8, Deepview found three non-zero failure probability entities in the region, 0.34 for storage cluster S0, 0.002 and 0.047 for compute clusters C0 and C1. Notice that because this storage cluster failure only affected a small number of VMs, we did not get a failure probability of 1 for S0. Further, the two compute clusters saw non-zero failure probabilities because they also saw VHD failure events. However, despite the weak signal, our algorithm was able to correctly pinpoint the failure to S0. Our hypothesis testing procedure computed a p-value of 3.9E−34 for S0, identifying it as a failed cluster with very high confidence. On the other hand, C0 and C1 had p-values 0.51 and 0.54, respectively, and signifying a lack of evidence. Using our prior threshold method for detection, we would have delayed the detection by 22 hours. As shown in Figure 8, the signal is weak: the number of VMs affected per hour in the beginning was only around 10, and the peak number was only 28.

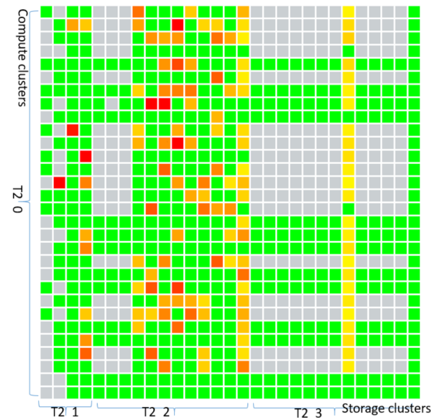


Figure 9: Deepview pattern for a network incident.

6.1.4 Network Failure

In our datacenter, switches other than ToRs have replicas. Single switch failures thus seldom lead to wide impact outages. However, in rare cases, a combination of capacity loss and traffic surge can cause network failures.

In one region, we have over 100 compute clusters and 50 storage clusters. They are connected by four T2 aggregated switches (numbered T2_0 to T2_3) with a T3 aggregated switch (T3_0) on top, as annotated along the axes in Figure 9. Each aggregated switch contains multiple switches. One day, a T3_0 switch underwent a major maintenance event, which triggered some T2 switches in T2_0 to mistakenly detect Frame Check Sequence (FCS) errors on the links to T3_0. Our automatic network service then kicked in and shut down most of links between the T3_0 switch and T2_0 except for three links saved by a built-in safety mechanism.

This loss in capacity together with a surge in storage replication traffic caused significant congestion between T2_0 and T3_0. As a consequence, we saw a significant increase in VHD failures experienced by customer VMs.

Figure 9 shows the pattern in the Deepview UI (partial Cluster View) with compute clusters on the y-axis and storage clusters on the x-axis. The switch aggregated per cluster is annotated on each axis. Yellow cells have a VHD failure rate at most 5%. The VHD failure rates are moderate because the VHD failures in this case were caused by network congestion; most of the time network connectivity was still working.

Deepview identified three aggregated switches with non-zero failure probabilities: 0.21%, 0.11%, 0.03% for T3_0, T2_0, and T2_2, respectively. Their corresponding p-values are 9.91E−12, 4.25E−04, 0.221. We point to T3_0 and T2_0 as the faulty network layers. The failure location is correct, as the root cause is the link conges-

tion between these two network layers. We note Deepview gave small failure probabilities because the VHD failure signals are weak: only a very small percentage of affected VMs crashed. But since T3_0 and T2_0 are high in the network hierarchy, they impact a large number of VMs.

We also experienced network incidents where network connectivity for many VMs were lost. They were easy for Deepview to detect and localize, as the signals were strong: many VMs died at the same time. We present this gray failure case to show the strength of Deepview.

To summarize, we have shown that Deepview can localize various incidents in which the signals can be weak or strong. Deepview has also deepened our understanding of VHD failures by identifying various patterns including horizontal patterns caused by incidents including unplanned ToR reboot, vertical patterns caused by storage outages, and network failure patterns.

6.2 Algorithm Comparison

Several algorithms that have been previously used to localize failures in the network can be extended to localize VHD failures. We compare with two tomography algorithms and a Bayesian network algorithm:

- **Boolean-Tomo [20, 19]:** Classify paths into good and bad paths based on a threshold (bad if at least γ VHD failures). Iteratively find the component on the largest number of unexplained bad paths, as the top suspect until all bad paths are explained. For the threshold γ, we tried $\gamma = 1,2,3,4,5$, and picked $\gamma = 1$ to maximize its recall and then precision.

- **SCORE [31]:** Classify paths into good and bad paths based on a threshold (γ). Iteratively compute for each component its hit ratio $\frac{numBadPaths(c)}{numPaths(c)}$ and coverage ratio $\frac{numUnexplainedBadPaths(c)}{totalNumUnexplainedBadPaths}$. Only consider components above a hit ratio threshold (η). Take the component with the highest coverage ratio as the top suspect. For the threshold γ and η, we tried $\gamma = 1,2,3,4,5$ and $\eta = 0.001, 0.01, 0.1$, and picked $\gamma = 1$ and $\eta = 0.01$ to maximize its recall and then precision.

- **Approximate Bayesian Network [35]:** The runtime to compute exact Bayesian network is exponential in the number of components, and thus is infeasible for us. We tried an approximation [35]. It uses mean-field variational inference to approximate the Bayesian network with a Noisy-OR model, and estimates the component j's failure rate as the posterior mean of a Beta distribution $B(\alpha_j, \beta_j)$. A component is blamed if $\frac{\hat{\alpha}_j}{\hat{\alpha}_j+\hat{\beta}_j}$ is above certain threshold. We do not include its accuracy numbers, because we are unable to make it give meaningful results on our data. The estimated posterior means of component failure rate allows us to apply a threshold. The computation takes 10 minutes for a single region, so this approach is not fast enough for our problem.

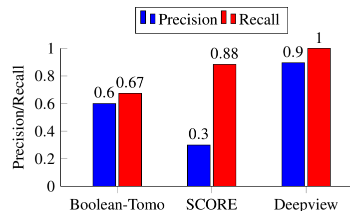


Figure 10: Precision/Recall comparison.

	Compute	Storage	Net	ToR
Precision	0.85	0.875	1.0	1.0
Recall	1.0	1.0	1.0	1.0

Table 3: Precision/Recall by failure type for Deepview.

Dataset. As we cannot run the other algorithms in production, we use trace data to compare algorithms. We had already hand-curated 42 incidents from a detailed study of tickets, so we use trace data from those incidents. They consist of 16 compute cluster issues (not ToR-related), 14 storage cluster issues, 10 unplanned ToR reboots, and 2 network issues. Only time periods when there is an incident are considered because a random sample is too sparse. Thus, we may overestimate the precision. But our comparison is fair since all algorithms use the same baseline ground truth.

Metrics. We compare each algorithm on recall and precision. Recall is the percentage of true failures that have been localized and precision is the percentage of localizations that are correct. In other words, high recall means we can localize most real failures, while high precision means we have few false positives.

Figure 10 summarizes the precision and recall for the 42 incidents. SCORE achieves a recall of 0.88, beating Boolean-Tomo, but it gives many false positives. Deepview, achieves both a high precision of 0.90 and a high recall of 1.0, beating both alternatives. Table 3 shows a breakdown of the precision and recall by failure types for Deepview. Overall, Deepview handles cases with a strong failure signal (Compute/ToR) and those with a weak failure signal (Storage/Network) well. Deepview also does well for unplanned ToR reboots and Network incidents. However, there were fewer of these incidents, so the estimates are to be taken with a grain of salt.

The other advantage of Deepview is that its parameters needs no manual tuning. Parameters are set by cross-validation (for λ) or using a standard interpretable criterion (false positive tolerance of 1% for p-value).

Boolean-Tomo and SCORE, instead, need careful tuning of their thresholds. In fact, we find that their precision and recall are sensitive to the thresholds. We picked those that maximize recall (as recall is typically more important than precision in production), while keeping precision as high as possible. We note that Deepview beats the performance of Boolean-Tomo and SCORE for all combinations of thresholds (omitted for lack of space).

6.3 Deepview Algorithm Analysis

We have introduced a set of techniques for our algorithm. Here, we analyze how useful each technique is.

Cross-validation and λ in Lasso Regression. The regularization parameter λ is set by cross-validation for each region. The optimal values found for incidents in Section 6.2 span three orders of magnitude with a minimum of 0.00012 and a maximum of 0.48. In fact, it is well known in statistical literature that choosing a universally optimal λ for all problems is impossible. The theoretical optimal [11] depends on the number of paths, the number of components, the structure of the network, and the error variance (i.e., how stable are VHD failures among different paths). When cross-validation is fast, it is preferred to a manual threshold.

Hypothesis Testing and Gray Failures. We use hypothesis testing to find a decision threshold to localize both big incidents and gray failures in the presence of random noise. The gray failure case studies in Section 6.1 show that hypothesis testing is essential. For the storage case, the failure probabilities are 0.34 for the truly failed storage cluster S0, and 0.002 and 0.047 for two normal compute clusters. Their p-values 3.9E−34 and 0.51 and 0.54 are needed to accentuate the difference and allow us to pick only S0. Similarly, for the network case, looking at p-values allow us to filter out T2_2.

6.4 Deepview Running Time

Algorithm Running Time. We measure the running time for Deepview algorithm in production. The worst-case running time is 18.3 seconds on a single server. It includes the time to read input data from Kusto, execute the algorithm and write the output data to Kusto.

Time to Detection (TTD) TTD is defined as the time between when an incident happens and when the failure is localized. The average time from a VHD failure event to its appearance in Kusto is 3.5 minutes. Adding the 5 minutes windowing time and the processing time, Kusto achieves a TTD under 10 minutes. This is a significant improvement over the previous TTD which typically lasted from tens of minutes to hours.

7 Discussion

Several architectural decisions were made when our IaaS was built. One is that a server connects to only a single ToR via a single NIC. While this makes ToR a single-point-of-failure (SPOF), the decision dramatically reduces networking cost. Another decision is that a VM can host its VHDs in any storage cluster in the same region. This makes load-balancing for storage clusters easy, but with potentially higher network latency and lower throughput. Further, both decisions may adversely impact VM availability. Using the data collected from Deepview, we can now study the impact of these decisions quantitatively.

7.1 ToR as a Single-Point-of-Failure

As we have described in Section 6.1, Deepview can detect unplanned ToR reboots. From the failure patterns, we find that there are two types of ToR failures: soft failures and hard failures. Soft failures can be recovered by rebooting the ToR, while hard failures cannot.

Our data shows that: (1) less than 0.1% switches experience unplanned reboots in a month; (2) 90% of the failures are soft failures, with the rest hard failures. The hard failure rate agrees with our ToR Return Merchandise Authorization (RMA) rate, which indicates that 0.1% switches need to be RMAed in one year. These numbers are obtained from a fleet of tens of thousands of ToRs.

The impact of a soft failure typically lasts for less than 20 minutes: 10 minutes for the ToRs to come up and 10 minutes for the VMs to recover. The impact of a hard failure lasts longer as the failed switch needs to be physically replaced. The impact to VMs can be shorter though as the VMs can be migrated to other hosts due to the separation of compute and storage. We conservatively use 2 hours as the impact period for hard failures.

If the ToR is the only failure source for VMs on that rack, the availability of our IaaS is no better than

$$1 - \frac{0.9 \times 20 + 0.1 \times 120}{1000 \times 30 \times 24 \times 60} = 99.99993\%$$

Even with ToR as the single point of failure, the service can achieve six-nines. This meets the rule of thumb that critical dependencies need to offer one additional 9 relative to the target service [40].

Thanks to Deepview data, for the first time, we are able to show that ToR as a single point of failure is an acceptable design choice for IaaS as it is not on the critical path for five-nines availability.

Note that simply examining ToR logs would not have given us these numbers, as many ToR reboots are planned, with no impact on VM availability.

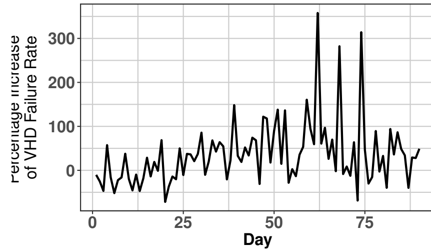


Figure 11: Daily percentage increase in VHD failure rate for VMs crossing T3 and above compared to those that only cross T2 for a 3-month period.

7.2 Co-locate or Disaggregate?

A VM can use VHDs from any storage cluster in the same region, due to the separation of compute and storage. We look at the network distance between VMs and the storage clusters for their VHDs. We find that some 51.8% of VHD paths go through T2, 41.0% need to go through T3 and the rest go above T3 in Azure.

A longer network path may result in higher network latency and packet drop rate. However, it is not clear whether it will also negatively affect VM availability.

Here we use the Deepview data to answer this quantitatively. We look at our data for three months. For each day, we first compute the VHD failure rates r_0 and r_1 for VMs crossing T2 only and VMs crossing T3 and above, respectively. Then, we find the percentage increase $(r_1 - r_0)/r_0$.

Figure 11 shows the daily percentage increase over a 3-month period. VMs whose network paths cross T3 network layer or above see a higher VHD failure rate than those that only need to cross T2 on most days. There is a 11.4% increase $((\overline{r_1} - \overline{r_0})/\overline{r_0})$ in the VHD failure rate if the VHD access needs to cross T3 or above.

One possible explanation is that as VHD requests go up the network tiers, they traverse more switches which may become oversubscribed. Thus VHD requests may become more likely to fail when network path lengths get longer. An implication of this study is that there is some benefit to colocating VMs and their VHDs in nearby clusters for availability.

8 Related Work

Machine Learning. Machine learning techniques have been used for failure localization, such as decision trees [6, 17], Naive Bayes [45], SVM [42], correlations [43], clustering [16], and outlier detection [36]. They allow domain knowledge to be encoded as features, but in general require a rich set of signals to discriminate different failure cases and may rely on assumptions about traffic that are not generally applicable. The most relevant work is NetPoirot [6], which targets a similar scenario as ours, but with a very different approach. NetPoirot is a single node solution where end-hosts independently run pre-trained classification models on local TCP statistics to infer failure locations. We believe NetPoirot and Deepview are complementary—TCP metrics from IaaS VMs may provide a useful signal to Deepview.

Tomography. There has been a large body work in network tomography (see [15] for a survey), and specifically binary tomography and its variants [20, 19, 31] for network failure localization. Typically, greedy heuristics are used to select among multiple solutions that all explain the observations. Various thresholds are often needed to tradeoff between precision and recall ratios. Compared with those approaches, Deepview avoids manual threshold tuning and achieves both higher recall and precision as shown in section 6.2.

Bayesian Network. Bayesian network [34] is a principled probabilistic approach to failure localization. It can model complex system behaviors [7] and handle measurement errors [28]. While exact inference is intractable [30], there are various approximation techniques such as using noisy-or to simplify conditional probability calculation [35, 7, 37], considering *k*-subset root-causes to shortcut marginalization [28, 7], using a simple factored form for joint posterior [35], or using message passing for faster inference [37]. For our problem, we find that using a combination of approximation techniques (we tried two [35]) was essential. It is future work to compare Deepview with some practical Bayesian network approach.

9 Conclusion

We identified VHD failures caused by compute-storage-separation as the main factor that reduces VM availability at our IaaS cloud. We introduced Deepview, a system that quickly localizes failures from a global view of different system components and a novel algorithm integrating Lasso regression and hypothesis testing. Data from production allowed us to quantitatively evaluate precision and recall across many failure events. We also used Deepview data to evaluate the impact of system architecture on VM availability.

Acknowledgement

We thank our Azure colleagues Brent Jensen, Girish Bablani, Dongming Bi, Rituparna Paul, Abhishek Mishra, Dong Xiang for their valuable discussions and support. We thank our MSR colleagues Pu Zhang, Myeongjae Jeon and Lidong Zhou, and intern Jin Ze for their contributions to an early prototype of Deepview. We thank our shepherd Mike Freedman and the anonymous reviewers for their feedback. This work was partially supported by the NSF (CNS-1616774).

References

[1] Amazon EC2 Root Device Volume. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html#RootDeviceStorageConcepts.

[2] Azure Kusto (Preview). https://docs.microsoft.com/en-us/connectors/kusto/.

[3] Introducing Application Insights Analytics. https://blogs.msdn.microsoft.com/bharry/2016/03/28/introducing-application-analytics/.

[4] ABADI, D. J., CARNEY, D., ETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. B. Aurora: a New Model and Architecture for Data Stream Management. *The VLDB Journal 12* (2003), 120–139.

[5] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM* (2008).

[6] ARZANI, B., CIRACI, S., LOO, B. T., SCHUSTER, A., AND OUTHRED, G. Taking the Blame Game Out of Data Centers Operations with NetPoirot. In *SIGCOMM* (2016).

[7] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *SIGCOMM* (2007).

[8] BALAZINSKA, M., BALAKRISHNAN, H., MADDEN, S., AND STONEBRAKER, M. Fault-tolerance in the Borealis Distributed Stream Processing System. In *SIGMOD Conference* (2005).

[9] BENJAMINI, Y., AND HOCHBERG, Y. Controlling the False Discovery Rate: a Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* (1995), 289–300.

[10] BEYER, B., JONES, C., PETOFF, J., AND MURPHY, N. R. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[11] BICKEL, P. J., RITOV, Y., AND TSYBAKOV, A. B. Simultaneous Analysis of Lasso and Dantzig Selector. *The Annals of Statistics* (2009), 1705–1732.

[12] BISHOP, T. Microsoft Says Google's Cloud Reliability Claim vs. Azure and Amazon Web Services Does Not Compute, 2017. https://www.geekwire.com/2017/microsoft-says-googles-cloud-reliability-claim-vs-azure-amazon-web-services-not-compute.

[13] CALDER, B., ET AL. Windows Azure Storage: a Highly Available Cloud Storage Service with Strong Consistency. In *SOSP* (2011).

[14] CASELLA, G., AND BERGER, R. L. *Statistical Inference*, vol. 2. Duxbury Pacific Grove, CA, 2002.

[15] CASTRO, R., COATES, M., LIANG, G., NOWAK, R., AND YU, B. Network Tomography: Recent Developments.

[16] CHEN, M. Y. ., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN* (2002).

[17] CHEN, M. Y., ZHENG, A. X., LLOYD, J., JORDAN, M. I., AND BREWER, E. A. Failure Diagnosis Using Decision Trees. In *ICAC* (2004).

[18] DEAN, J. Designs, Lessons and Advice From Building Large Distributed Systems. *Keynote from LADIS 1* (2009).

[19] DHAMDHERE, A., TEIXEIRA, R., DOVROLIS, C., AND DIOT, C. NetDiagnoser: Troubleshooting Network Unreachabilities Using End-to-end Probes and Routing Data. In *CoNEXT* (2007).

[20] DUFFIELD, N. Network Tomography of Binary Network Performance Characteristics. *IEEE Transactions on Information Theory 52*.

[21] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in Globally Distributed Storage Systems. In *OSDI* (2010).

[22] FRIEDMAN, J., HASTIE, T., AND TIBSHIRANI, R. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of statistical software 33*, 1 (2010), 1.

[23] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or Die: High-Availability Design Principles Drawn From Googles Network Infrastructure. In *SIGCOMM* (2016).

[24] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM* (2009).

[25] GUO, C., ET AL. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM* (2015).

[26] HASTIE, T. J., TIBSHIRANI, R., AND FRIEDMAN, J. H. The elements of statistical learning: data mining, inference, and prediction, 2nd Edition. In *Springer series in statistics* (2009).

[27] HUANG, P., GUO, C., ZHOU, L., LORCH, J. R., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *HotOS* (2017).

[28] KANDULA, S., KATABI, D., AND VASSEUR, J.-P. Shrink: a Tool for Failure Diagnosis in IP Networks. In *MineNet* (2005).

[29] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed Diagnosis in Enterprise Networks. In *SIGCOMM* (2009).

[30] KOLLER, D., AND FRIEDMAN, N. Probabilistic Graphical Models - Principles and Techniques.

[31] KOMPELLA, R. R., YATES, J., GREENBERG, A., AND SNOEREN, A. C. IP Fault Localization via Risk Modeling. In *NSDI* (2005).

[32] LEOPOLD, G. AWS Rates Highest on Cloud Reliability, 2015. https://www.enterprisetech.com/2015/01/06/aws-rates-highest-cloud-reliability.

[33] MOGUL, J. C., ISAACS, R., AND WELCH, B. Thinking About Availability in Large Service Infrastructures. In *HotOS* (2017).

[34] PEARL, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. 1988.

[35] PLATT, J. C., KICIMAN, E., AND MALTZ, D. A. Fast Variational Inference for Large-scale Internet Diagnosis. In *NIPS* (2007).

[36] ROY, A., ZENG, H., BAGGA, J., AND SNOEREN, A. C. Passive Realtime Datacenter Fault Detection and Localization. In *NSDI* (2017).

[37] STEINDER, M., AND SETHI, A. S. End-to-end Service Failure Diagnosis using Belief Networks. In *NOMS* (2002).

[38] THALER, D., AND HOPPS, C. Multipath Issues in Unicast and Multicast Next-Hop Selection, 2000. IETF RFC 2991.

[39] TIBSHIRANI, R. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), 267–288.

[40] TREYNOR, B., DAHLIN, M., RAU, V., AND BEYER, B. The Calculus of Service Availability. *ACM Queue 15* (2017).

[41] VISWAV, P. Microsoft Dismisses Google's Cloud Reliability Claim, 2017. https://mspoweruser.com/microsoft-dismisses-googles-cloud-reliability-claim.

[42] WIDANAPATHIRANA, C., LI, J. C., SEKERCIOGLU, Y. A., IVANOVICH, M. V., AND FITZPATRICK, P. G. Intelligent Automated Diagnosis of Client Device Bottlenecks in Private Clouds. *2011 Fourth IEEE International Conference on Utility and Cloud Computing* (2011), 261–266.

[43] YU, M., GREENBERG, A. G., MALTZ, D. A., REXFORD, J., YUAN, L., KANDULA, S., AND KIM, C. Profiling Network Performance for Multi-tier Data Center Applications. In *NSDI* (2011).

[44] ZAHARIA, M., DAS, T., LI, H., HUNTER, T., SHENKER, S., AND STOICA, I. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP* (2013).

[45] ZHANG, S., COHEN, I., GOLDSZMIDT, M., SYMONS, J., AND FOX, A. Ensembles of Models for Automated Diagnosis of System Performance Problems. In *DSN* (2005).

[46] ZHAO, S., SHOJAIE, A., AND WITTEN, D. In Defense of the Indefensible: A Very Naive Approach to High-Dimensional Inference. *arXiv preprint arXiv:1705.05543* (2017).

A P-value Correction for Multiple Testing

To decide if a component has failed, we could make a decision based on a threshold for the estimated failure probability for that component. But we can make a more principled decision by conducting a hypothesis test for each component as specified in (6). In this appendix, we explain the details in doing this testing. We explain our p-value, and motivate and explain how we do multiple testing.

A.1 Interpretation of p-values

To conduct the test for each component, we construct the test statistics as in (7) for each of the N components. We then compute the p-value for each test to decide whether to reject the null hypothesis. The p-value is defined as, the probability, assuming the null hypothesis is true, of the sampling test statistic having a value at least as extreme as observed. If the null hypothesis is true, we should expect a moderate p-value. However, if the computed p-value is small, we have evidence to believe that the null hypothesis is false. In fact, when the p-value is too small (e.g., below the conventional 1%, 5%, and 10% significance level), we should reject the null hypothesis $H_0(j)$, since it is highly unlikely that it can explain the values we have observed.

If the p-value is greater than the significance level, then the test is inconclusive. However, we give extra attention to borderline cases to decrease the false negative rate. For example, we produce warnings with lower priority for those components whose p-values are only slightly greater than the significance level.

A.2 Choice of Significance Level

How to choose an appropriate significance level? For testing a single hypothesis, conventional choices of significance level include 1%, 5%, and 10%.

However, when testing multiple hypotheses, we need to be more careful about false positives. Suppose we are testing 100 null hypotheses, all of which are true. If we use 5% as the significance level, then there is roughly 5% probability that we incorrectly reject the null hypothesis—committing a false positive. Further, if these 100 tests are independent, then we are almost certain to make at least one false positive:

$$\begin{aligned}\mathbb{P}(\text{at least one false positive}) &= 1 - \mathbb{P}(\text{no false positive}) \\ &= 1 - 0.95^{100} = 0.994.\end{aligned} \tag{8}$$

Intuitively, the more hypotheses we test simultaneously, the more likely we are to make a mistake.

To reduce the tendency of making mistakes when testing multiple hypotheses, we need to provide a stricter significance level than a single test. This is called the multiple testing correction.

A.3 Multiple Testing Correction

There are two approaches to multiple testing correction: family-wise error rate (FWER) control correction or false discovery rate (FDR) control correction. We use FDR control in Deepview algorithm since it is the more powerful alternative.

Let V be the number of false positives (the healthy components that we falsely blame), and R be the number of rejected hypotheses (the total number of components we blame). Then the false discovery rate (FDR) is defined as

$$FDR := \mathrm{E}[Q] := \mathrm{E}[V/R] \tag{9}$$

The Benjamini-Hochberg procedure [9] is the most popular FDR control procedure due to its simplicity and effectiveness. The procedure is as follows:

1. Do N individual tests and get their p-values P_1, $P_2, \dots, P_N$ corresponding to null hypothesis $H_0(1)$, $H_0(2), \dots, H_0(N)$.
2. Sort these p-values in ascending order and denote them by $P_{(1)}, P_{(2)}, \cdots, P_{(N)}$.
3. For a given threshold on FDR α, find the largest K such that $P_{(K)} \leq \frac{K}{N}\alpha$.
4. Reject all null hypotheses for which their p-values are smaller than or equal to $P_{(K)}$.

This procedure controls the FDR under α.

LiveTag: Sensing Human-Object Interaction Through Passive Chipless WiFi Tags

Chuhan Gao and Yilong Li
University of Wisconsin-Madison
{cgao57, yli758}@wisc.edu

Xinyu Zhang
University of California San Diego
xyzhang@ucsd.edu

Abstract

Many types of human activities involve interaction with passive objects. Thus, by wirelessly sensing human interaction with them, one can infer activities at a fine resolution, enabling a new wave of ubiquitous computing applications. In this paper, we propose LiveTag to achieve this vision. LiveTag is a fully passive, thin metal tag that can be printed on paper-like substrates and attached on objects. It has no batteries, silicon chips or discrete electronic components. But when touched by fingers, it disturbs ambient WiFi channel in a deterministic way. Multiple metallic structures can be printed on the same tag to create unique touch points. Further, LiveTag incorporates customized multi-antenna beamforming algorithms that allow WiFi receivers to sense the tag and discriminate the touch events, amid multipath reflections/interferences. Our prototypes of LiveTag have verified its feasibility and performance. We have further applied LiveTag to real-world usage scenarios to showcase its effectiveness in sensing human-object interaction.

1. Introduction

Information about the objects a person touches is an essential input to many applications in ubiquitous computing. On one hand, the ability to sense touch in the physical world can form the basis of the tangible user interface [17], which allows human to use omnipresent objects as a command-and-control interface to the digital world. On the other hand, the sequence of objects used can enable inference of human activities [32, 47]. Logs of objects touched can become the basis of "experience sampling" [3] or "life-logging" [11, 16, 52] that try to reconstruct a user's day. Post-processing of the logs can support many activity-aware applications, such as stroke rehabilitation assessment in homes, consumer analytics for retail stores [12, 28], *etc.*

To harvest these benefits, a practical system needs to sense touches on different objects, and on different spots of the same object. The system should be inexpensive for ubiquitous deployment, and should be unobtrusive—always functioning but without distracting users and with little maintenance cost. In addition, it should preserve privacy, capturing nothing more than the user's interest. These salient properties will embody Mark Weiser's vision of ubiquitous computing by weaving the system into physical environment, rendering the underlying technology invisible [51]. Although many conventional sensors can detect object use (*e.g.*, motion sensors [43, 47] and cameras [9, 15, 54]), they often require augmenting the objects with batteries/circuits, or may provoke strong visual privacy concerns.

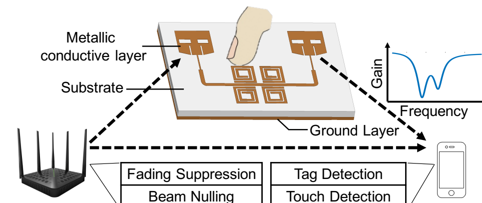


***Figure* 1:** Overview of LiveTag.

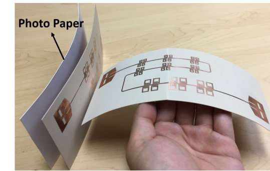


***Figure* 2:** Printed thin, flexible LiveTag tags, in comparison with a piece of photo paper.

In this paper, we propose LiveTag, a new wireless sensing modality to detect manipulation with physical objects. Fig. 1 illustrates the working principle. LiveTag uses thin radio-frequency (RF) tags as a user interface, either attached on the objects, or working independently as a thin keypad or control panel. These tags are fully passive, chipless, and battery-free, only made of a layer of metal foil printed on a thin substrate (*e.g.*, flexible ceramic-PTEF laminate commonly used for thin PCB printing, as shown in Fig. 2). Touch commands on the tags are detected remotely by WiFi devices which can react accordingly. More specifically, the tag is designed as a strong *reflector* for 2.4/5 GHz signals from a cooperating WiFi *transmitter*, and touches upon its metal structure create a known non-linear channel distortion which can be remotely detected by a WiFi *receiver*.

To satisfy the targeted use cases of LiveTag, the tags need to be sensitive to WiFi signals, contain multiple distinguishable touch points, and bear identifiable characteristics. In meeting these requirements, we create RF surface capacitors/inductors/resistors by printing metal structures with special geometries. These surface electronic components eventually form a *resonator* that absorbs WiFi signals of specific frequency, acting like a bandstop filter to create a "notch" on the WiFi channel response. Multiple resonators can be co-located on the same tag with different notch positions. Together, these resonators create a *spectrum signature* that makes the entire tag uniquely distinguishable from others. In addition,

finger touch on each resonator nullifies the notch, resulting in a unique change in the WiFi channel response.

To realize these salient properties, we empirically model the relation between tag geometry and corresponding frequency characteristics, as well as the impact of touching. These models allow us to make tradeoffs between the tag size, capacity (*i.e.*, number of touch points), and number of coexisting tags. We have also fabricated the tags using thin PCB laminates. Under controlled setup where the signals directly pass through the tag, we observe up to 35 dB of attenuation at the desired notch points.

In practical over-the-air usage scenarios, however, detecting the tags and touch events entails a number of challenges unseen in conventional communication systems or actively modulated RFID systems. *First*, the line-of-sight (LOS) channel between the WiFi transmitter and receiver is much stronger and can easily overwhelm the signals reflected by the tag. *Second*, due to frequency-selective fading caused by ambient multipath reflections, the spectrum signatures tend to be interfered by random channel gain variations across the frequency band. To tackle such uncertainties, we design redundancies into the tag and combine multiple resonators to enhance the spectrum signature. To enable robust detection, we design a fading suppression and LOS nulling mechanism, taking advantage of the multiple antennas on the WiFi transceivers. The touch event is then detected as a known change in the spectrum, following a stochastic model that guarantees a prescribed false alarm rate.

We have verified these solutions through an enhanced tag design, as well as a tag detection system comprised of a pair of WiFi-compatible transceivers. Our experiments in practical indoor scenarios demonstrate that both the presence of and touch upon a multi-resonator tag can be detected accurately, even when the tag is placed 4.8 *m* away from the transmitter. The miss detection rate (P_m) and false alarm rate (P_f) is only around 3%, and approach 0 with multi-resonator redundancy. The tag-to-receiver distance needs to be shorter (around 0.5 *m*). So a user-carried WiFi device like smartphone is mostly suitable as a LiveTag receiver. In terms of tag capacity, each tag is able to provide up to 8 touch points, created by 8 frequency notches that span the entire WiFi band. We have also conducted 3 case studies of LiveTag involving human-object interaction: a batteryless keypad, on-clothes music controller, and water-level detector attached to a cup, which demonstrate LiveTag's capabilities in augmenting everyday life in a non-intrusive way.

In summary, the main contributions of LiveTag include:

(i) Tag design. Although the concept of passive RF tags has existed for long, existing designs mainly focused on manipulating the tag signatures to embed more information, and they need dedicated ultra-wideband, full-duplex readers. To our knowledge, LiveTag represents the first system to design touch-sensitive, WiFi-detectable tags that can sense human-object interaction.

(ii) Tag detection. We design new beamforming mechanisms to suppress ambient multipath and LOS interference, enabling a pair of WiFi Tx/Rx to detect the passive tag and multipoint touch events in practical environment.

(iii) Implementation and experimental validation. We implement the tags using standard PCB printing technique (which allows mass production and tag customization). Our experiments verify LiveTag's feasibility and accuracy, and its usefulness in enabling new sensing applications that involve human-object interaction.

2. Related Work

Sensing interaction with objects. Existing techniques for detecting object manipulation either monitor the objects directly or augment/modify the objects using sensors. The former is represented by computer vision solutions that extract humans-object relations from images [9,15,54]. Whereas image-features lead to high detection accuracy in controlled settings, practical systems have proved very difficult to engineer, especially under unknown background, moving scenes, and challenging light conditions. The computational cost is also high and unsuitable for real-time touch-command applications. Barcode [30] may reduce the feature processing time, but requires scanning in line-of-sight with a handheld device.

On the other hand, active sensors [39,43,47], while extremely accurate, carry circuit components and need battery maintenance, rendering them unsuitable for scaling to a large number of low-value objects. RFID can overcome such limitation by attaching energy-harvesting tags on objects. Early research embedded an RFID reader into a glove to sense interaction with tagged objects [41]. Recent work augmented RFID tags with low-power sensors that live on the energy harvested from interrogating signals [27, 40, 46]. IDSense [25] can discriminate touch and movement of an RFID tag, by learning the RSS/phase features. RIO [35] detects gestures by recognizing phase changes caused by finger contact. PaperID [24] creates an ungrounded monopole antenna, which can respond to the reader only upon human touch (and hence grounding). These RFID solutions require an expensive, dedicated reader, and cannot distinguish different touch positions on one tag.

Overall, LiveTag can be considered a blended technology that inherits the advantages of aforementioned two categories. It augments the objects with lightweight, WiFi-readable tags that have no silicon chips, batteries, or discrete circuit components. LiveTag also overcomes all the aforementioned limitations of video/image processing, enabling ubiquitous, real-time sensing even in low-light and non-line-of-sight (NLOS) conditions.

Chipless RFID tags. LiveTag is inspired by the chipless RFID tags [31]—passive reflectors made from sur-

face metallic structures with identifiable electromagnetic properties. Chipless RFID is motivated by the vision of bringing RF tags' cost to a level comparable to visible barcode [10, 38]. Chipless tags encode information either in time or frequency domain. Time-domain approaches use multiple RF circulators to induce different delays to passing signals, thus creating signatures. Frequency-domain approaches create signatures on the tag's frequency response using multiple RF filters with different stopband frequencies. Existing literature in chipless RFID primarily focused on improving the tag capacity, *i.e.*, number of bits encoded. Since it is extremely challenging to create narrowband surface filters, embedding multiple filters on the tag requires huge spectrum bandwidth. High-end ultra-wide-band (UWB) readers (on the 3.1-10.6 GHz band) [37] have to be used, which are costly and can only achieve sub-meter range [37] due to the FCC's transmission power regulation on the UWB band. State-of-the-art research in chipless RFID [37, 45] mostly uses dedicated radios or network analyzers as readers, and omits ambient multipath reflections considering the short tag-reader distance.

LiveTag differs from conventional chipless RFID tags in two fundamental ways: *(i)* It aims to make the tags responsive to touch, rather than increase the tag capacity. *(ii)* Through customized tag design and detection algorithms, it can repurpose commodity WiFi transceivers as readers, which work even in practical multipath environment. Recent advances in backscatter communications have enabled a new species of radios that communicate by modulating ambient RF signals [18–20], and can harvest RF energy to power touch sensors [26]. These backscatter radios build on discrete circuit components. In contrast, LiveTag is a low-profile, fully-passive, paper-like substrate that obviates discrete circuit componnets. It can potentially be mass produced through conductive inkjet printing at extremely low cost.

3. Designing Touch-Sensitive Passive Tags

The 3D structure of a LiveTag tag is illustrated in Fig. 1. When the interrogation signal reaches the tag, it is first received by one antenna, and then passed through a *transmission line* and filtered by a *multi-resonator* network. The resulting signal is eventually emitted through the other antenna. The signal path is bidirectional—Each antenna simultaneously receives interrogating signals and propagates them towards the opposite direction. Ultimately, the entire tag acts as a reflector that backscatters the interrogating signals. The key design goal of the tag is to maximize the change of spectrum upon touch, by optimizing each resonator's *filter gain*, defined as the ratio between incidental and emitting signal strength. Below we describe how LiveTag approaches this objective.

3.1 Resonator Model

The resonator is essentially a 2D bandstop filter printed on a planar substrate. Such an RF filter can be realized using a variety of geometrical structures (Fig. 3), all with similar working mechanisms [31, 36]. At resonance frequency, microwaves form standing waves in the resonator, oscillating with large amplitudes, thus confining the energy within the resonator. The resonator can be modeled by an equivalent circuit (Fig. 4), comprised of a cascade of capacitor C_r, inductor L_r, and resistor R_r, whose values are determined by the resonator's material and geometry [22, 23]. When placed next to a transmission line with impedance Z_L, the resonator is coupled through parallel-line coupling or equivalently mutual inductance coupling [5]. The coupled circuit can be modeled as a grounded cascade RLC (Fig. 4) structure [44]. The equivalent impedance can be straightforwardly formulated as a function of the angular frequency ω:

$$Z_R(\omega) = R_r + j\omega L'_r - j\frac{1}{\omega C_r} = R_r + j\omega L'_r\Big(\frac{\omega^2 - \omega_c^2}{\omega^2}\Big), \quad (1)$$

where the $\omega_c = 1/\sqrt{L'_r C_r}$. The impedance $|Z_R(\omega)|$ reaches its minimum at $\omega = \omega_c$. So ω_c is the *resonance frequency* of the resonator, *i.e.*, the center frequency of its stopband. The frequency response of the above circuit model can be formulated as,

$$20\log\big(\frac{V_{out}}{V_{in}}\big) = 20\log\big(\frac{|Z_R(\omega)|}{|Z_R(\omega)| + Z_L}\big). \quad (2)$$

Obviously, the frequency response reaches its minimum at the resonance frequency ω_c, when $|Z_R(\omega)|$ reaches its minimum value R_r. We remark that this circuit model is highly simplified, and an accurate characterization can be obtained only via electromagnetic simulation [44].

3.2 Single Resonator Design

Ideal resonators for LiveTag should meet 3 requirements. *(i)* A resonator's frequency response should not be affected by other adjacent ones (*i.e.*, minimum mutual coupling). *(ii)* The filter bandwidth of the resonator must be narrow, so that we can pack multiple stopbands into the limited WiFi spectrum. *(iii)* The filter gain at resonance frequency should be large, allowing the tag presence and touch events to be easily detectable. Fig. 1 provides an example of the frequency response that consists of two notches, which correspond to two stopbands.

To support multi-touch, multiple resonators must be co-located, but with no mutual coupling. Prior research [36, 37] has shown that the interference between planar resonators becomes negligible if they are coupled to a common transmission line in a non-contact manner. To satisfy this condition, LiveTag adopts the spiral and L-shaped resonators (Fig. 3), whose resonating frequency is independent of their relative positions along the transmission line [36]. On the other hand, *the frequency response of each resonator is determined by three factors*: the substrate, material of the conductive layer, and the resonator's geometry, which we elaborate on below.

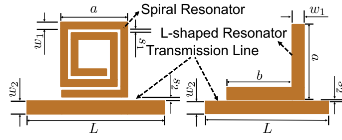


***Figure* 3:** Spiral and L-shaped resonators.

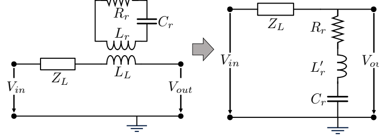


***Figure* 4:** Circuit model of a single resonator.

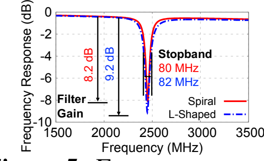


***Figure* 5:** Frequency response curves of resonator structures.

Material of substrate and conductive layer. LiveTag adopts either the woven glass (FR-4) or ceramic-PTEF flexible PCB as substrate. The thickness and dielectric constant of substrate affect the center frequency of the stopbands. Therefore, given a new substrate, the resonator structure should be adjusted to keep center frequency to the desired value. The resonator, transmission line and antennas are realized by printing a thin copper layer onto the substrate. The thickness and electrical resistivity of the copper material affect the conductivity, and hence the loss of EM waves propagating through. However, it does not affect the resonating frequency in a noticeable way, since the conductivity of printable metal materials (*e.g.*, copper and silver) is sufficiently high.

To isolate the tag from other objects behind, we ground the tag by printing a thin copper layer on the back of the substrate. The conductive *ground layer* shields the EM wave from external materials behind the tag.

Impact of the resonator's geometrical parameters. To identify the geometry that approaches the desired frequency response, conventional planar antenna design often undergoes multiple iterations of empirical design and validation [31]. Following this common practice, we employ a mix of simulation and empirical models to design the geometry of our tag. We use Advanced Design System (ADS) [1], an RF electronic design automation tool, to simulate the frequency response of a conductive layer, using its geometry, substrate thickness H and dielectric constant ε_r as input.

In designing RF systems, the *impedances* of series connected systems should match in order to maximize the power transfer from input to output. Specific to LiveTag, the impedance of the multi-resonator structure must match that of the antenna (designed as 50Ω following the common practice). The impedance of microstrip transmission line follows a well-known model [34]:

$$Z = \frac{87}{\sqrt{\varepsilon_r + 1.41}} \cdot \ln\left(\frac{5.98H}{0.8w_2 + T}\right), \quad (3)$$

where w_2 and T are the width and thickness of the transmission line. ε_r, H and T are fixed and known at fabrication time. Therefore, for impedance matching, we only need to compute w_2 so that $Z = 50\Omega$. The line length (L) only affects the phase of the EM wave, and does not impact the spectrum signature which only concerns the magnitude of the CSI. Therefore, *we can flexibly extend or twist the transmission line, depending on the specific outline needed by the touch interface.*

On the other hand, the *impact of the resonator's geometry* can be characterized through ADS simulation. Our simulation focuses on 3 key metrics: center frequency, bandwidth of the stopband, and filter gain. Although such metrics have been partly studied in simulating planar RF filters [33,34], a comprehensive quantitative study is still critical to make LiveTag work in the WiFi band. To this end, we first empirically configure the resonator geometry so that a stop-filtering effect appears on the 2.4/5 GHz spectrum. We then fine-tune the geometrical parameters to optimize the resonator performance. Under default settings, Fig. 5 depicts the simulated frequency response of the spiral and L-shaped resonators.

***(i)** Resonator size a:* Our simulation results in Fig. 6(a) indicate that *a larger resonator leads to lower resonance frequencies*, because it increases the wavelength of the standing wave. Equivalently, both the L'_r and C'_r increase in the circuit model (Fig. 4), which leads to a smaller ω_c. In addition, *the L-shaped resonator is generally much larger ($a = 15$ mm) than spiral resonator ($a = 7$ mm) when operating at the same 2.4 GHz band*. On the other hand, the equivalent RLC bandstop filter (Fig. 4) has a 3 dB bandwidth of approximately $\frac{R_r}{L'_r}$ [34], so a *larger a (and hence larger L'_r) decreases the bandwidth of the notch*. Fig. 6 (b) further shows that *the resonator size does not affect the filter gain significantly*.

***(ii)** Gap between resonator and transmission line s_2:* Intuitively, the properties of the stopband, *i.e.*, center frequency and notch bandwidth, only depend on the resonator itself. Our simulation results in Fig. 7(a) indeed corroborate this. On the other hand, *as s_2 increases*, the coupling between the resonator and the transmission line weakens, resulting in less signal energy being passed to the resonator, and hence *a sharp reduction in the filter gain*, as shown in Fig. 7 (b).

Under the same simulation setup, we also found that the other parameters, w_1, s_1, N_t and b (Fig. 3), have negligible impacts on the frequency response. Since the frequency response is primarily determined by the patterns of micro strip lines, we find slightly bending the tag does not affect the tag's response or tag/touch detection. We omit the details for the sake of space. To summarize the foregoing exploration, *the resonance frequency can be controlled by adjusting the resonator size, while the gap between resonator and transmission line should always be kept as small as possible to achieve a high filter gain and small bandwidth occupation.* Note that alternative

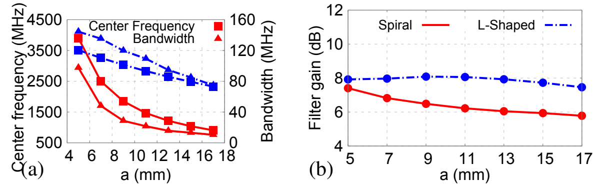


***Figure* 6:** Impact of resonator size a on: (a) center frequency and bandwidth, (b) filter gain.

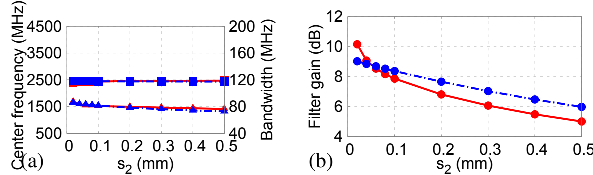


***Figure* 7:** Impact of gap between resonator and transmission line s_2 on: (a) center frequency and bandwidth, (b) filter gain.

RF resonator structures may further improve the performance, but are out of this paper's scope.

3.3 Resonator Responses to Touch

The human body can be modeled as an RC network consisting of resistors and capacitors [8]. So a finger touching the resonator can be approximated as adding a capacitor and a resistor in parallel to the original resonator's components, and then to the ground. This leads to an equivalent increase of C_r, thus making the notch's center frequency disappear from its original position.

We quantitatively verify the touch impact under a controlled setup, where the two ends of a transmission line are connected to an Agilent E8364A (50 GHz) network analyzer via SMA adapters. This isolates the multipath effects and measures the tag's intrinsic frequency response. Fig. 8 shows that *the notch indeed disappears from its original position after the touch*, likely due to the overwhelming attenuation effect that neutralizes the filter gain. We also found that touching part of the resonator has the same impact as touching its whole body. The same figure also plots the simulated frequency response before touch, which matches the measurement well.

3.4 Tag Antennas

LiveTag adopts two types of planar antennas—patch and monopole—whose front-side structures are shown in Fig. 9, whereas the back side is a metal foil acting as the ground. The design principles of such 2D antenna structures are well established [7, 29]. Both antennas have been designed to have high gain on both 2.4 GHz and 5 GHz WiFi bands. The monopole antenna has a close to omni-directional radiation pattern, while patch antenna provides higher directionality. The different gain patterns (Fig. 10) imply that, *the patch antenna is suitable for scenarios where less than half-space need to be covered, whereas the monopole fits mobile tags with sporadic pointing directions.*

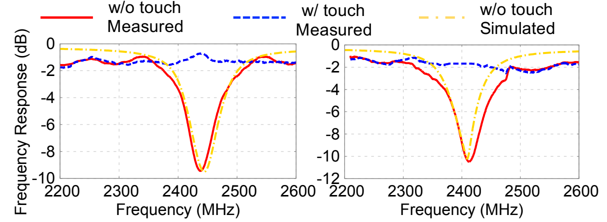


***Figure* 8:** Impact of finger touch on spiral (left) and L-shaped (right) resonator.

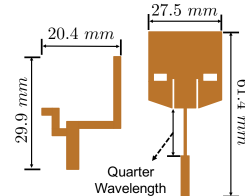


***Figure* 9:** Monopole (left) and patch (right) antenna.

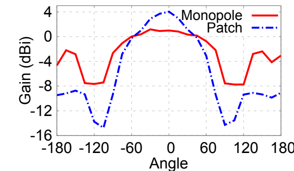


***Figure* 10:** Radiation pattern of tag antennas.

4. Creating Multiple Touch Points and Tags

4.1 Embedding Multiple Resonators in a Tag

Multiple resonators: creating spectrum signatures and enhancing filter gain. To create multiple touch points, we extend the single-resonator design (Sec. 3.2) by placing multiple resonators with different resonance frequencies along side the transmission line. In addition, we place multiple identical resonators with the same resonance frequency close to each other, to form a *compound touch point*. This is equivalent to connecting multiple identical bandstop filters in series, which increases the filter gain multiplicatively. Fig. 11 plots the filter gain of tags with different number of spiral resonators, measured using the network analyzer. We observe that the *filter gain increases linearly (in dB scale) with the number of redundant resonators*. In practice, we can simply use the central area among these resonators as the touch point, so that a single touch detunes them simultaneously.

To profile the mutli-resonator structure's sensitivity, we use the network analyzer to measure the *difference* of its frequency response before and after touch. We use an actual tag with 5 pairs of resonators, creating 5 different notches at 5170 MHz, 5305 MHz, 5515 MHz, 5665 MHz, and 5800 MHz. The measurement results (Fig. 13) verify that touching each pair creates 6-9.5 dB of filter gain change on the resonating frequency. We also design a tag with 6 identical resonators. Our measurement result in Fig. 14 shows that the total filter gain decreases dramatically with the number of resontors being simultaneously touched.

Coupling between resonators. The coupling effect occurs when signals backscattered from a resonator generate resonant current in an adjacent resonator through inductive coupling, which may distort the frequency response. Fortunately, coupling happens only in the near-field when the resonators are placed in close proximity.

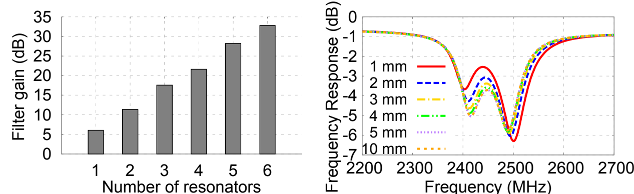


Figure **11:** Filter gain with redundant resonators.

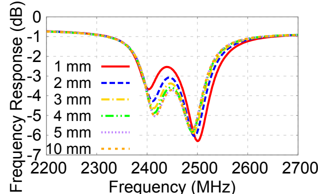


Figure **12:** Impact of resonator separation distance.

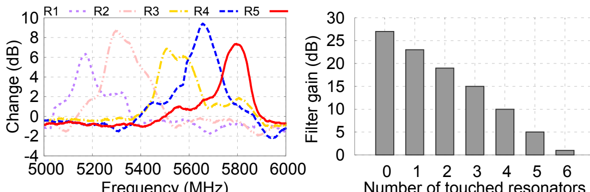


Figure **13:** Change of frequency response after touching each pair of resonators.

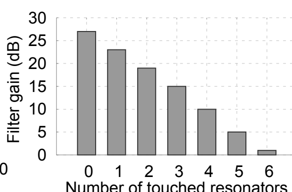


Figure **14:** Filter gain when touching different number of identical resonators.

To quantify the impact, we simulate the frequency response of a tag under the same default parameter configuration as in Sec. 3.2, except that two resonators are used with center frequencies 2415 MHz and 2489 MHz. Fig. 12 shows the spectrum signature as the resonators' separation varies. The center frequencies of the two notches drift away from the original values as the resonators are placed closer than 5 *mm*. But, as long as the separation exceeds 5 *mm*, the drifting effect becomes negligible. A similar experiment shows that the minimum separation is 4 mm for resonators at 5 GHz.

4.2 Tag Capacity and Multi-Tag Coexistence

Tag capacity. The number of resonators that can be packed into the tag depends on two factors.

(i) Area capacity, which is constrained by physical size of the tag and each resonator. Each 5 GHz spiral resonator occupies $3.5 \times 3.5\,mm^2$, and needs a minimum separation of 4 *mm* from other resonators and negligible separation from the transmission line. For impedance matching, the transmission line width needs to be 3 *mm*. Since the resonators can be placed on both sides of the transmission line, the maximum 1D occupation of one resonator is approximately $3.5+3/2+4/2 = 7$ *mm*, *i.e.*, it only occupies $7 \times 7\,mm^2$.

(ii) Frequency capacity, which is constrained by the usable spectrum width, and the bandwidth of the notch created by each resonator. At the 2.4 GHz and 5 GHz unlicensed band, the available spectrum is around 85 MHz and 480 MHz [4], whereas the stopband bandwidth created by the resonator is around 80 MHz (Fig. 7) and 180 MHz (Fig. 13), respectively. To pack more notches into the spectrum, we design the resonators such that the adjacent notches have an overlap of half of the notch bandwidth. In addition, only the center of each notch needs to fall inside the WiFi spectrum band. This enables us to pack up to 3 notches and 6 notches, in the 2.4 GHz and 5 GHz band, respectively.

Multi-tag coexistence. When multiple tags coexist, the mutual interference is negligible as long as one tag is always much closer to the WiFi receiver than the others. This is the case when the WiFi receiver is a mobile device (*e.g.*, smartphone) that always accompanies the user that touches the tags. Otherwise, when multiple tags exist in close proximity, they have to use orthogonal spectrum signatures, in the same way as placing distinct resonators on the same tag. We will quantify the impact of tag separation in Sec. 7.

5. Detecting Touches on Tag with WiFi

Once a tag is deployed, LiveTag executes three mechanisms to detect and discriminate the touch events. First, the WiFi transmitter continuously runs a joint beamforming and beam nulling algorithm to suppress multipath fading, making the tag-induced channel features more pronounced. Second, the WiFi receiver continuously measures the CSI, and detects the presence of a tag with known spectrum signature using a maximum likelihood algorithm. Once a tag is identified, the WiFi receiver continues to detect touch based on the pattern of CSI changes. Below we describe these mechanisms in detail.

5.1 Extracting Frequency Response of a Tag

LiveTag's transmitter beamforming mechanism facilitates over-the-air estimation of a tag's frequency response, and isolates it from the LOS or ambient multipath signals. Unlike conventional beamforming, the key challenge lies in the fully passive tag, which cannot process incoming signals or estimate its own channel response.

Creating artificial fading with orthogonal beamforming. To isolate the ambient multipath, our key idea is to use multiple transmit antennas to create artificial fast fading effects, by generating multiple transmit beam patterns with minimum correlation. Different beams may encounter different ambient reflectors, resulting in diverse paths and destructive/constructive effects across different frequencies. From the perspective of the tag, although the beams may come from different angles, the resonator will cause the same notch position on the CSI (measured by the WiFi receiver). Thus, the WiFi receiver can smooth out the fading effect by taking advantage of the CSI diversity of all these beams.

The question is: given a certain number of antennas, how many, and which beam directions should be used? Since the beamwidth depends on the number of antennas and cannot be arbitrarily small, increasing the number of beams blindly would result in overlapped beam patterns and hence correlated channel. To minimize the correlation of for a fixed number of AP antennas, LiveTag steers the beams' main lobe directions to be equally spaced,

and uses a delay-sum beamformer [49] for beam steering. Without loss of generality, consider a linear antenna array with half-wavelength separation between elements. Then each beam is mirror symmetric relative to the array dimension, so we only consider the angle range from $-90°$ to $90°$, where $0°$ is the direction perpendicular to the antenna array. To generate K beam directions, LiveTag sets the ith main lobe direction to be $\frac{180°}{K}i - 90°$, where $i = 0, 1, \cdots, K-1$. To harness the benefits of beamforming with minimal beam correlation, we always set K to be equal to the number of AP antennas.

It's worth noting that since the reflection and fading experienced by each beam is determined by the environment, our beamforming technique does not provide hard performance guarantee on fading suppression, although our empirical evaluation shows it works well even under environment dynamics (Sec. 7.1.1).

Smoothing CSI with PCA. In order to extract the correlated tag response among CSI with uncorrelated fading, we apply principal component analysis (PCA) on the CSI. Then the K components after PCA is ordered by the amount of information contained, or variance. We find that preserving only the 2nd and 3rd component tends to generate the best results. The first component is discarded because it turns out to contain strong correlated noises, which is most likely contributed by the common fading notches that affect multiple beams' CSI.

Simultaneously suppressing the LOS channel. While suppressing the ambient multipaths, LiveTag needs to simultaneously suppress the LOS channel from the AP to the client. Conventional MIMO beam nulling techniques, commonly used to suppress certain receivers [2], are not directly applicable for LiveTag—since the WiFi receiver can only estimate the compound channel, a direct beam nulling will suppress the tag-to-receiver channel as well. To isolate the LOS channel, LiveTag adopts a two-stage beamforming method. In the first stage, the WiFi client device transmits a packet, while the AP processes the CSI and estimates the Angle of Arrival (AoA) profile using the classical MUSIC algorithm [42]. In the second stage, the AP nullifies the angle with the strongest signal strength (most likely to be the LOS angle), denoted as θ_j. Suppose θ_i is the beam direction used for the fading-suppression. For a linear array with N half-wavelength spaced antennas, to create a beam with main lobe steered towards angle θ, the weight vector applied to all antennas should follow [49]:

$$\boldsymbol{a}(\theta) = [1 \;\; e^{j\pi\sin(\theta)} \;\; e^{j2\pi\sin(\theta)} \cdots \; e^{j(N-1))\pi\sin(\theta)}]^T \qquad (4)$$

To steer the beam towards θ_i, the weight vector $\boldsymbol{a}(\theta_i)$ should be applied. Meanwhile, to nullify the signals towards angle θ_j, the weight vectors can be obtained by

$$\boldsymbol{a}_{null}(\theta_i, \theta_j) = \boldsymbol{P}_{\theta_j}^{\perp} \cdot \boldsymbol{a}(\theta_i), \qquad (5)$$

where $\boldsymbol{P}_{\theta_j}^{\perp}$ is the operator that projects the original beamforming weights onto the subspace that is orthogonal to the subspace spanned by the LOS direction θ_j. Following the definition, $\boldsymbol{P}_{\theta_j}^{\perp}$ can be computed as,

$$\boldsymbol{P}_{\theta_j}^{\perp} = \boldsymbol{I} - \frac{\boldsymbol{a}(\theta_j)\boldsymbol{a}^H(\theta_j)}{\boldsymbol{a}^H(\theta_j)\boldsymbol{a}(\theta_j)}, \qquad (6)$$

where $\boldsymbol{a}(\theta_j)\boldsymbol{a}^H(\theta_j)$ projects a vector onto its own subspace. Therefore, $\boldsymbol{a}_{null}(\theta_i, \theta_j)$ can be reorganized as,

$$\boldsymbol{a}_{null}(\theta_i, \theta_j) = \boldsymbol{a}(\theta_i) - \boldsymbol{a}(\theta_j) \cdot \frac{\boldsymbol{a}^H(\theta_j)\boldsymbol{a}(\theta_i)}{\boldsymbol{a}^H(\theta_j)\boldsymbol{a}(\theta_j)}. \qquad (7)$$

Note that the LOS path between the AP and client might be blocked, and another multipath reflection may become the strongest. The beam nulling mechanism will suppress this path in the same way as the LOS, and enhance detection performance except in the rare case when the tag reflection becomes strongest. We will empirically verify LiveTag in practical environment in Sec. 7.

It should be noted that *LiveTag does not require the tag or ambient environment to be static*, because such dynamics do not affect the frequency response of the tag, which solely depends on the resonators and touch event. In fact, just like the artificial fading effects, such environment dynamics randomize the irrelevant fading, making the tag's spectrum signatures more prominent.

WiFi channel stitching and antenna calibration. To effectively use the available spectrum, the WiFi AP in LiveTag interrogates the tag by switching across 3 (11) non-overlapping channels on the 2.4 GHz (5 GHz) band. The measured CSI of each channel is stitched together to obtain the frequency response of the tag. In addition, the resonators' frequency notches are designed intentionally to avoid the DC and guard-band subcarriers which carry no CSI. Note that the AP and client can switch across multiple channels with coarse synchronization [50]. Also, since the antenna does not have uniform gain across 2.4/5 GHz band (Sec. 3.4), we always normalize the frequency response curve by the gain of the antenna at corresponding frequencies. The antenna gain can be measured as a one-time initialization step.

Coexisting with communications. Similar to all active WiFi sensing applications, LiveTag needs to coexist with data transmissions, which can be achieved in two ways. Since LiveTag only needs the CSI of each WiFi channel as input, we can piggyback the sensing onto communication. Each WiFi packet has a known preamble which enables the receiver to extract the CSI. An alternative approach is to treat sensing as normal data communication, and have our system contend for channel access in the same way as normal data communication in 802.11. This will not burden the network, as our system only needs to send an extremely short frame (without payload) on each channel.

5.2 Tag Presence Detection and Identification

LiveTag identifies the presence of a tag by comparing the measured CSI with the ground-truth spectrum signature of each deployed tag. The ground-truth can be

obtained and stored either at design time, or through a one-time initial measurement. Tags in different environments (*e.g.*, home vs. office) can have the same signatures, as long as certain coarse context/location information is available to distinguish them.

To compare the measured/ground-truth response, the simplest way is curve matching. However, classical curve matching methods, like nearest neighbor based on Euclidean distance, or dynamic time warping [21], either involve high computational complexity or are vulnerable to noise from irrelevant features. To alleviate such limitations, we design a model based on signal space representation (SSR) of the spectrum signature. Instead of using the entire frequency response curve as a tag's feature, we first reduce the feature dimension and only preserve the points that set different tags apart. Suppose there are B tags deployed in the given environment, whose frequency responses can be stacked to form a matrix

$$\boldsymbol{H_S} = \begin{bmatrix} \boldsymbol{h_1} & \boldsymbol{h_2} & \cdots & \boldsymbol{h_B} \end{bmatrix}, \tag{8}$$

where $\boldsymbol{h}_i, i = 1 \cdots B$, represents the spectrum signature of the ith tag, containing C frequency domain sampling points. In our implementation, we sample the CSI of 129 and 768 subcarriers on the 2.4 GHz and 5 GHz band, which makes $C = 897$.

A singular value decomposition (SVD) of $\boldsymbol{H_S}$ yields $\boldsymbol{H_S} = \boldsymbol{U\Sigma V}^T$, where $\boldsymbol{U}$ and $\boldsymbol{V}$ contain orthogonal column vectors $\boldsymbol{u_i}$ and $\boldsymbol{v_i}$, and $\boldsymbol{\Sigma}$ is a diagonal matrix containing the singular values σ_i. Then the frequency response can be represented as $\boldsymbol{h_k} = \sum_{i=1}^{B} \sigma_i \boldsymbol{v_i}(k)\boldsymbol{u_i}$, where $\boldsymbol{u}_i$ are the bases of the new signal space, and $\sigma_i \boldsymbol{v_i}(k) = \boldsymbol{h_k}^T \boldsymbol{u_i}, \forall i = 1 \cdots B$, are the representation of $\boldsymbol{h_k}$ in this space. We remark that the SVD can run offline, and only takes the ground truth spectrum signature as input. At run time, we only need to compare the signature in the B dimension signal space, which is much smaller than C.

Further, note that the frequency response curves of different tags are only different in notch positions and shapes. Therefore, only the first L out of B singular values may be dominant ($L < B$), *i.e.*, $\boldsymbol{h_k} \approx \sum_{i=1}^{L} \sigma_i \boldsymbol{v_i}(k)\boldsymbol{u_i}$, In this way, the dimension of the signal space is reduced from B to the L most discriminative points, and the signature of the kth tag can now be represented as

$$\boldsymbol{s_k} = \begin{bmatrix} \sigma_1 \boldsymbol{v_1}(k) & \sigma_2 \boldsymbol{v_2}(k) & \cdots & \sigma_L \boldsymbol{v_L}(k) \end{bmatrix}. \tag{9}$$

In our implementation, we pick the value of L so that $\sum_{i=1}^{L} \sigma_i = 0.9 \cdot \sum_{i=1}^{B} \sigma_i$, which preserves the features that contribute to 90% of the covariance (differences) between frequency response curves.

At run time, the LiveTag client first measures the frequency response $\boldsymbol{h_r}$ of a tag (Sec. 5.1), and reduces it to an L dimension vector $\boldsymbol{s_r}$, where the ith element $\boldsymbol{s_r}(i) = \boldsymbol{h_r}^T \boldsymbol{u_i}$. It then compares $\boldsymbol{s_r}(i)$ with the signatures $\boldsymbol{s_k}$ of each known tag k, using the Euclidean distance metric. To discriminate the case where no tag is present, we also add a special frequency signature with no notches. Overall, the matching complexity is only $\mathcal{O}(BL^2)$, where computing the Euclidean distance between two L dimension vectors is $\mathcal{O}(L^2)$, and the obtained frequency response should be compared to the B ones stored.

5.3 Touch Detection

Once the tag is identified, LiveTag can detect touch events by monitoring the disappearance of its notches. To make the detection robust to channel fading and avoid the need for a constant threshold, we formulate it as a Constant False Alarm Rate (CFAR) [6] detection problem, which is commonly adopted in radar signal processing.

Given a measured frequency response $\boldsymbol{h}$, each of the element $\boldsymbol{h}(i)$ represents the CSI of one subcarrier. Each subcarrier has a very narrow width, and can be modeled as a flat fading channel, *i.e.*, the real and imaginary part of $\boldsymbol{h}(i)$ follow Gaussian distribution $\mathcal{N}(\mu_i, {\varrho_i}^2)$ [48]. We assume each subcarrier bears the same level of noise, *i.e.*, $\varrho_i = \varrho$. But the mean values μ_i differ due to the frequency-selective fading.

To identify a touch event, LiveTag tracks the change across two consecutive snapshots of the frequency response, denoted as $\boldsymbol{h_t}$ and $\boldsymbol{h_{t-\Delta t}}$, respectively. The change can be quantified as

$$\boldsymbol{h'_t} = \boldsymbol{h_t} - \boldsymbol{h_{t-\Delta t}}, \text{ where} \tag{10}$$

$$\text{Re}\{\boldsymbol{h'_t}(i)\} \sim \mathcal{N}(0, 2\varrho^2), \text{Im}\{\boldsymbol{h'_t}(i)\} \sim \mathcal{N}(0, 2\varrho^2) \tag{11}$$

Therefore, the amplitude $\|\boldsymbol{h'_t}(i)\|$ follows a Rayleigh distribution with scale parameter $\sqrt{2}\varrho$, and the CDF

$$F(x) = 1 - \exp\Big(-\frac{x^2}{4\varrho^2}\Big), \; x \geq 0. \tag{12}$$

When the notch point disappears due to touch, $||\boldsymbol{h'_t}(i)||$ will experience a peak centered at the notch frequency. Thus, LiveTag confirms the detection of touch if $\|\boldsymbol{h'_t}(i)\| > V_{\text{th}}$, where the threshold V_{th} can be configured based on the target false alarm rate:

$$P_f(V_{\text{th}}) = \exp\Big(-\frac{{V_{\text{th}}}^2}{4\varrho^2}\Big). \tag{13}$$

ϱ is estimated and kept updated by measuring the mean value of $||\boldsymbol{h'_t}||$. According to the property of Rayleigh distribution [13], $\varrho = \frac{\mathbb{E}||\boldsymbol{h'_t}||}{\sqrt{\pi}}$. Note that a small $P_f(V_{\text{th}})$ may lead to large miss-detection rate (P_m). The intrinsic tradeoff will be evaluated empirically in Sec. 7.

Improving robustness through redundancy. To further improve the robustness of touch detection, LiveTag adds redundancy to the touch point, through frequency and temporal diversity. *First*, LiveTag can use multiple co-located resonators with different frequency notches to represent one compound touch point. A touch event is detected if over half of the notches are detected to change. Such diversity benefit comes at the cost of reducing tag capacity, but is desirable if robustness is of first concern. *Second*, in mobile scenarios, the variation of CSI leads to temporal diversity, and LiveTag can sample more than one set of CSI over time, before applying

***Figure* 15:** Experiment setup with commodity WiFi.

the PCA to obtain the final tag response. This weakens more fading profiles that are likely to be uncorrelated, albeit at the cost of increasing detection latency.

6. Implementation and Experimental Setup

Printing tags. To produce the tag with the desired frequency response, we first conduct ADS simulations following Sec. 3 to design the tag geometries. For fabrication, we use two types of substrates that are common for microwave systems: FR-4 and RT/duroid 6010 laminates (shown in Fig. 2). A laminate board consists of two copper layers and the dielectric substrate in between. The thickness and relative permittivity is 1.588 *mm* and 4.6 for FR-4, and 0.254 *mm* and 10.7 for RT/duroid 6010, respectively. We employ standard PCB milling technology to fabricate the 2D layout of the conductive layer on one side of the laminate, whereas the back side is directly used as the ground layer.

To verify each tag fabrication, we first fabricate one version where the transmission line ends with SMA connectors (without the two antennas). Then we follow the network analyzer setup (Sec. 3.3) to measure the intrinsic frequency response. Once the measurement matches simulation, we proceed to design a full-fledged tag that replaces the SMA interface with the patch/monopole antenna for over-the-air experiments.

Detection algorithms. We use the Atheros AR9462 WiFi cards, with the ath9k driver, to implement and test a basic version of LiveTag without beamforming (Fig. 15). Since the WiFi cards do not support customized beamforming, we use a 6-antenna WARP software radio as AP, and a single-antenna WARP board as client, to implement and test LiveTag's beamforming mechanism. The implementation realizes 802.11ac-compatible preamble generation, packet detection/synchronization, and per subcarrier, per-antenna CSI estimation. The LiveTag signal processing modules build atop this WiFi PHY. The transmitter-side fading suppression and LOS nulling algorithms replace the beamforming weights of the normal 802.11ac beamforming with the customized weights specified in Sec. 5. The receiver-side processes the CSI following the tag identification and touch detection algorithms (Sec. 5). Our tests show that it only takes 105 ms on average to stitch the CSI by transmitting one packet per channel, and sequentially switch across all channels.

We conduct all the experiments in a 14 m $\times 7$ m office, with walls and various objects together creating a reflection-rich multipath environment. Ordinary human activities are always present in the room. There also exist

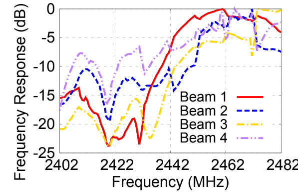


***Figure* 16:** Frequency response obtained from 4 individual beams.

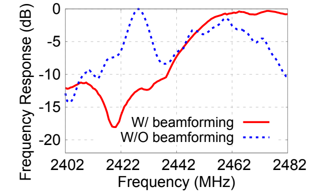


***Figure* 17:** Beamforming strengthens the spectrum signature.

10 external WiFi APs nearby, mostly with moderate traffic. Although our experiments occasionally experience such external interferences, the impacts are negligible because of the short packet duration. The interference can be automatically avoided once we migrate LiveTag to normal 802.11 nodes with a full-fledged MAC layer.

When evaluating LiveTag, we use filter gain as a microscopic metric, and miss detection (P_m), false alarm (P_f) rate as system level metrics. Given the frequency response, the *filter gain* is computed using the average gain of the passband minus the gain at the center frequency point of the notch.

7. Evaluation

7.1 Microbenchmarks on Tag/Touch Detection

7.1.1 *Fading Suppression and LOS Nulling*

Microscopic verification. We first verify LiveTag's transmitter side function using a 5-resonator tag with 2 patch antennas and 5 identical L-shaped resonators, with center frequency 2.42 GHz and absolute filter gain 35 dB according to our network analyzer measurement. The tag is 1.5 *m* and 0.3 *m* from the AP and client, respectively.

For clarity, we define *air filter gain* as the filter gain detected over wireless, which is usually different from the one measured by network analyzer due to noise, fading as well as LiveTag's signal processing algorithm.

We use 4 antennas on the AP to run the fading suppression and LOS nulling, and measure the over-the-air frequency response (CSI) at the client. Fig. 16 plots the result, where each frequency response curve is normalized to 0 dB relative to its peak value. Although all the curves manifest a notch at the resonator's center frequency, each suffers from frequency selective fading, resulting in random notches across the spectrum. Thus, *using the frequency response of a single beam may lead to severe false alarms*. In contrast, LiveTag's fading suppression smooths out the multipath artifacts, making the resonator notch much more pronounced, as shown in Fig. 17. When LiveTag's beamforming mechanisms are disabled, the tag notch becomes unobservable (dotted curve), which again verifies the critical role of fading suppression and beam nulling.

Fig. 19 (a) also shows that nulling alone increases the

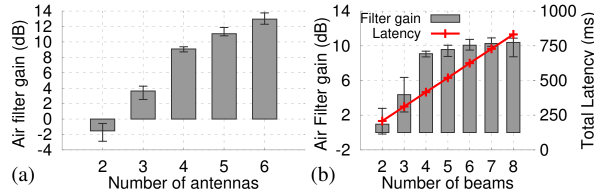


***Figure* 18:** Performance of beamforming with different number of (a) antennas and (b) beams. Error bars denote max/min across 10 runs.

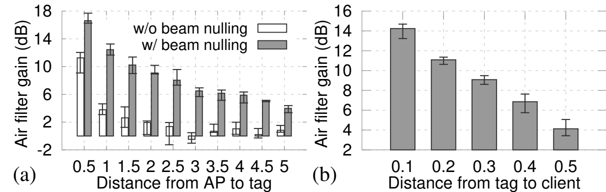


***Figure* 19:** Performance of beamforming under different (a) AP-to-tag (b) tag-to-client range. Error bars denote max/min across 10 runs.

filter gain by 3-5 dB. Without it, the LOS signals overwhelm the tag reflection, so the air filter gain drops sharply to below 4 dB, even at an AP-to-tag distance of 1 *m*.

Impact of the number of AP antennas. Due to the limited number of antennas, the beams generated by the AP tend to span a wide angle and bear multiple side lobes [53]. So the corresponding fading profiles are not completely uncorrelated. In general, using more antennas can narrow the beamwidth, which can improve the effectiveness of fading suppression. Under the same setup as above, our measurement shows that *the filter gain increases significantly with the number of antennas* (Fig. 18 (a)), from around 0 dB with 2 antennas, to 13 dB with 6 antennas. This verifies the effectiveness of LiveTag's beamforming, as higher air filter gain directly translates into higher detection accuracy. In the rest of the experiments we all use 4 antennas, which most commodity 802.11ac APs have.

Impact of beam selection. Now we rerun the fading suppression but vary the number of beam patterns K. The result (Fig. 18 (b)) shows that, given 4 Tx antennas, increasing K beyond 4 only offers marginal benefit. On the other hand, the total packet transmission latency increases linearly with K, *e.g.*, up to around 800 ms for $K = 8$, which may be intolerable for use cases with quick touches. This tradeoff justifies the design choice of setting K equal to the number of Tx antennas (Sec. 5.1). In the following experiments, we will use 4 Tx antennas and 4 beams by default.

Detection range. To test the working range of LiveTag, we first fix the tag-to-client distance at 0.3 *m*, while varying the AP-to-tag distance. Fig. 19 (a) shows that, as the tag moves away from the AP, the air filter gain decreases. This is because a smaller fraction of signals are affected by the tag, resulting in lower impact on the channel response measured at the client. Nonetheless, even at a distance of 4 *m*, LiveTag achieves 6 dB air filter gain, which will be shown to be enough for achieving high detection accuracy (Sec. 7.1.2).

Now we fix the AP-to-tag distance at 2 *m*, and increase the tag-to-client distance. Fig. 19 (b) shows that the air filter gain drops accordingly, but at a much dramatic rate—to below 6 dB at 0.5 *m*. This implies that *the detectability of LiveTag is more sensitive to the tag-to-client distance*. It happens because the client only has a single antenna, and the muiltipath fading between the tag and client is not suppressed as effectively as that of the AP-to-tag.

These experiments imply that the WiFi client should be within close proximity to the tag to ensure high detection performance. However, this requirement does not restrict the applications of LiveTag, because the user must be close to the tagged object when interacting with it, and LiveTag can leverage the user's WiFi device (*e.g.*, smartphone in a pocket) as the client. In addition, this requirement enables many tags to coexist within the same space, because only the tag close to the user would have a high air filter gain, and generate a detectable signature.

Detecting tag with commodity WiFi. We repeat the over-the-air tag response measurement using the Atheros WiFi devices and two tags—one with a single notch at 2420 MHz, and the other with 5 notches at 5 GHz band (the same tag as used for Fig. 13). The extracted CSI amplitude is plotted in Fig. 20 (Note the 5360 MHz to 5460 MHz band is unaccessible due to FCC regulation). The results demonstrate that the spectrum signatures are clearly distinguishable even with single-antenna commodity WiFi devices. We expect the air filter gains can be even higher once we have control over the beamforming functions on such devices.

7.1.2 Performance of Tag Identification

Tag identification accuracy. To evaluate LiveTag's tag identification performance, we use a multiresonator tag with 8 compound touch points (Fig. 27), and hence 8 notches in its spectrum signature. To create multiple sets of spectrum signatures to represent different tags, we can short-circuit any resonator with a copper wire, thus eliminating its frequency notch. We create 11 tags in total, including a dummy tag with no signatures. By default, the tag is placed 2 *m* away from the Tx and 0.2 *m* from the Rx. We run LiveTag's identification algorithm, along with the nearest-neighbor (NN) and DTW algorithm (Sec. 5.2), every 1.2 minutes for 100 times over a period of 2 hours, with natural human activities around. The results show that LiveTag achieves over 95% accuracy, followed by NN (73%), and DTW (28%). DTW is designed to tolerate the shift/misalignment between curves, but it often erroneously matches two tags with different notch positions, and tends to overrate the fre-

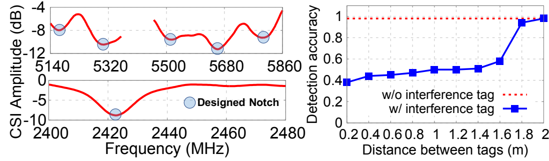


***Figure* 20:** Tag response measured by WiFi card.

***Figure* 21:** Detection of co-existing tags.

quency notches created by the multipath artifacts. LiveTag's signal-space transformation best harnesses the known spectrum signatures of the tags, by weighting on the points near the tag's intrinsic frequency notches.

Multi-tag coexistence. Next, we place a 5-resonator tag (target tag) at a fix location, 1.5 *m* (0.2 *m*) away from the AP (client). We then add a second tag (interference tag) and vary its distance from the target tag. Both tags have frequency notch at 2470 MHz. Fig. 21 shows that, without the interference tag, LiveTag detects the target tag with 98% accuracy. When the interference tag falls within 1.4 *m*, the detection accuracy drops to around 45%. Nonetheless, the accuracy improves as the interference tag moves away, and reaches above 94% at a distance beyond 1.8 *m*. Overall, a minimum separation of 2 *m* would ensure harmonious multi-tag coexistence.

7.1.3 *Performance of Touch Detection*

Our microbenchmarks focused on the filter gain metric. Now we create different air filter gains and measure the the actual P_f and P_m. We set the V_{th} to achieve $P_f = 3\%$. For each distance setting, the tag is touched 100 times across 1 hour, with human activities around. The same tag with 5 identical L-shaped resonators at 2.42 GHz is used. To adjust filter gain, we simply short different number of resonators with copper foil. Fig. 22(a) shows that the measured P_f is indeed kept close to the target. In addition, when the detected air filter gain is above 8 dB, the P_m is extremely low (<3%). As the air filter gain decreases, P_m increases rapidly (to 9% under 6 dB air filter gain and 28% under 4 dB), because the change of frequency response upon touch occurs at a similar level as channel noise/fading. We also found that increasing P_f decreases P_m, but the effect becomes negligible when $P_f > 3\%$, which can be used as a sweet spot to configure V_{th}.

Fig. 22(b) further shows the touch detection performance in NLOS scenarios. Without blockage, the measured air filter gain is 8 dB and $P_m = 3\%$. When a human body blocks the Tx-to-tag or tag-to-Rx path, P_m increases to 11% and 14%, respectively, whereas P_f still remains low due to the use of CFAR. Thus, to ensure consistently low P_m in NLOS, the tags and AP should be deployed to minimize the likelihood of blockage.

To verify the frequency-domain redundancy mechanism (Sec. 5.2), we add up to 6 additional resonators. The results show that the P_m decreases from 9% (under

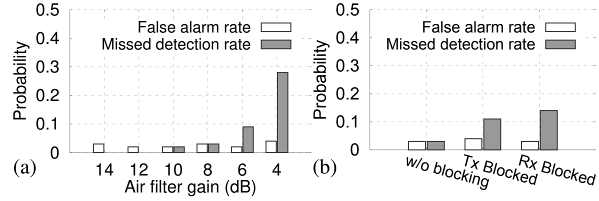


***Figure* 22:** Performance of touch detection under (a) LOS and (b) NLOS scenarios.

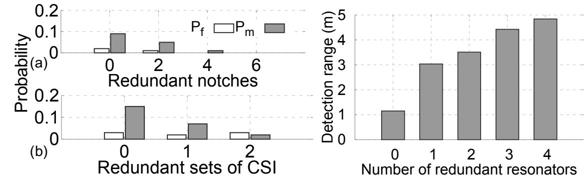


***Figure* 23:** Touch detection under redundant (a) notches (b) CSI.

***Figure* 24:** Detection range with redundant resonators.

6dB air filter gain in Fig. 22) down to 0 (Fig.23(a)). We further verify the time-domain redundancy by walking with the tag and Rx at 1 m/s. By aggregating 2 additional CSI measurements, the P_m can reduce from 15% to 2% (Fig.23(b)).

To verify the redundancy benefits brought by multiple redundant resonators, we repeat the experiment in Fig. 22, but measure the effective detection range between Tx to tag (a point where $P_m < 10\%$ when P_f is set to 3%). Fig. 24 shows that the range increases significantly, from 1.1 *m* to 4.8 *m*, as the number of redundant resonators increases from 0 to 4. Therefore, *LiveTag can harness redundant resonators to significantly improve the range and robustness of touch detection, albeit at the cost of tag size.*

7.2 Case Studies

We conduct three case studies, using to 3 different tags printed on the thin RT/duroid 6010 substrate, to verify LiveTag's ability to sense human-object interaction. Our experiments run in the same dynamic environment as above, where the tag is 1.5 *m* away from the Tx, and 0.3 *m* from the Rx. The P_f is configured to 3% by default.

Ubiquitous batteryless touch-pad and control panel. In smart-home environment, one can use LiveTag to create ad-hoc keypad or control panels, attached on walls in kitchen/bathroom, to remotely operate music players, lights, door locks, and numerous other IoT appliances. We have designed an example tag (Fig. 25) consisting of 9 compound touch points, each with 4 identical resonators. Following the same experimental methods in Sec. 7.1.3, we measure the detection accuracy. The results (Fig. 26) show that the P_m and P_f typically fall below 5%, which verifies the effectiveness of LiveTag in practical usage scenarios.

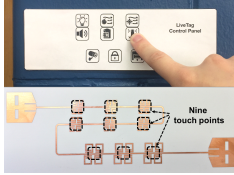


***Figure* 25:** Control panel.

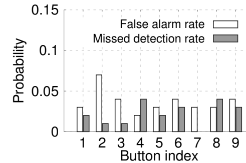


***Figure* 26:** Performance of control panel tag.

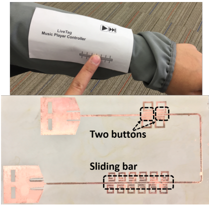


***Figure* 27:** Music controller tag.

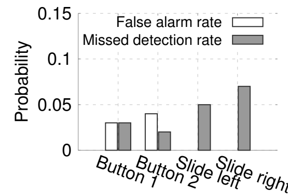


***Figure* 28:** Performance of music controller tag.

We have also designed a LiveTag-enabled music controller, with a start/pause button, next-track button, and a sliding bar for tuning volume (Fig. 27). The sliding bar comprises 6 resonators with different notch frequencies at 5 GHz band. The tag can be attached to a nightstand, kitchen wall, couch armrest, or even clothes, to remotely control WiFi-connected speakers or music players.

In our implementation, a sliding event is detected if at least 4 of the touch points on the sliding bar are detected sequentially. Our measurement (Fig. 28) shows that the detection accuracy of the two buttons is similar to Fig. 22. But the sliding detection is more robust, with $P_f \approx 0$, due to the joint effects of multiple touch points.

Augmenting everyday objects with touch-sensitivity. Since LiveTag is printable on thin, flexible substrates, it can be easily attached to plain objects, making them alive and enabling touch-related activity tracking. Here we adapt LiveTag to design a water level detector, which infers a user's water intake by tracking the water level in a cup. This simple application can deliver reminders to the user's WiFi device, alleviating the dehydration issue that many people suffer from[1].

Specifically, we design a tag with 2 monopole antennas and 12 L-shaped resonators, all with the same resonance frequency of 5.6 GHz. The tag is attached vertically to a water bottle, with its conductive layer facing inwards (Fig. 29). The water can affect the tag's frequency response, in the same way as finger touch, as long as the bottle is not made of metal which has a shielding effect. To avoid the impact of water on the antenna, we fold the antenna part outward.

Fig. 30 shows the detected filter gain over different water levels. The first resonator is detuned by the water when water level reaches 4 *cm*. With higher water levels, more resonators are disabled, causing the filter gain to drop proportionally. This simple relation can be harnessed by LiveTag to detect the water level. Note that the bottle movement may also affect the filter gain, but LiveTag can still track the short-term relative change within a period when the bottle remains stable. Due to space constraint, we leave a full-fledged design and implementation of this application as future work.

[1] 43% of the US adults suffer from dehydration without being aware of it [14].

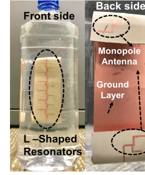


***Figure* 29:** Layout of water level detector tag.

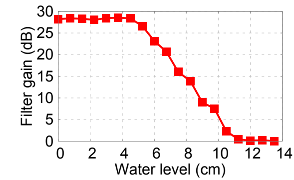


***Figure* 30:** Detected filter gain over water level.

8. Limitations and Future Work

Detection range and reliability. Our experiments indicate that LiveTag can reliably detect touches up to a Tx-to-tag distance of 4.8 *m*. The detection accuracy drops to below 90% at longer ranges, in NLOS conditions, or when the tag-to-Rx distance increases. There exist multiple potential ways to mitigate the performance loss, *e.g.*, increasing number of Tx antennas, designing detection algorithms that accommodate multiple Rx antennas, and incorporating redundant interrogating packets/signals to improve reliability. We plan to explore these mechanisms as future work.

Other use cases. Although our main focus lies in touch detection, LiveTag does show high potential in other usage scenarios such as inter-object interaction. Besides, owing to the ability to detect the presence and identity of different tags, LiveTag itself can act as a low-cost WiFi-detectable chipless RFID tag, which brings the vision of chipless RFID closer to everyday life. To make LiveTag more lightweight, we will explore inkjet printing [45], with conductive ink and photopaper as fabrication materials, which can make LiveTag easily available for customizing wireless sensing.

9. Conclusion

We have demonstrated the feasibility and effectiveness of LiveTag, a passive, batteryless, chipless metallic tag which responds to touches in a way that can be remotely detected by WiFi receivers. LiveTag marks the first step in achieving two visions: *(i)* Reconfigurable wireless sensing. Since LiveTag holds potential to be inkjet-printed on photopapers, it allows users to customize various types of WiFi-detectable touch interfaces in smart home environment. *(ii)* Converting dumb objects into smart ones. By attaching the tag, even plain everyday objects can be "smart", track human activities, and become part of the Internet of Things through the LiveTag WiFi detector. A more in-depth exploration of these visions is a matter of our future work.

References

[1] Advanced Design system (ADS). http://www.keysight.com/en/pc-1297113/advanced-design-system, 2016.

[2] ANAND, N., LEE, S.-J., AND KNIGHTLY, E. W. STROBE: Actively Securing Wireless Communications Using Zero-Forcing Beamforming. In *Proceedings of IEEE INFOCOM* (2012).

[3] BARRETT, L. F., AND BARRETT, D. J. An Introduction to Computerized Experience Sampling in Psychology. *Social Science Computer Review 19*, 2 (2001).

[4] BEJARANO, O., KNIGHTLY, E. W., AND PARK, M. Ieee 802.11 ac: from channelization to multi-user mimo. *IEEE Communications Magazine 51*, 10 (2013), 84–90.

[5] BELL, H. C. L-resonator bandstop filters. *IEEE transactions on microwave theory and techniques 44*, 12 (1996), 2669–2672.

[6] BLUM, R. S., KASSAM, S. A., AND POOR, H. V. Distributed detection with multiple sensors ii. advanced topics. *Proceedings of the IEEE 85*, 1 (1997), 64–79.

[7] CHEN, Z. N., AND CHIA, M. Y. W. *Broadband planar antennas: design and applications*. John Wiley & Sons, 2006.

[8] CHO, N., YOO, J., SONG, S.-J., LEE, J., JEON, S., AND YOO, H.-J. The human body characteristics as a signal transmission medium for intrabody communication. *IEEE transactions on microwave theory and techniques 55*, 5 (2007), 1080–1086.

[9] DELAITRE, V., SIVIC, J., AND LAPTEV, I. Learning Person-object Interactions for Action Recognition in Still Images. In *International Conference on Neural Information Processing Systems (NIPS)* (2011).

[10] DEY, S., SAHA, J. K., AND KARMAKAR, N. C. Smart Sensing: Chipless RFID Solutions for the Internet of Everything. *IEEE Microwave Magazine 16*, 10 (2015).

[11] FORBES. Adventures in Self-Surveillance, aka The Quantified Self, aka Extreme Navel-Gazing, Apr. 2011.

[12] GARCIA-PERATE, G., DALTON, N., CONROY-DALTON, R., AND WILSON, D. Ambient Recommendations in the Pop-up Shop. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)* (2013).

[13] GARDINER, C. W. *Stochastic methods*. Springer-Verlag, Berlin–Heidelberg–New York–Tokyo, 1985.

[14] GOODMAN, A. B. Behaviors and Attitudes Associated with Low Drinking Water Intake among US adults, Food Attitudes and Behaviors Survey. *Preventing Chronic Disease 10* (2013).

[15] GUPTA, A., KEMBHAVI, A., AND DAVIS, L. S. Observing Human-Object Interactions: Using Spatial and Functional Compatibility for Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence 31*, 10 (2009).

[16] HOYLE, R., TEMPLEMAN, R., ARMES, S., ANTHONY, D., CRANDALL, D., AND KAPADIA, A. Privacy Behaviors of Lifeloggers Using Wearable Cameras. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)* (2014).

[17] ISHII, H. The Tangible User Interface and Its Evolution. *Communications of the ACM 51*, 6 (2008).

[18] IYER, V., CHAN, J., AND GOLLAKOTA, S. 3D Printing Wireless Connected Objects. In *Proceedings of ACM SIGGRAPH Asia* (2017).

[19] KELLOGG, B., PARKS, A., GOLLAKOTA, S., SMITH, J. R., AND WETHERALL, D. Wi-Fi Backscatter: Internet Connectivity for RF-powered Devices.

[20] KELLOGG, B., TALLA, V., GOLLAKOTA, S., AND SMITH, J. Passive Wi-Fi: Bringing Low Power to Wi-Fi Transmissions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2016).

[21] KEOGH, E., AND RATANAMAHATANA, A. Everything You Know About Dynamic Time Warping is Wrong. *SIG-KDD Workshop on Mining Temporal and Sequential Data* (2004).

[22] KHANNA, A., AND GARAULT, Y. Determination of loaded, unloaded, and external quality factors of a dielectric resonator coupled to a microstrip line. *IEEE Transactions on Microwave Theory and Techniques 31*, 3 (1983), 261–264.

[23] LEE, Y.-T., LIM, J.-S., KIM, C.-S., AHN, D., AND NAM, S. A compact-size microstrip spiral resonator and its application to microwave oscillator. *IEEE microwave and wireless components letters 12*, 10 (2002), 375–377.

[24] LI, H., BROCKMEYER, E., CARTER, E. J., FROMM, J., HUDSON, S. E., PATEL, S. N., AND SAMPLE, A. PaperID: A Technique for Drawing Functional Battery-Free Wireless Interfaces on Paper. In *CHI Conference on Human Factors in Computing Systems (CHI)* (2016).

[25] LI, H., YE, C., AND SAMPLE, A. P. IDSense: A Human Object Interaction Detection System Based on Passive UHF RFID. In *Proc. of ACM Conference on Human Factors in Computing Systems (CHI)* (2015).

[26] LIU, V., PARKS, A., TALLA, V., GOLLAKOTA, S., WETHERALL, D., AND SMITH, J. R. Ambient Backscatter: Wireless Communication Out of Thin Air. In *Proc. of ACM SIGCOMM* (2013).

[27] MARQUARDT, N., TAYLOR, A. S., VILLAR, N., AND GREENBERG, S. Rethinking RFID: Awareness and Control for Interaction with RFID Systems. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)* (2010).

[28] MELI-SEGU, J., AND POUS, R. Human-Object Interaction Reasoning Using RFID-Enabled Smart Shelf. In *International Conference on the Internet of Things (IOT)* (2014).

[29] MILLIGAN, T. A. *Modern antenna design*. John Wiley & Sons, 2005.

[30] NISHI, T., SATO, Y., AND KOIKE, H. SnapLink: Interactive Object Registration and Recognition for Augmented Desk Interface. In *Proc. of IFIP Conference on HCI* (2001).

[31] PERRET, E. *Radio Frequency Identification and Sensors: From RFID to Chipless RFID*. Willey-ISTE, 2014.

[32] PHILIPOSE, M., FISHKIN, K. P., PERKOWITZ, M., PATTERSON, D. J., FOX, D., KAUTZ, H., AND HAHNEL, D. Inferring Activities from Interactions with Objects. *IEEE Pervasive Computing 3*, 4 (2004).

[33] POZAR, D. M. *Microwave and Rf Design of Wireless Systems*, 1st ed. Wiley, 2000.

[34] POZAR, D. M. *Microwave Engineering*. Wiley, 2012.

[35] PRADHAN, S., CHAI, E., SUNDARESAN, K., QIU, L., KHOJASTEPOUR, M. A., AND RANGARAJAN, S. Rio: A pervasive rfid-based touch gesture interface. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (2017).

[36] PRERADOVIC, S., BALBIN, I., KARMAKAR, N. C., AND SWIEGERS, G. F. Multiresonator-based chipless rfid system for low-cost item tracking. *IEEE Transactions on Microwave Theory and Techniques 57*, 5 (2009), 1411–1419.

[37] PRERADOVIC, S., AND KARMAKAR, N. C. Design of fully printable planar chipless rfid transponder with 35-bit data capacity. In *Microwave Conference, 2009. EuMC 2009. European* (2009), IEEE, pp. 013–016.

[38] PRERADOVIC, S., AND KARMAKAR, N. C. Chipless RFID: Bar Code of the Future. *IEEE Microwave Magazine 11*, 7 (2010).

[39] RANJAN, J., AND WHITEHOUSE, K. Object Hallmarks: Identifying Object Users Using Wearable Wrist Sensors. In *ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)* (2015).

[40] SAMPLE, A. P., YEAGER, D. J., AND SMITH, J. R. A Capacitive Touch Interface for Passive RFID Tags. In *IEEE International Conference on RFID* (2009).

[41] SCHMIDT, A., GELLERSEN, H. W., AND MERZ, C. Enabling Implicit Human Computer Interaction: a Wearable RFID-Tag Reader. In *International Symposium on Wearable Computers (ISWC)* (2000).

[42] SCHMIDT, R. Multiple emitter location and signal parameter estimation. *IEEE transactions on antennas and propagation 34*, 3 (1986), 276–280.

[43] SCHMITZ, M., BAUS, J., AND DÖRR, R. *The Digital*

Sommelier: Interacting with Intelligent Products. 2008.

[44] SEDRA, A. S., AND SMITH, K. C. *Microelectronic circuits*, vol. 1. New York: Oxford University Press, 1998.

[45] SHAO, B. *Fully Printed Chipless RFID Tags towards Item-Level Tracking Applications*. PhD thesis, Royal Institute of Technology, 2014.

[46] SIMON, T. M., THOMAS, B. H., SMITH, R. T., AND SMITH, M. Adding input controls and sensors to rfid tags to support dynamic tangible user interfaces. In *Proceedings of the International Conference on Tangible, Embedded and Embodied Interaction (TEI)* (2013).

[47] TAPIA, E. M., INTILLE, S. S., AND LARSON, K. Activity Recognition in the Home Using Simple and Ubiquitous Sensors. In *Proceedings of the Second International Conference on Pervasive Computing (PERVASIVE)* (2004).

[48] TSE, D., AND VISWANATH, P. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[49] VAN VEEN, B. D., AND BUCKLEY, K. M. Beamforming: A Versatile Approach to Spatial Filtering. *IEEE ASSP Magazine 5*, 2 (1988).

[50] VASISHT, D., KUMAR, S., AND KATABI, D. Decimeter-level localization with a single wifi access point. In *USENIX NSDI* (2016).

[51] WEISER, M. The Computer for the 21st Century (Reprint). *ACM SIGMOBILE Mobile Computing and Communications Review 3*, 3 (1999).

[52] WILSON, H. J. You, By the Numbers. *Harvard Business Review* (Sep. 2012).

[53] XIE, X., CHAI, E., ZHANG, X., SUNDARESAN, K., KHOJASTEPOUR, A., AND RANGARAJAN, S. Hekaton: Efficient and practical large-scale mimo. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (2015), ACM, pp. 304–316.

[54] YAO, B., AND FEI-FEI, L. Grouplet: A Structured Image Representation for Recognizing Human and Object Interactions. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)* (2010).

Inaudible Voice Commands: The Long-Range Attack and Defense

Nirupam Roy, Sheng Shen, Haitham Hassanieh, Romit Roy Choudhury
University of Illinois at Urbana-Champaign

Abstract

Recent work has shown that inaudible signals (at ultrasound frequencies) can be designed in a way that they become audible to microphones. Designed well, this can empower an adversary to stand on the road and silently control Amazon Echo and Google Home-like devices in people's homes. A voice command like "Alexa, open the garage door" can be a serious threat.

While recent work has demonstrated feasibility, two issues remain open: (1) The attacks can only be launched from within $5ft$ of Amazon Echo, and increasing this range makes the attack audible. (2) There is no clear solution against these ultrasound attacks, since they exploit a recently discovered loophole in hardware non-linearity.

This paper is an attempt to close both these gaps. We begin by developing an attack that achieves $25ft$ range, limited by the power of our amplifier. We then develop a defense against this class of voice attacks that exploit non-linearity. Our core ideas emerge from a careful forensics on voice, i.e., finding indelible traces of non-linearity in recorded voice signals. Our system, *LipRead*, demonstrates the inaudible attack in various conditions, followed by defenses that only require software changes to the microphone.

1 Introduction

A number of recent research papers have focused on the topic of inaudible voice commands [37, 48, 39]. Backdoor [37] showed how hardware non-linearities in microphones can be exploited, such that *inaudible ultrasound signals* can become audible to any microphone. DolphinAttack [48] developed on Backdoor to demonstrate that no software is needed at the microphone, i.e., a voice enabled device like Amazon Echo can be made to respond to inaudible voice commands. A similar paper independently emerged in arXiv [39], with a video demonstration of such an attack [3]. These attacks are becoming increasingly relevant, particularly with the proliferation of voice enabled devices including Amazon Echo, Google Home, Apple Home Pod, Samsung refrigerators, etc.

While creative and exciting, these attacks are still deficient on an important parameter: *range*. DolphinAttack can launch from a distance of $5ft$ to Amazon Echo [48] while the attack in [39] achieves $10ft$ by becoming partially audible. In attempting to enhance range, we realized strong tradeoffs with inaudibility, i.e., the output of the speaker no longer remains silent. This implies that currently known attacks are viable in short ranges, such as Alice's friend visiting Alice's home and silently attacking her Amazon Echo [11, 48]. However, the general, and perhaps more alarming attack, is the one in which the attacker parks his car on the road and controls voice-enabled devices in the neighborhood, and even a person standing next to him does not hear it. This paper is an attempt to achieve such an attack radius, followed by defenses against them. We formulate the core problem next and outline our intuitions and techniques for solving them.

Briefly, non-linearity is a hardware property that makes high frequency signals arriving at a microphone, say s_{hi}, get shifted to lower frequencies s_{low} (see Figure 1). If s_{hi} is designed carefully, then s_{low} can be almost identical to s_{hi} but shifted to within the audibility cutoff of $20kHz$ inside the microphone. As a result, even though humans do not hear s_{hi}, non-linearity in microphones produces s_{low}, which then become legitimate voice commands to devices like Amazon Echo. This is the root opportunity that empowers today's attacks.

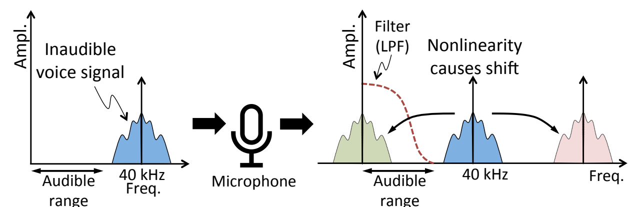


Figure 1: Hardware non-linearity creates frequency shift. Voice commands transmitted over inaudible ultrasound frequencies get shifted into the lower audible bands after passing through the non-linear microphone hardware.

Two important points need mention at this point. (1) Non-linearity triggers at high frequencies and at high power – if s_{hi} is a soft signal, then the non-linear effects do not surface. (2) Non-linearity is fundamental to acoustic hardware and is equally present in speakers as in microphones. Thus, when s_{hi} is played through speak-

ers, it will also undergo the frequency shift, producing an audible s_{low}. Dolphin and other attacks sidestep this problem by operating at low power, thereby forcing the output of the speaker to be almost inaudible. This inherently limits the range of the attack to $5ft$; any attempt to increase this range will result in audibility.

This paper breaks away from the zero sum game between range and audibility by an alternative transmitter design. Our core idea is to use multiple speakers, and stripe segments of the voice signal across them such that leakage from each speaker is narrow band, and confined to low frequencies. This still produces a garbled, audible sound. To achieve true inaudibility, we solve a min-max optimization problem on the length of the voice segments. The optimization picks the segment lengths in a way such that the aggregate leakage function is completely below the human auditory response curve (i.e., the minimum separation between the leakage and the human audibility curve is maximized). This ensures, by design, the attack is inaudible.

Defending against this class of non-linearity attacks is not difficult if one were to assume hardware changes to the receiver (e.g., Amazon Echo or Google Home). An additional ultrasound microphone will suffice since it can detect the s_{hi} signals in air. However, with software changes alone, the problem becomes a question of forensics, i.e., can the shifted signal s_{low} be discriminated from the same legitimate voice command, s_{leg}. In other words, does non-linearity leave an indelible trace on s_{low} that would otherwise not be present in s_{leg}.

Our defense relies on the observation that voice signals exhibit well-understood structure, composed of fundamental frequencies and harmonics. When this structure passes through non-linearity, part of it remains preserved in the shifted and blended low frequency signals. In contrast, legitimate human voice projects almost no energy in these low frequency bands. An attacker that injects distortion to hide the traces of voice, either pollutes the core voice command, or raises the energy floor in these bands. This forces the attacker into a zero-sum game, disallowing him from erasing the traces of non-linearity without raising suspicion.

Our measurements confirm the possibility to detect voice traces, i.e., even though non-linearity superimposes many harmonics and noise signals on top of each other, and attenuates them significantly, cross-correlation still reveals the latent voice fingerprint. Of course, various intermediate steps of contour tracking, filtering, frequency-selective compensation, and phoneme correlation are necessary to extract out the evidence. Nonetheless, our final classifier is transparent and does not require any training at all, but succeeds for voice signals only, as opposed to the general class of inaudible microphone attacks (such as jamming [37]). We leave this broader problem to future work.

Our overall system *LipRead* is built on multiple platforms. For the inaudible attack at long ranges, we have developed an ultrasound speaker array powered by our custom-made amplifier. The attacker types a command on the laptop, MATLAB converts the command to a voice signal, and the laptop sends this through our amplifier to the speaker. We demonstrate controlling Amazon Echo, iPhone Siri, and Samsung devices from a distance of $25ft$, limited by the power of our amplifier. For defense, we record signals from Android Samsung S6 phones, as well as from off-the-shelf microphone chips (popular in today's devices). We attack the system with various ultrasound commands, both from literature as well as our own. *LipRead* demonstrates defense against all attacks with 97% precision and 98% recall. The performance remains robust across varying parameters, including multipath, power, attack location, and various signal manipulations.

Current limitations: Our long-range attacks have been launched from within a large room, or from outside a house with open windows. When doors and windows were closed, the attack was unsuccessful since our high-frequency signals attenuated while passing through the wall/glass. We believe this is a function of power, however, a deeper treatment is necessary around this question. In particular: (1) Will high power amplifiers be powerful enough for high-frequency signals to penetrate such barriers? (2) Will high-power and high-frequency signals trigger non-linearity inside human ears? (3) Are there other leakages that will emerge in such high power and high frequency regimes. We leave these questions to future work.

In sum, our core contributions may be summarized as follows:

- A transmitter design that breaks away from the tradeoff between attack range and audibility. The core ideas pertain to carefully striping frequency bands across an array of speakers, such that individual speakers are silent but the microphone is activated.

- A defense that identifies human voice traces at very low frequencies (where such traces should not be present) and uses them to protect against attacks that attempt to erase or disturb these traces.

The subsequent sections elaborate on these ideas, beginning with some relevant background on non-linearity, followed by threat model, attack design, and defense.

2 Background: Acoustic Non-linearity

Microphones and speakers are in general designed to be linear systems, meaning that the output signals are linear combinations of the input. In the case of power amplifiers inside microphones and speakers, if the input sound signal is $s(t)$, then the output should ideally be:

$$s_{out}(t) = A_1 s(t)$$

where A_1 is the amplifier gain. In practice, however, acoustic components in microphones and speakers (like diaphragms, amplifiers, etc.) are linear only in the audible frequency range ($< 20kHz$). In ultrasound bands ($> 25kHz$), the responses exhibit non-linearity [28, 19, 16, 38, 22]. Thus, for ultrasound signals, the output of the amplifier becomes:

$$\begin{aligned} s_{out}(t) = & \sum_{i=1}^{\infty} A_i s^i(t) = A_1 s(t) + A_2 s^2(t) + A_3 s^3(t) + \ldots \\ \approx & \; A_1 s(t) + A_2 s^2(t) \end{aligned} \tag{1}$$

Higher order terms are typically extremely weak since $A_{4+} \ll A_3 \ll A_2$ and hence can be ignored.

Recent work [37] has shown ways to exploit this phenomenon, i.e., it is possible to play ultrasound signals that cannot be heard by humans but can be directly recorded by any microphone. Specifically, an ultrasound speaker can play two inaudible tones: $s_1(t) = \cos(2\pi f_1 t)$ at frequency $f_1 = 38$kHz and $s_2 = \cos(2\pi f_2 t)$ at frequency $f_2 = 40$kHz. Once the combined signal $s_{hi}(t) = s_1(t) + s_2(t)$ passes through the microphone's nonlinear hardware, the output becomes:

$$\begin{aligned} s_{out}(t) = & \; A_1 s_{hi}(t) + A_2 s_{hi}^2(t) \\ = & \; A_1(s_1(t) + s_2(t)) + A_2(s_1(t) + s_2(t))^2 \\ = & \; A_1 \cos(2\pi f_1 t) + A_1 \cos(2\pi f_2 t) \\ + & \; A_2 \cos^2(2\pi f_1 t) + A_2 \cos^2(2\pi f_2 t) \\ + & \; 2A_2 \cos(2\pi f_1 t)\cos(2\pi f_2 t) \end{aligned}$$

The above signal has frequency components at f_1, f_2, $2f_1$, $2f_2$, $f_2 + f_1$, and $f_2 - f_1$. This can be seen by expanding the equation:

$$\begin{aligned} s_{out}(t) = & \; A_1 \cos(2\pi f_1 t) + A_1 \cos(2\pi f_2 t) \\ + & \; A_2 + 0.5A_2 \cos(2\pi 2 f_1 t) + 0.5A_2 \cos(2\pi 2 f_2 t) \\ + & \; A_2 \cos(2\pi(f_1 + f_2)t) + A_2 \cos(2\pi(f_2 - f_1)t) \end{aligned}$$

Before digitizing and recording the signal, the microphone applies a low pass filter to remove frequency components above the microphone's cutoff of $24KHz$. Observe that f_1, f_2, $2f_1$, $2f_2$, and $f_1 + f_2$ are all > 24kHz. Hence, what remains (as acceptable signal) is:

$$s_{low}(t) = A_2 + A_2 \cos(2\pi(f_2 - f_1)t) \tag{2}$$

This is essentially a $f_2 - f_1 = 2$kHz tone which will be recorded by the microphone. However, this demonstrates the core opportunity, i.e., by sending a *completely inaudible signal*, we are able to generate an audible "copy" of it inside any unmodified off-the-shelf microphone.

3 Inaudible Voice Attack

We begin by explaining how the above non-linearity can be exploited to send inaudible commands to *voice enabled devices* (VEDs) at a short range. We identify deficiencies in such an attack and then design the longer range, truly inaudible attack.

3.1 Short Range Attack

Let $v(t)$ be a baseband voice signal that once decoded translates to the command: "Alexa, mute yourself". An attacker moves this baseband signal to a high frequency $f_{hi} = 40$kHz (by modulating a carrier signal), and plays it through an ultrasound speaker. The attacker also plays a tone at $f_{hi} = 40$kHz. The played signal is:

$$s_{hi}(t) = \cos(2\pi f_{hi} t) + v(t)\cos(2\pi f_{hi} t) \tag{3}$$

After this signal passes through the non-linear hardware and low-pass filter of the microphone, the microphone will record:

$$s_{low}(t) = \frac{A_2}{2}\left(1 + v^2(t) + 2v(t)\right) \tag{4}$$

This shifted signal contains a strong component of $v(t)$ (due to more power in the speech components), and hence, gets decoded correctly by almost all microphones.

■ **What happens to $v^2(t)$?**

Figure 2 shows the power spectrum $V(f)$ corresponding to the voice command $v(t) =$"Alexa, mute yourself". Here the power spectrum corresponding to $v^2(t)$ which is equal to $V(f) * V(f)$ where $(*)$ is the convolution operation. Observe that the spectrum of the human voice is between $[50 - 8000]Hz$ and the relatively weak components of $v^2(t)$ line up underneath the voice frequencies after convolution. A component of $v^2(t)$ also falls at DC, however, degrades sharply. The overall weak presence of $v^2(t)$ leaves the $v(t)$ signal mostly unharmed, allowing VEDs to decode the command correctly.

However, to help $v(t)$ enter the microphone through the "non-linear inlet", $s_{hi}(t)$ must be transmitted at sufficiently high power. Otherwise, $s_{low}(t)$ will be buried in noise (due to small A_2). *Unfortunately, increasing the transmit power at the speaker triggers non-linearities at the speaker's own diaphragm and amplifier*, resulting in an audible $s_{low}(t)$ at the output of the speaker. Since $s_{low}(t)$ contains the voice command $v(t)$, the attack becomes audible. Past attacks sidestep this problem by operating at low power, thereby forcing the output of the

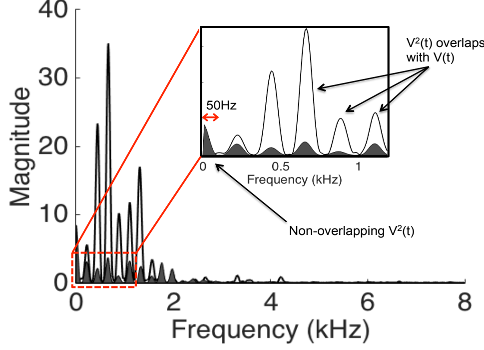


Figure 2: Spectrum of $V(f) * V(f)$ which is the non-linear leakage after passing through the microphone

speaker to be almost inaudible [49]. This inherently limits the radius of attack to a short range of $5ft$. Attempts to increase this range results in audibility, defeating the purpose of the attack.

Figure 3 confirms this with experiments in our building. Five volunteers visited marked locations and recorded their perceived loudness of the speaker's leakage. Clearly, speaker non-linearity produces audibility, a key problem for long range attacks.

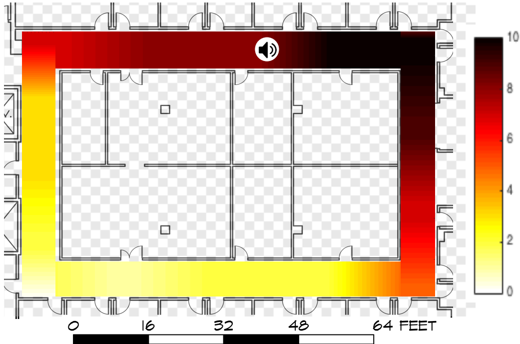


Figure 3: Heatmap showing locations at which $v(t)$ leakage from the speaker is audible.

3.2 Long Range Attack

Before developing the long range attack, we concisely present the assumptions and constraints on the attacker.

■ **Threat Model:** We assume that:

- The attacker cannot enter the home to launch the attack, otherwise, the above short range attack suffices.
- The attacker cannot leak any audible signals (even in a beamformed manner), otherwise such inaudible attacks are not needed in the first place.
- The attacker is resourceful in terms of hardware and energy (perhaps the attacking speaker can be carried in his car or placed at his balcony, pointed at VEDs in surrounding apartments or pedestrians).
- In case the receiver device (e.g., Google Home) is voice fingerprinted, we assume the attacker can synthesize the legitimate user's voice signal using known techniques [46, 5] to launch the attack.
- The attacker cannot estimate the precise *channel impulse response* (CIR) from its speaker to the voice enabled device (VED) that it intends to attack.

■ **Core Attack Method:**
LipRead develops a new speaker design that facilitates considerably longer attack range, while eliminating the audible leakage at the speaker. Instead of using one ultrasound speaker, *LipRead* uses multiple of them, physically separated in space. Then, *LipRead* splices the spectrum of the voice command $V(f)$ into carefully selected segments and plays each segment on a different speaker, thereby limiting the leakage from each speaker.

■ **The Need for Multiple Speakers:**
To better understand the motivation, let us first consider using two ultrasound speakers. Instead of playing $s_{hi}(t) = \cos(2\pi f_{hi}t) + v(t)\cos(2\pi f_{hi}t)$ on one speaker, we now play $s_1(t) = \cos(2\pi f_{hi}t)$ on the first speaker and $s_2(t) = v(t)\cos(2\pi f_{hi}t)$ on the second speaker where $f_{hi} = 40kHz$. In this case, the 2 speakers will output:

$$\begin{aligned} s_{out1} &= \cos(2\pi f_{hi}t) + \cos^2(2\pi f_{hi}t) \\ s_{out2} &= v(t)\cos(2\pi f_{hi}t) + v^2(t)\cos^2(2\pi f_{hi}t) \end{aligned} \tag{5}$$

For simplicity, we ignore the terms A_1 and A_2 (since they do not affect our understanding of frequency components). Thus, when s_{out1} and s_{out2} emerge from the two speakers, human ears filter out all frequencies $> 20kHz$. What remains audible is only:

$$\begin{aligned} s_{low1} &= 1/2 \\ s_{low2} &= v^2(t)/2 \end{aligned}$$

Observe that neither s_{low1} nor s_{low2} contains the voice signal $v(t)$, hence the actual attack command is no longer audible with two speakers. However, the microphone under attack will still receive the aggregate ultrasound signal from the two speakers, $s_{hi}(t) = s_1(t) + s_2(t)$, and its own non-linearity will cause a "copy" of $v(t)$ to get shifted into the audible range (recall Equation 4). Thus, this 2-speaker attack activates VEDs from greater distances, while the actual voice command remains inaudible to bystanders.

Although the voice signal $v(t)$ is inaudible, signal $v^2(t)$ still leaks and becomes audible (especially at higher power). This undermines the attack.

■ **Suppressing $v^2(t)$ Leakage:**
To suppress the audibility of $v^2(t)$, *LipRead* expands to N ultrasound speakers. It first partitions the audio spectrum

$V(f)$ of the command signal $v(t)$, ranging from f_0 to f_N, into N frequency bins: $[f_0, f_1]$, $[f_1, f_2]$, $\cdots$, $[f_{N-1}, f_N]$ as shown in Fig. 4. This can be achieved by computing an FFT of the signal $v(t)$ to obtain $V(f)$. $V(f)$ is then multiplied with a rectangle function $rect(f_i, f_{i+1})$ which gives a filtered $V_{[f_i,f_{i+1}]}(f)$. An IFFT is then used to generate $v_{[f_i,f_{i+1}]}(t)$ which is multiplied by an ultrasound tone $\cos(2\pi f_{hi} t)$ and outputted on the ith ultrasound speaker as shown in Fig. 4.

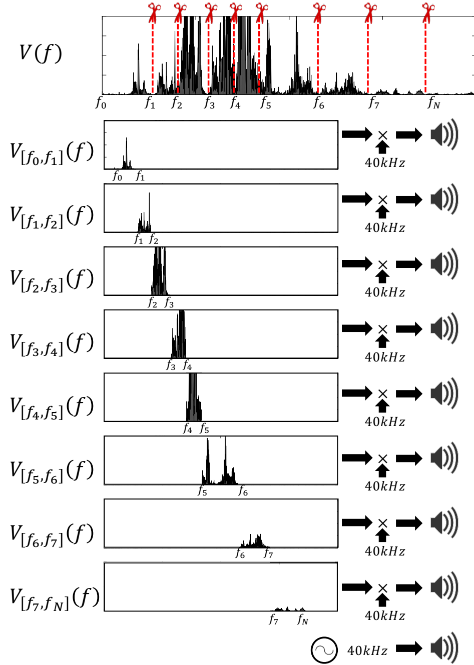


Figure 4: Spectrum Splicing: optimally segmenting the voice command frequencies and playing it through separate speakers so that the net speaker-output is silent.

In this case, the audible leakage from i^{th} ultrasound speaker will be $s_{low,i}(t) = v^2_{[f_i,f_{i+1}]}(t)$. In the frequency domain, we can write this leakage as:

$$S_{low,i}(f) = \quad V_{[f_i,f_{i+1}]}(f) * V_{[f_i,f_{i+1}]}(f)$$

This leakage has two important properties:

$$(1) \quad E\left[|S_{low,i}(f)|^2\right] \leq E\left[|V(f) * V(f)|^2\right]$$
$$(2) \quad BW(S_{low,i}(f)) \leq BW(V(f) * V(f))$$

where $E[|.|^2]$ is the power of audible leakage and $BW(.)$ is the bandwidth of the audible leakage due to non-linearities at each speaker. The above properties imply that splicing the spectrum into multiple speakers reduces the audible leakage from any given speaker. It also reduces the bandwidth and hence concentrates the audible leakage in a smaller band below 50 Hz.

While per-speaker leakage is smaller, they can still add up to become audible. The total leakage power can be written as:

$$L(f) = \left|\sum_{i=1}^{N} V_{[f_i,f_{i+1}]}(f) * V_{[f_i,f_{i+1}]}(f)\right|^2$$

To achieve true inaudibility, we need to ensure that the total leakage is not audible. To address this challenge, we leverage the fact that humans cannot hear the sound if the sound intensity falls below certain threshold, which is frequency dependent. This is known as the "Threshold of Hearing Curve", $T(f)$. Fig. 5 shows $T(f)$ in dB as function of frequency. Any sound with intensity below the threshold of hearing will be inaudible.

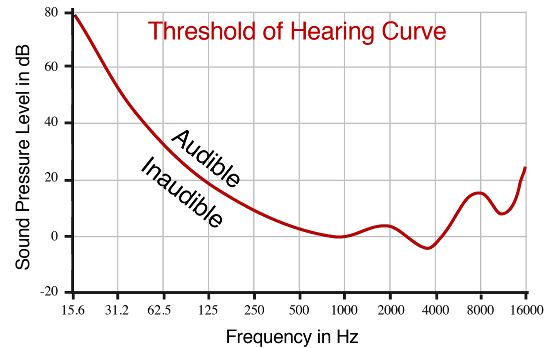


Figure 5: Threshold of Hearing Curve

LipRead aims to push the total leakage spectrum, $L(f)$, below the "Threshold of Hearing Curve" $T(f)$. To this end, *LipRead* finds the best partitioning of the spectrum, such that the leakage is below the threshold of hearing. If multiple partitions satisfy this constraint, *LipRead* picks the one that has the largest gap from the threshold of hearing curve. Formally, we solve the below optimization problem:

$$\begin{aligned} &\underset{\{f_1,f_2,\cdots,f_{N-1}\}}{\text{maximize}} \ \min_f [T(f) - L(f)] \\ &\text{subject to} f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_N \end{aligned} \tag{6}$$

The solution partitions the frequency spectrum to ensure that the leakage energy is below the hearing threshold for every frequency bin. This ensures inaudibility at any human ear.

■ Increasing Attack Range:
It should be possible to increase attack range with more speakers, while also limiting audible leakage below the required hearing threshold. This holds in principle due to the following reason. For a desired attack range, say r, we can compute the minimum power density (i.e., power per frequency) necessary to invoke the VED. This power P_r needs to be high since the non-linear channel will strongly attenuate it by the factor A_2. Now consider the worst case where a voice command has equal magnitude in all frequencies. Given each frequency needs power P_r and each speaker's output needs to be below *threshold*

of hearing for all frequencies, we can run our *min-max optimization* for increasing values of N, where N is the number of speakers. The minimum N that gives a feasible solution is the answer. Of course, this is the upper bound; for a specific voice signal, N will be lower.

Increasing speakers can be viewed as beamforming the energy towards the VED. In the extreme case for example, every speaker will play one frequency tone, resulting in a strong DC component at the speaker's output which would still be inaudible. In practice, our experiments are bottlenecked by ADCs, amplifiers, speakers, etc., hence we will report results with an array of 61 small ultrasound speakers.

4 Defending Inaudible Voice Commands

Recognizing inaudible voice attacks is essentially a problem of acoustic forensics, i.e., detecting evidence of nonlinearity in the signal received at the microphone. Of course, we assume the attacker knows our defense techniques and hence will try to remove any such evidence. Thus, the core question comes down to: *is there any trace of non-linearity that just cannot be removed or masked?*

To quantify this, let $v(t)$ denote a human voice command signal, say "Alexa, mute yourself". When a human issues this command, the recorded signal $s_{leg} = v(t) + n(t)$, where $n(t)$ is noise from the microphone. When an attacker plays this signal over ultrasound (to launch the non-linear attack), the recorded signal s_{nl} is:

$$s_{nl} = \frac{A_2}{2}(1 + 2v(t) + v^2(t)) + n(t) \quad (7)$$

Figure 6 shows an example of s_{leg} and s_{nl}. Evidently, both are very similar, and both invoke the same response in VEDs (i.e., the text-to-speech converter outputs the same text for both s_{leg} and s_{nl}). A defense mechanism would need to examine any incoming signal s and tell if it is low-frequency legitimate or a shifted copy of the high-frequency attack.

4.1 Failed Defenses

Before we describe *LipRead*'s defense, we mention a few other possible defenses which we have explored before converging on our final defense system. We concisely summarize 4 of these ideas.

■ **Decompose Incoming Signal $s(t)$:**
One solution is to solve for $s(t) = \frac{A_2}{2}(1 + 2\hat{v}(t) + \hat{v}^2(t))$, and test if the resulting $\hat{v}(t)$ produces the same text-to-speech (T2S) output as $s(t)$. However, this proved to be a fallacious argument because, if such a $\hat{v}(t)$ exists, it will always produce the same T2S output as $s(t)$. This is because such a $\hat{v}(t)$ would be a cleaner version of the voice command (without the non-linear component); if the polluted version s passes the T2S test, the cleaner version obviously will.

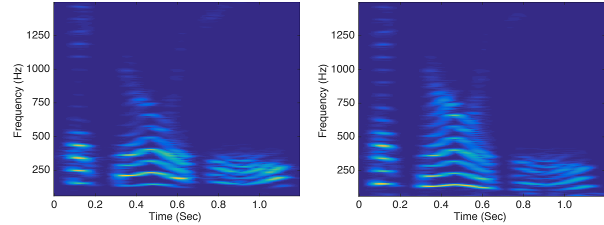


Figure 6: Spectrogram for s_{leg} and s_{nl} for voice command "Alexa, mute yourself".

■ **Energy at Low Frequencies from $v^2(t)$:**
Another solution is to extract portions of $s(t)$ from the lower frequencies – since regular voice signals do not contain sub-$50Hz$ components, energy detection should offer evidence. Unfortunately, environmental noise (e.g., fans, A/C machines, wind) leaves non-marginal residue in these low bands. Moreover, an attacker could deliberately reduce the power of its signal so that its leakage into sub-$50Hz$ is small. Our experiments showed non-marginal false positives in the presence of environmental sound and soft attack signals.

■ **Amplitude Degradation at Higher Frequencies:**
The air absorbs ultrasound frequencies far more than voice (which translates to sharper reduction in amplitude as the ultrasound signal propagates). Measured across different microphones separated by $\approx 7.3cm$ in Amazon Echo and Google Home, the amplitude difference should be far greater for ultrasound. We designed a defense that utilized the maximum amplitude slope between microphone pairs – this proved to be a robust discriminator between s_{leg} and s_{nl}. However, we were also able to point two (reasonably synchronized) ultrasound beams from opposite directions. This reduced the amplitude gradient, making it comparable to legitimate voice signals (Alexa treated the signals as multipath). In the real-world, we envisioned 2 attackers launching this attack by standing at 2 opposite sides of a house. Finally, this solution would require an array of microphones on the voice enabled device. Hence, it is inapplicable to one or two microphone systems (like phones, wearables, refrigerators).

■ **Phase Based Separation of Speakers:**
Given that long range attacks need to use at least 2 speakers (to bypass speaker non-linearity), we designed an angle-of-arrival (AoA) based technique to estimate the physical separation of speakers. In comparison to human voice, the source separation consistently showed success, so long as the speakers are more than $2cm$ apart. While practical attacks would certainly require multiple speakers, easily making them $2cm$ apart, we aimed at solving

the short range attack as well (i.e., where the attack is launched from a single speaker). Put differently, the right evidence of non-linearity should be one that is present regardless of the number of speakers used.

4.2 LipRead Defense Design

Our final defense is to search for traces of $v^2(t)$ in sub-50Hz. However, we now focus on exploiting the structure of human voice. The core observation is simple: voice signals exhibit well-understood patterns of fundamental frequencies, added to multiple higher order harmonics (see Figure 6). We expect this structure to partly reflect in the sub-50Hz band of $s(t)$ (that contains $v^2(t)$), and hence correlate with carefully extracted spectrum above-50Hz (which contains the dominant $v(t)$). With appropriate signal scrubbing, we expect the correlation to emerge reliably, however, if the attacker attempts to disrupt correlation by injecting sub-50Hz noise, the stronger energy in this low band should give away the attack. We intend to force the attacker into this zero sum game.

■ **Key Question: Why Should $v^2(t)$ Correlate?**
Figure 7(a) shows a simplified abstraction of a legitimate voice spectrum, with a narrow fundamental frequency band around f_j and harmonics at integer multiples nf_j. The lower bound on f_j is $> 50Hz$ [41]. Now recall that when this voice spectrum undergoes non-linearity, each of f_j and nf_j will self-convolve to produce "copies" of themselves around DC (Figure 7(b)). Of course, the A_2 term from non-linearity strongly attenuates this "copy". *However, given the fundamental band around f_j and the harmonics around nf_j are very similar in structure, each of $\approx 20Hz$ bandwidth, the energy between $[0, 20kHz]$ superimposes.* This can be expressed as:

$$E_{[0,20]} \approx E\left[A_2 \sum_{n=1}^{N} |V_{[nf_j-20,\ nf_j+20]} * V_{[nf_j-20,\ nf_j+20]}|^2\right] \tag{8}$$

The net result is distinct traces of energy in sub-20Hz bands, and importantly, this energy variation (over time) mimics that of f_j. For a legitimate attack, on the other hand, the sub-20Hz is dominantly uncorrelated hardware and environmental noise.

Figure 8(a) and (b) zoom into sub-50Hz and compare the traces of energy for s_{leg} and s_{nl}, respectively. The s_{nl} signal clearly shows more energy concentration, particularly when the actual voice signal is strong. Figure 9 plots the power in the sub-50Hz band with increasing voice loudness levels for both s_{leg} and s_{nl}. Note that loudness level is expressed in $dBSpl$, where Spl denotes "sound pressure level", the standard metric for measuring sound. Evidently, non-linearity shows increasing power due to the self-convolved spectrum overlapping in

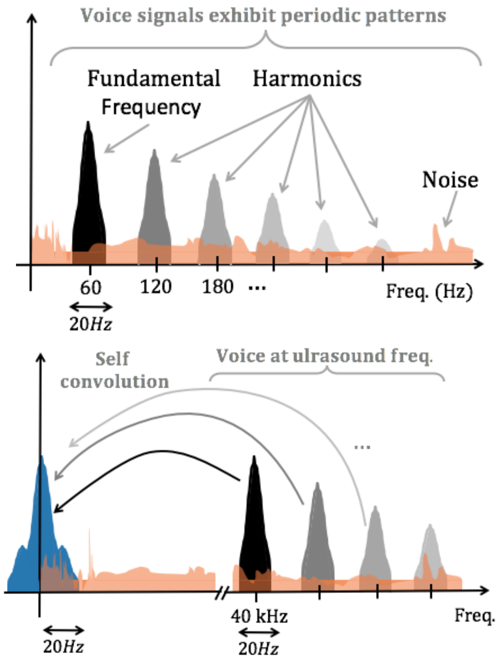


Figure 7: (a) A simplified voice spectrum showing the structure. (b) Voice spectra after non-linear attack.

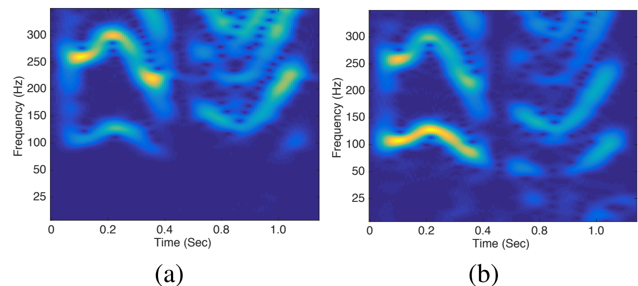


Figure 8: Spectrogram of the (a) audible and (b) inaudible attack voice. The attack signal contains higher power below 50Hz, indicated by lighter color.

the lower band. Legitimate voice signals generate significantly less energy in these bands, thereby remaining flat for higher loudness.

■ **Correlation Design**
The width of the fundamental frequencies and harmonics are time-varying, however, at any given time, if it is $B\,Hz$, then the self-convolved signal gets shifted into $[0,B]Hz$ as well. Note that this is independent of the actual values of center frequencies, f_j and nf_j. Now, let $s_{<B}(t)$ denote the sub-$B\ Hz$ signal received by the microphone and $s_{>B}(t)$ be the signal above $B\ Hz$ that contains the voice command. *LipRead* seeks to correlate the energy variation over time in $s_{<B}(t)$ with the energy variation at the fundamental frequency, f_j in $s_{>B}(t)$. We track the fundamental frequency in $s_{>B}(t)$ using standard acoustic libraries, but then average the power around $B\ Hz$ of this frequency. This produces a power profile over time, P_{f_j}. For $s_{<B}(t)$, we also track the average power

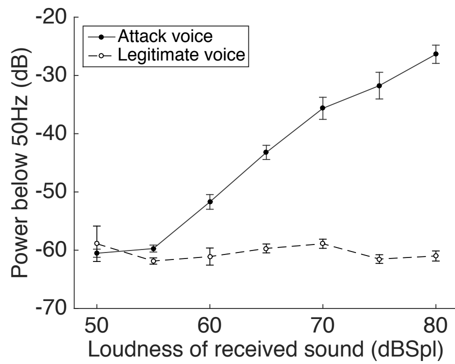


Figure 9: The loudness vs sub-50Hz band power plot.

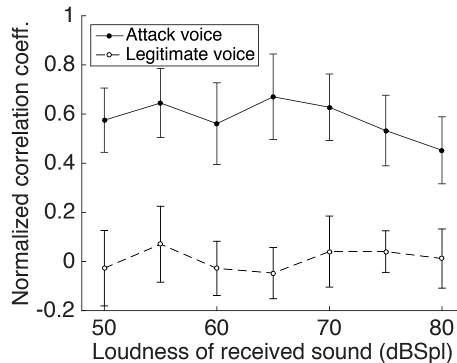


Figure 10: The loudness vs correlation between P_{f_j} and $s_{<B}(t)$, denoting the power variation of the fundamental frequency and the sub 20Hz band, respectively.

over time. However, to avoid weak signals and disruption from noise, we remove time windows in which f_j's power is below its average. We stitch together the remaining windows from both P_{f_j} and $s_{<B}(t)$ and compute their correlation co-efficient. We use an average value of $B = 20Hz$.

Figure 10 shows the correlation for increasing loudness levels of the recorded signal (loudness below 60$dBSpl$ is not audible). The comparison is against a legitimate voice command. Evidently, we recorded consistent correlation gap, implying that non-linearity is leaving some trace in the low-frequency bands, and this trace preserves some structure of the actual voice signal. Of course, we have not yet accounted for the possibility that the attacker can inject noise to disrupt correlation.

■ **Improved Attack via Signal Shaping**

The natural question for the attacker is how to modify/add signals such that the correlation gap gets narrowed. Several possibilities arise:

(1) Signal $-v^2(t)$ can be added to the speaker in the low frequency band and transmitted with the high frequency ultrasound $v(t)$. Given that ultrasound will produce $v^2(t)$ after non-linearity, and $-v^2(t)$ will remain as is, the two should interact at the microphone and cancel. Unfortunately, channels for low frequencies and ultrasound are different and unknown, hence it is almost impossible to design the precise $-v^2(t)$ signal. Of course, we will still attempt to attack with such a deliberately shaped signal.

(2) Assuming the ultrasound $v(t)$ has been up-converted to $[40, 44]kHz$, the attacker could potentially concatenate spurious frequencies from say $[44, 46]kHz$. These frequencies would also self-convolve and get "copied" around DC. This certainly affects correlation since these spurious frequencies would not correlate well (in fact, they can be designed to not correlate). The attacker's hope should be to lower correlation while maintaining a low energy footprint below 20Hz.

The attacker can use the above approaches to try to defeat the zero-sum game. Figure 11 plots results from 4000 attempts to achieve low correlation and low energy. Of these, 3500 are random noises injected in legitimate voice commands, while the remaining 500 are more carefully designed distortions (such as frequency concatenation, phase distortions, low frequency injection, etc.). Of course, in all these cases, the distorted signal was still correct, i.e., the VED device responded as it should.

On the other hand, 450 different legitimate words were spoken by different humans (shown as hollow dots), at various loudness levels, and accents, and styles. Clusters emerge suggesting promise of separation. However, some commands were still too close, implying the need for greater margin of separation.

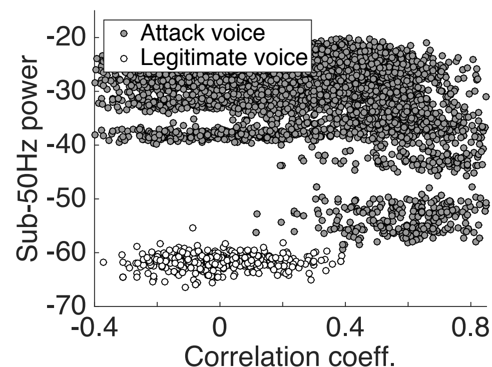
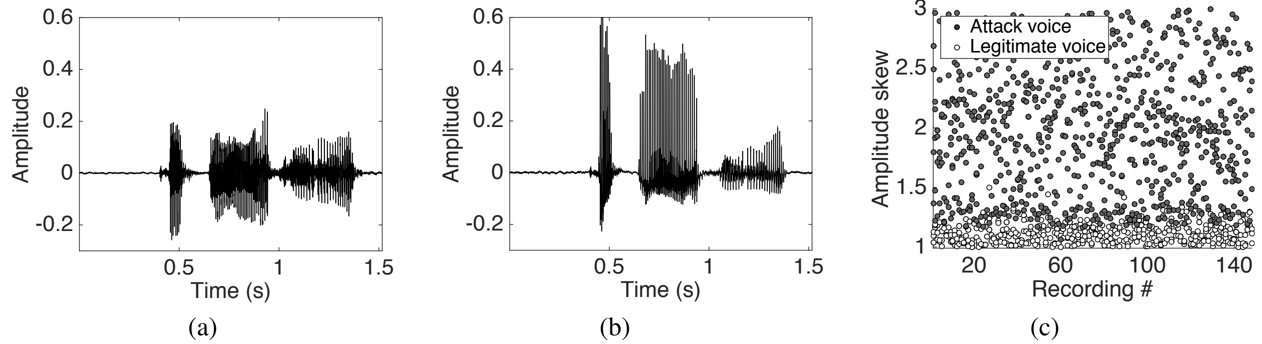


Figure 11: Zero sum game between correlation and power at sub-50Hz bands. Attacker attempts to reduce correlation by signal shaping or noise injection at sub-50Hz band.

■ **Leveraging Amplitude Skew from $v^2(t)$**

In order to increase the separation margin, *LipRead* leverages the amplitude skew resulting from $v^2(t)$. Specifically, two observations emerge: (1) When the harmonics in voice signals self-convolve to form $v^2(t)$, they fall at the same frequencies of the harmonics (since the gaps between the harmonics are quite homogeneous). (2) The signal $v^2(t)$ is a time domain signal with only posi-



(a) (b) (c)

Figure 12: (a) Sound signals in time domain from s_{leg} and (b) s_{nl}, demonstrating a case of amplitude skew. (c) Amplitude skew for various attack and legitimate voice commands.

tive amplitude. Combining these together, we postulated that amplitudes of the harmonics would be positively biased, especially for those that are strong (since $v^2(t)$ will be relatively stronger at that location). In contrast, amplitudes of legitimate voice signals should be well balanced on the positive and negative. Figure 12(a,b) shows one contrast between a legitimate voice s_{leg} and the recorded attack signal s_{nl}. In pursuit of this opportunity, we extract the ratio of the maximum and minimum amplitude (we average over the top 10% for robustness against outliers). Using this as the third dimension for separation, Figure 12(c) re-plots the s_{leg} and s_{nl} clusters. While the separation margin is close, combining it with correlation and power, the separation becomes satisfactory.

■ ***LipRead*'s Elliptical Classifier**

LipRead leverages 3 features to detect an attack: power in sub-50Hz, correlation coefficient, and amplitude skew. Analyzing the *False Acceptance Rate* (FAR) and *False Rejection Rate* (FRR), as a function of these 3 parameters, we have converged on a ellipsoidal-based separation technique. To determine the optimal decision boundary, we compute *False Acceptance Rate* (FAR) and *False Rejection Rate* (FRR) for each candidate ellipsoid. Our aim is to pick the parameters of the ellipse that minimize both FAR and FRR. Figure 13 plots the FAR and FRR as intersecting planes in a logarithmic scale (Note that we show only two features since it is not possible to visualize the 4D graph). The coordinate with minimum value along the canyon – indicating the *equal error rates* – gives the optimal selection of ellipsoid. Since it targets speech commands, this classifier is designed offline, one-time, and need not be trained for each device or individual.

5 Evaluation

We evaluate *LipRead* on 3 main metrics: (1) attack range, (2) inaudibility of the attack, and the recorded sound quality (i.e., whether the attacker's command sounds human-like), and (3) accuracy of the defense under various environments. We summarize our findings below.

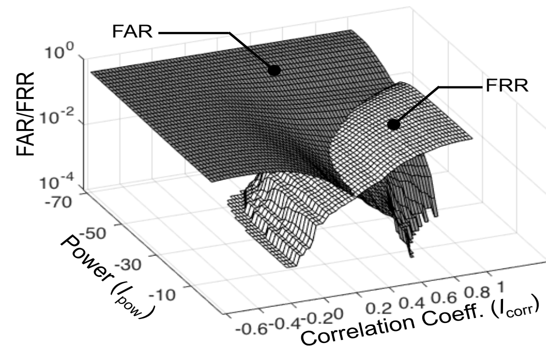


Figure 13: The False Acceptance Rate plane (dark color) and the False Rejection Rate plane (light color) for different sub-50Hz power and correlation values.

- We test our attack prototype with 984 commands to Amazon Echo and 200 commands to smartphones – the attacks are launched from various distances with 130 different background noises. Figure 15 shows attack success at $24ft$ for Amazon Echo and $30ft$ for smartphones at a power of $6watt$.
- We record 12 hours of microphone data – 5 hours of human voice commands and 7 hours of attack commands through ultrasound speakers. Figure 16(c) shows that attack words are recognized by VEDs with equal accuracy as legitimate human words. Figure 16(b) confirms that all attacks are inaudible, i.e., the leakage from our speaker array is 5-10dB below human hearing threshold.
- Figure 17(a) shows the precision and recall of our defense technique, as 98% and 99%, respectively, when the attacker does not manipulate the attack command. Importantly, precision and recall remain steady even under signal manipulation.

Before elaborating on these results, we first describe our evaluation platforms and methodology.

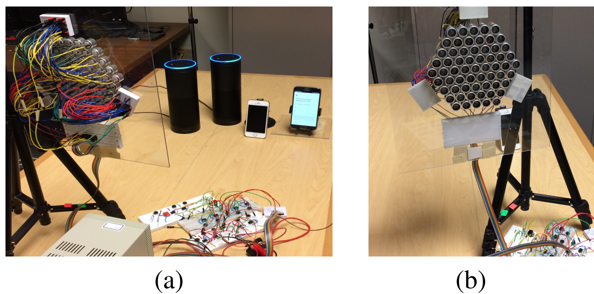
5.1 Platform and Methodology

(1) Attack speakers: Figure 14(b) shows our custom-designed speaker system consisting of 61 ultrasonic piezoelectric speakers arranged as a hexagonal planar array. The elements of the array are internally connected

in two separate clusters. A dual channel waveform generator (*Keysight 33500b series* [4]) drives the first cluster with the voice signal, modulated at the center frequency of $40kHz$. This cluster forms smaller sub-clusters to transmit separate segments of the spliced spectrum. The second cluster transmits the pure $40kHz$ tone through each speaker. The signals are amplified to 30 Volts using a custom-made NE5534AP op-amp based amplifier circuit. This prototype is limited to a maximum power of $6watt$ because of the power ratings of the operational amplifiers. More powerful amplifiers are certainly available to a resourceful attacker.

(2) Target VEDs: We test our attack on 3 different VEDs – Amazon Echo, Samsung $S6$ smartphone running Android v7.0, and Siri on an iPhone $5S$ running iOS v10.3. Unlike Echo, Samsung S-voice and Siri requires personalization of the wake-word with user's voice – adding a layer of security through voice authentication. However, voice synthesis is known to be possible [46, 5], and we assume that the synthesized wake-word is already available to the attacker.

Experiment setup: We run our experiments in a lab space occupied by 5 members and also in an open corridor. We place the VEDs and the ultrasonic speaker at various distances ranging up to $30ft$. During each attack, we play varying degrees of interfering signals from 6 speakers scattered across the area, emulating natural home/office noises. The attack signals were designed by first collecting real human voice commands from 10 different individuals; MATLAB is used to modulate them to ultrasound frequencies. For speech quality of the attack signals, we used the open-source *Sphinx4* speech processing tool [1].



(a) (b)

Figure 14: *LipRead* evaluation setup: (a) Ultrasonic speaker and voice enabled devices. (b) The ultrasonic speaker array for attack.

5.2 Attack Performance

Activation distance: This experiment attempts to activate the VEDs from various distances. We repeatedly play the inaudible wake-word from the ultrasound speaker system at regular intervals and count the fraction of successful activation. Figure 15(a) shows the activation hit rate against increasing distance – higher hit-rates indicate success with less number of attempts. The average distance achieved for 50% hit rate is $24ft$, while the maximum for Siri and Samsung S-voice are measured to be 27 and $30ft$ respectively.

Figure 15(b) plots the attack range again, but for the entire voice command. We declare "success" if the text to speech translation produces every single word in the command. The range degrades slightly due to the stronger need to decode every word correctly.

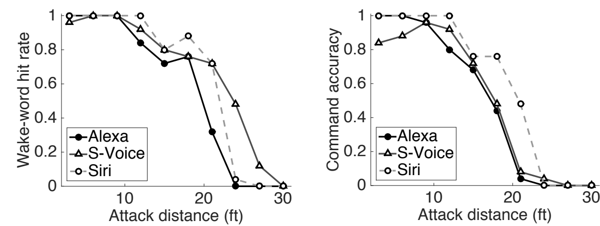


Figure 15: (a) The wake-word hit rate and (b) the command detection accuracy against increasing distances.

Figure 16(a) reports the attack range to Echo for increasing input power to the speaker system. As expected, the range continues to increase, limited by the power of our $6Watt$ amplifiers. More powerful amplifiers would certainly enhance the attack range, however, for the purposes of prototyping, we designed our hardware in the lower power regime.

Leakage audibility: Figure 16(b) plots the efficacy of our spectrum splicing optimization, i.e., how effectively does *LipRead* achieve speaker-side inaudibility for different ultrasound commands. Observe that without splicing (i.e., "no partition"), the ultrasound voice signal is almost $5dB$ above the human hearing threshold. As the number of segments increase, audibility falls below the hearing curve. With 60 speakers in our array, we use 6 segments, each played through 5 speakers; the remaining 31 were used for the second $cos(2\pi f_c t)$ signal. Note that the graph plots the minimum gap between the hearing threshold and the audio playback, implying that this is a conservative worst case analysis. Finally, we show results from 20 example attack commands – the other commands are below the threshold.

Received speech quality: Given 6 speakers were transmitting each spliced segment of the voice command, we intend to understand if this distorts speech quality. Figure 16(c) plots the word recognition accuracy via Sphinx [1], an automatic speech recognition software. Evidently, *LipRead*'s attack quality is comparable to human quality, implying that our multi-speaker beamforming preserves the speech's structure. In other words, speech quality is not the bottleneck for attack range.

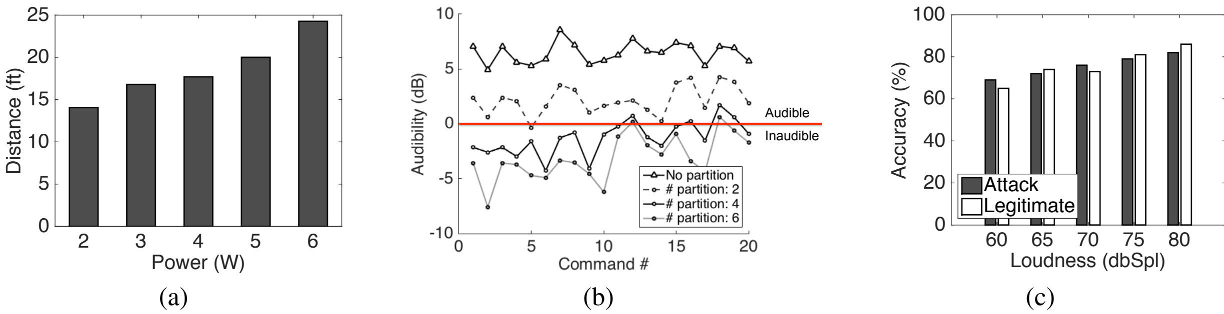


Figure 16: (a) Maximum activation distance for different input power. (b) Worst case audibility of the leakage sound after optimal spectrum partitioning. (c) Word recognition accuracy with automatic speech recognition software for attack and legitimate voices.

5.3 Defense Performance

Metrics: Our defense technique essentially attempts to classify the attack scenarios distinctly from the legitimate voice commands. We report the "*Recall*" and "*Precision*" of this classifier for various sound pressure levels (measured in *dBSPL*), varying degrees of ambient sounds as interference, and deliberate signal manipulation. Recall that our metrics refer to:

• *Precision:* What fraction of our detected attacks are correct?
• *Recall:* What fraction of the attacks did we detect?

We now present the graphs beginning with the basic classification performance.

Basic attack detection: Figure 17(a) shows the attack detection performance in normal home environment without significant interference. The average precision and recall of the system is 99% across various loudness of the received voice. This result indicates best case performance of our system with minimum false alarm.

Impact of ambient noise: In this section we test our defense system for common household sounds that can potentially mix with the received voice signal and change its features leading to misclassification. To this end, we played 130 noise sounds through multiple speakers while recording attack and legitimate voice signals with a smartphone. We replayed the noises at 4 different sound pressure levels starting from a typical value of 50 dBSPL to extremely loud 80 dBSPL, while the voice loudness is kept constant at 65 dBSpl. Figure 17(b) reports the precision and recall for this experiment. The recall remains close to 1 for all these noise levels, indicating that we do not miss attacks. However, at higher interference levels, the precision slightly degrades since the false detection rate increases a bit when noise levels are extremely high which is not common in practice.

Impact of injected noise: Next, we test the defense performance against deliberate attempts to eliminate nonlinearity features from the attack signal. Here the attackers strategy is to eliminate the $v^2(t)$ correlation by injecting noise in the attack signal. We considered four different categories of noise – white Gaussian noise to raise the noise floor, band-limited noise on the *Sub-50Hz* region, water-filling noise power at low frequencies to mask the correlated power variations, and intermittent frequencies below 50 Hz. As shown, in Figure 17(c), the process does not significantly impact the performance because of the power-correlation trade-off exploited by the defense classifier. Figure 17(d) shows that the overall accuracy of the system is also above 99% across all experiments.

6 Points of Discussion

We discuss several dimensions of improvement.

■ **Lack of formal guarantee:** We have not formally proved our defense. Although *LipRead* is systematic and transparent (i.e., we understand why it should succeed) it still leaves the possibility that an attack may breach the defense. Our attempts to mathematically model the self-convolution and correlation did not succeed since frequency and phase responses for general voice commands were difficult to model, as were real-world noises. A deeper treatment is necessary, perhaps with help from speech experts who can model the phase variabilities in speech. We leave this to future work.

■ **Generalizing to any signal:** Our defense is designed for the class of voice signals, which applies well to inaudible voice attacks. A better defense should find the true trace of non-linearity, not just for the special case of voice. This remains an open problem.

■ **Is air non-linear as well?** There is literature that claims air is also a non-linear medium [17, 10, 45]. When excited by adequately powerful ultrasound signals, self-convolution occurs, ultimately making sounds audible. Authors in [36, 2] are designing *acoustic spotlighting systems* where the idea is to make ultrasound signals audible only along a direction. We have encountered traces of air non-linearity, although in rare occasions. This certainly call for a separate treatment in the future.

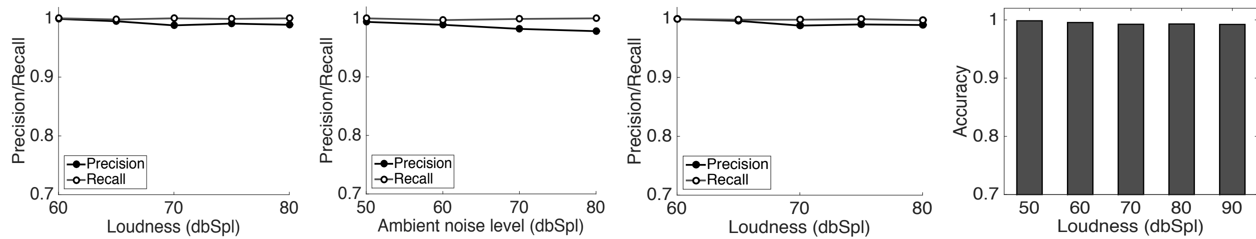


Figure 17: Precision and Recall of defense performance under various condition: (a) basic performance without external interference, (b) performance under ambient noise, and (c) performance under injected noise. (d) Overall accuracy across all experiments.

■ **Through-wall attack:** Due to the limited maximum power (6*watt*) of our amplifiers, we tested our system in non-blocking scenarios. If the target device is partially blocked (e.g. furnitures in the room blocking line-of-sight), the SNR reduces and our attack range will reduce. This level of power has not allowed us to launch through-wall attacks yet. We leave this to future work.

7 Related Work

■ **Attack on Voice Recognition Systems:** Recent research [11, 42] shows that spoken words can be mangled such that they are unrecognizable to humans, yet decodable by voice recognition (VR) systems. *GVS-Attack* [14] exploits this by creating a smartphone app that gives adversarial commands to its voice assistant. More recently, *BackDoor* [37] has taken advantage of the microphone's nonlinearity to design ultrasonic sounds which are inaudible to humans, but becomes recordable inside the off-the-shelf microphones. The application includes preventing acoustic eavesdropping with inaudible jamming signals. As follow up, [48, 39] show that the principles of *BackDoor* can be used to send inaudible attack commands to a VED, but requires physical proximity to remain audible. *LipRead* demonstrates the feasibility to increase the inaudible attack range, but more importantly, designs a defense against the inaudible attacks.

In past, researchers use near-ultrasound [27, 32, 40, 9, 21, 30] and exploited aliasing to record inaudible sound with microphone. A number of papers use other sound to camouflage audible signal in order to make it indistinguishable to human [24, 20, 12]. *CovertBand* [33] use music to hide audible harmonic components at the speaker. *LipRead*, on the other hand, use high frequency ultrasound as inaudible signal and leverages hardware nonlinearity to make them recordable to microphone.

■ **Acoustic Non-linearity:** A body of research [17, 10, 45], inspired by Westervelt's seminal theory [44, 43] on nonlinear Acoustics, studies the distortions of sound while moving through nonlinear mediums including the air. This raises the possibility that ultrasonic sound can naturally self-demodulate in the air to generate audible sounds, making it possible to develop a highly directional speaker [17, 10, 45]. Recently, *AudioSpotlight* [2], *SoundLazer* [7, 6], and other projects [47, 8, 34] have rolled out commercial products based on this concept. Ultrasonic hearing aids [29, 13, 15, 35, 31] and headphones [25] explore the human body as a nonlinear medium to enable voice transfer through bone conduction. Our work, however, is opposite of these efforts – we attempt to retain the inaudible nature of ultrasound while making it recordable inside electronic circuits.

■ **Speaker Linearization:** A number of research [23, 26, 18] studies the possibility of adaptive linearization of general speakers. Through simulations, the authors have shown that by pre-processing the input signal, they can achieve as much as $27dB$ reduction [18] of the nonlinear distortion in the noise-free case. Their techniques are not yet readily applicable to real speakers, since they have all assumed very weak nonlinearities, and over-simplified electrical and mechanical structures of speakers. With real speakers, especially ultrasonic piezoelectric speakers, it is difficult to fully characterize the parameters of the nonlinear model. Of course, if future techniques can fully characterize such models, our system can be made to achieve longer range with fewer speakers.

8 Conclusion

This paper builds on existing work to show that inaudible voice commands are viable from distances of 25+ ft. Of course, careful design is necessary to ensure the attack is truly inaudible – small leakages from the attacker's speakers can raise suspicion, defeating the attack. This paper also develops a defense against inaudible voice commands that exploit microphone non-linearity. We show that non-linearity leaves traces in the recorded voice signal, that are difficult to erase even with deliberate signal manipulation. Our future work is aimed at solving the broader class of non-linearity attacks for any signals, not just voice.

Acknowledgement

We sincerely thank our shepherd Prof. Shyamnath Gollakota and the anonymous reviewers for their valuable feedback. We are grateful to the Joan and Lalit Bahl Fellowship, Qualcomm, IBM, and NSF (award number: 1619313) for partially funding this research.

References

[1] Cmu sphinx. http://cmusphinx.sourceforge.net. Last accessed 6 December 2015.

[2] Holosonics webpage. https://holosonics.com. Last accessed 28 November 2016.

[3] Inaudible voice commands demo. https://www.youtube.com/watch?v=wF-DuVkQNQQ&feature=youtu.be. Last accessed 24 September 2017.

[4] Keysight waveform generator. http://literature.cdn.keysight.com/litweb/pdf/5991-0692EN.pdf. Last accessed 24 September 2017.

[5] Lyrebird. https://lyrebird.ai. Last accessed 24 September 2017.

[6] Soundlazer kickstarter. https://www.kickstarter.com/projects/richardhaberkern/soundlazer. Last accessed 28 November 2016.

[7] Soundlazer webpage. http://www.soundlazer.com. Last accessed 28 November 2016.

[8] Woody norris ted talk. https://www.ted.com/speakers/woody_norris. Last accessed 28 November 2016.

[9] AUMI, M. T. I., GUPTA, S., GOEL, M., LARSON, E., AND PATEL, S. Doplink: Using the doppler effect for multi-device interaction. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing* (2013), ACM, pp. 583–586.

[10] BJØRNØ, L. Parametric acoustic arrays. In *Aspects of Signal Processing*. Springer, 1977, pp. 33–59.

[11] CARLINI, N., MISHRA, P., VAIDYA, T., ZHANG, Y., SHERR, M., SHIELDS, C., WAGNER, D., AND ZHOU, W. Hidden voice commands. In *USENIX Security Symposium* (2016), pp. 513–530.

[12] CONSTANDACHE, I., AGARWAL, S., TASHEV, I., AND CHOUDHURY, R. R. Daredevil: indoor location using sound. *ACM SIGMOBILE Mobile Computing and Communications Review 18*, 2 (2014), 9–19.

[13] DEATHERAGE, B. H., JEFFRESS, L. A., AND BLODGETT, H. C. A note on the audibility of intense ultrasonic sound. *The Journal of the Acoustical Society of America 26*, 4 (1954), 582–582.

[14] DIAO, W., LIU, X., ZHOU, Z., AND ZHANG, K. Your voice assistant is mine: How to abuse speakers to steal information and control your phone. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (2014), ACM, pp. 63–74.

[15] DOBIE, R. A., AND WIEDERHOLD, M. L. Ultrasonic hearing. *Science 255*, 5051 (1992), 1584–1585.

[16] DOBRUCKI, A. Nonlinear distortions in electroacoustic devices. *Archives of Acoustics 36*, 2 (2011), 437–460.

[17] FOX, C., AND AKERVOLD, O. Parametric acoustic arrays. *The Journal of the Acoustical Society of America 53*, 1 (1973), 382–382.

[18] GAO, F. X., AND SNELGROVE, W. M. Adaptive linearization of a loudspeaker. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on* (1991), IEEE, pp. 3589–3592.

[19] GONZÁLEZ, G. G. G., AND NÄSSI, I. M. S. V. Measurements for modelling of wideband nonlinear power amplifiers for wireless communications. *Department of Electrical and Communications Engineering, Helsinki University of Technology* (2004).

[20] GRUHL, D., LU, A., AND BENDER, W. Echo hiding. In *International Workshop on Information Hiding* (1996), Springer, pp. 295–315.

[21] GUPTA, S., MORRIS, D., PATEL, S., AND TAN, D. Soundwave: using the doppler effect to sense gestures. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2012), ACM, pp. 1911–1914.

[22] HAWKSFORD, M. J. Distortion correction in audio power amplifiers. *Journal of the Audio Engineering Society 29*, 1/2 (1981), 27–30.

[23] JAKOBSSON, D., AND LARSSON, M. Modelling and compensation of nonlinear loudspeaker.

[24] JAYARAM, P., RANGANATHA, H., AND ANUPAMA, H. Information hiding using audio steganography–a survey. *The International Journal of Multimedia & Its Applications (IJMA) Vol 3* (2011), 86–96.

[25] KIM, S., HWANG, J., KANG, T., KANG, S., AND SOHN, S. Generation of audible sound with ultrasonic signals through the human body. In *Consumer Electronics (ISCE), 2012 IEEE 16th International Symposium on* (2012), IEEE, pp. 1–3.

[26] KLIPPEL, W. J. Active reduction of nonlinear loudspeaker distortion. In *INTER-NOISE and NOISE-CON Congress and Conference Proceedings* (1999), vol. 1999, Institute of Noise Control Engineering, pp. 1135–1146.

[27] LAZIK, P., AND ROWE, A. Indoor pseudo-ranging of mobile devices using ultrasonic chirps. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems* (2012), ACM, pp. 99–112.

[28] LEE, K.-L., AND MAYER, R. Low-distortion switched-capacitor filter design techniques. *IEEE Journal of Solid-State Circuits 20*, 6 (1985), 1103–1113.

[29] LENHARDT, M. L., SKELLETT, R., WANG, P., AND CLARKE, A. M. Human ultrasonic speech perception. *Science 253*, 5015 (1991), 82–85.

[30] LIN, Q., YANG, L., AND LIU, Y. Tagscreen: Synchronizing social televisions through hidden sound markers. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE* (2017), IEEE, pp. 1–9.

[31] NAKAGAWA, S., OKAMOTO, Y., AND FUJISAKA, Y.-I. Development of a bone-conducted ultrasonic hearing aid for the profoundly sensorineural deaf. *Transactions of Japanese Society for Medical and Biological Engineering 44*, 1 (2006), 184–189.

[32] NANDAKUMAR, R., GOLLAKOTA, S., AND WATSON, N. Contactless sleep apnea detection on smartphones. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services* (2015), ACM, pp. 45–57.

[33] NANDAKUMAR, R., TAKAKUWA, A., KOHNO, T., AND GOLLAKOTA, S. Covertband: Activity information leakage using music. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies 1*, 3 (2017), 87.

[34] NORRIS, E. Parametric transducer and related methods, May 6 2014. US Patent 8,718,297.

[35] OKAMOTO, Y., NAKAGAWA, S., FUJIMOTO, K., AND TONOIKE, M. Intelligibility of bone-conducted ultrasonic speech. *Hearing research 208*, 1 (2005), 107–113.

[36] POMPEI, F. J. *Sound from ultrasound: The parametric array as an audible sound source*. PhD thesis, Massachusetts Institute of Technology, 2002.

[37] ROY, N., HASSANIEH, H., AND CHOUDHURY, R. R. Backdoor: Making microphones hear inaudible sounds. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (2017), ACM.

[38] SELF, D. *Audio power amplifier design handbook*. Taylor & Francis, 2006.

[39] SONG, L., AND MITTAL, P. Inaudible voice commands, 2017.

[40] SUN, Z., PUROHIT, A., BOSE, R., AND ZHANG, P. Spartacus: spatially-aware interaction for mobile devices through energy-efficient audio sensing. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services* (2013), ACM, pp. 263–276.

[41] TITZE, I. R., AND MARTIN, D. W. Principles of voice production. *The Journal of the Acoustical Society of America 104*, 3 (1998), 1148–1148.

[42] VAIDYA, T., ZHANG, Y., SHERR, M., AND SHIELDS, C. Cocaine noodles: Exploiting the gap between human and machine speech recognition. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)* (Washington, D.C., 2015), USENIX Association.

[43] WESTERVELT, P. J. The theory of steady forces caused by sound waves. *The Journal of the Acoustical Society of America 23*, 3 (1951), 312–315.

[44] WESTERVELT, P. J. Scattering of sound by sound. *The Journal of the Acoustical Society of America 29*, 2 (1957), 199–203.

[45] YANG, J., TAN, K.-S., GAN, W.-S., ER, M.-H., AND YAN, Y.-H. Beamwidth control in parametric acoustic array. *Japanese Journal of Applied Physics 44*, 9R (2005), 6817.

[46] YE, H., AND YOUNG, S. High quality voice morphing. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on* (2004), vol. 1, IEEE, pp. I–9.

[47] YONEYAMA, M., FUJIMOTO, J.-I., KAWAMO, Y., AND SASABE, S. The audio spotlight: An application of nonlinear interaction of sound waves to a new type of loudspeaker design. *The Journal of the Acoustical Society of America 73*, 5 (1983), 1532–1536.

[48] ZHANG, G., YAN, C., JI, X., ZHANG, T., ZHANG, T., AND XU, W. Dolphinatack: Inaudible voice commands. *arXiv preprint arXiv:1708.09537* (2017).

[49] ZHANG, G., YAN, C., JI, X., ZHANG, T., ZHANG, T., AND XU, W. Dolphinattack: Inaudible voice commands. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2017), ACM.

PowerMan: An Out-of-Band Management Network for Datacenters using Power Line Communication

Li Chen, Jiacheng Xia, Bairen Yi, Kai Chen
SING Group, Hong Kong University of Science and Technology

Abstract

Management tasks in datacenters are usually executed in-band with the data plane applications, making them susceptible to faults and failures in the data plane. In this paper, we introduce power line communication (PLC) to datacenters as an out-of-band management channel. We design PowerMan, a novel datacenter management network that can be readily built into existing datacenter power systems. With commercially available PLC devices, we implement a small 2-layer PowerMan prototype with 12 servers. Using this real testbed, as well as large-scale simulations, we demonstrate the potential of PowerMan as a management network in terms of performance, reliability, and cost.

1 Introduction

A typical datacenter [29, 65, 67] contains more than thousands of servers, switches, storage units, etc. Datacenter operations and management tasks [42, 52, 85] include device installation, bring-up/restart, configuration, monitoring, diagnostics, and Software Defined Networking (SDN) applications [58], etc. At such scale, delivering management traffic is a critical task.

In existing datacenters, management traffic is usually carried in-band with the data plane traffic. Separate service queues and/or VLANs [11] may be reserved for reliable and timely delivery of management messages. However, this approach introduces *fate sharing* [85] between the data plane traffic and management traffic: failures in data plane network will cut off management traffic to the exact network regions at fault, rendering important and relevant management tasks, such as diagnostics and recovery, impossible.

Therefore, an out-of-band management network (MN) is desirable for datacenter operations. A practical out-of-band MN for datacenters should be:

- **Survivable:** MN should be always available, and should survive faults and failures in the datacenter, in order to perform diagnostic and recovery tasks.
- **Scalable:** MN should be scalable enough to access all the devices in the datacenter.
- **Deployable:** MN should be deployable at low cost, and compatible with existing infrastructure.

Prior proposals do not meet these requirements simultaneously. Out-of-band MNs can be constructed as a parallel electrical network[1] using the same networking equipments as the data plane. To reach all devices, this parallel network needs a port count larger than the data plane network; because this fabric not only accesses all the servers like a data plane network, it also needs to reach the management ports of all the switches and other devices. Thus, the cost is prohibitive to build an parallel high port count electric fabric as a MN.

Non-electrical communication channels in datacenters, such as WiFi [36, 47, 84, 85] and free space optics (FSO) [41, 48], are usually built to accommodate dynamic data plane traffic demands. As out-of-band MNs (parallel to data plane network), deploying them results in significant changes to datacenter infrastructure (e.g., raising the ceiling [48, 84], installing reflective surfaces [36, 41, 48, 84], etc). Furthermore, it is also expensive to build a wireless or FSO fabric that reaches the port count required by MNs with current technologies (§6.3).

We believe, for a datacenter MN, power line communication (PLC) technology is an appealing option. PLC [39], proposed in 1900s [66], allows communication between devices connected by power lines. PLC is known to be challenging [60, 69]. However, over short distances and among limited nodes, current PLC modems for home-use can support Gigabit connections using OFDM [61] in PHY layer and CSMA/CA [33] in the MAC layer, providing Ethernet networking to home appliances, e.g., smart TVs, WiFi extender, home networking, etc. We believe these emerging technologies

[1] Although PLC also uses electrical components and electrical wiring to transmit data, for clarity, we use "electrical network" to refer to the electrical packet switching network [19, 43] in the data plane of current datacenters [65, 67].

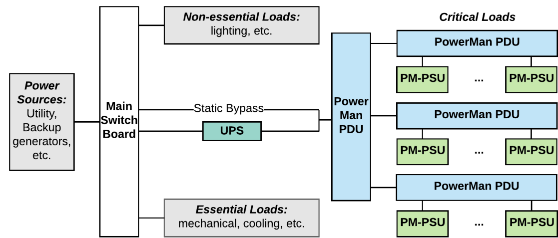


Figure 1: Datacenter Power System with PowerMan

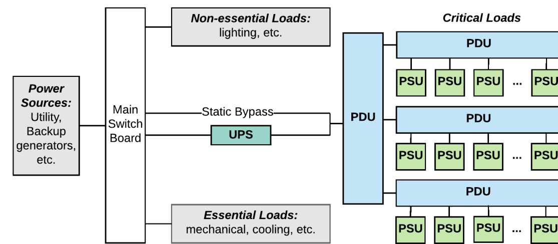


Figure 2: Typical Datacenter Power System (DCPS)

(described by HomePlug standards [24, 56, 81]) open up the opportunity of building a low-cost PLC network with necessary bandwidth and latency for management applications in datacenters.

First of all, a MN using PLC technology naturally meets the survivability requirement, as power system is foundational to any datacenter. A built-in PLC management network is always available, long-term survivable, reaching every device, and independent of the data plane. Furthermore, deployment of PLC network reuses the power system wiring, and requires no change to the existing room designs or floor plan, which is economic.

Yet, one question remains: *can a MN using PLC also meet the other two requirements—being scalable and deployable in existing datacenters?* To answer this question, we build a PLC testbed (§3) with commodity-off-the-shelf (COTS) PLC modems designed for households (using OFDM in PHY layer and CSMA/CA in the MAC layer). Our experiments show that task completion times and user experience of real management applications on PLC network is comparable to that on a Gigabit electrical network. However, we also conclude that a MN directly using COTS PLC devices *cannot* meet the above two requirements: 1) additional in-rack wiring may exceed existing rack designs; 2) due to PLC signal interference, the network can only scale to 6 nodes within a small range (usually 100s of meters [13]) on a single power circuit.

To tackle these problems, we design PowerMan (§4), a MN that can be constructed with existing PLC technology, to support datacenters with more than 10^5 servers. As shown in Figure 1, PowerMan redesigns and replaces two key components in existing datacenter power systems (Figure 2): a power supply unit (PSU) for servers and switches, and a power distribution unit (PDU).

- **PowerMan PSU** lowers the wiring complexity by increasing the integration level. It combines a normal PSU module with a PLC modem module, and acts as a network interface to the server OS. Using PowerMan PSU, PLC network can be deployed easily in the current server racks.
- **PowerMan PDU** addresses the scalability issue. Due to available carrier frequencies and signal quality constraints, COTS PLC modems designed for home-use only support communication within 64 nodes in the same PLC LAN (PLAN) within limited range (§2.2). To scale beyond this, PowerMan PDU eliminates signal interference on the boundary of PLANs using low pass filters, enabling the reuse of the same carrier frequencies across different PLANs. By connecting multiple PLANs into a tree topology, we can scale the PLC network to reach potentially $>10^5$ servers, providing datacenter-wide coverage.

We have implemented a 2-layer PowerMan prototype (§5) connecting 12 servers across two racks. The prototype is built with existing PLC technology in an academic datacenter without any modification to the existing infrastructure, e.g., room plan, power line wiring, and ceiling height. We demonstrate the potential of PowerMan as a datacenter MN (in terms of performance (§6.1), availability (§6.2), and cost (§6.3)) by running real management applications in our small testbed as well as large-scale simulations. Our key contributions are:

- We introduce PLC as an out-of-band channel for datacenter MN. To validate the idea, we build a real small-scale PLC testbed to quantify the throughput, latency, and packet loss conditions for management applications (§3). We find that, due to various sources of interference in datacenter [60], PLC testbed exhibits lower performance than advertised (e.g. $\leq$50Mbps TCP throughput (measured) v.s. 1000Mbps PHY bit-rate (advertised)). We further expose the wiring complexity and scalability issues that cannot be addressed with existing PLC devices.
- We design PowerMan to address the wiring complexity and scalability problems identified above. We validate the design by implementing a PowerMan prototype. On the prototype, our experiments with production traces show $<$24ms average flow completion time (FCT) and $>$10Mbps throughput for the 1-to-N/N-to-1 management traffic patterns. Experiments with real management tasks show that, compared to a Gigabit electrical network, the completion times of all tasks are prolonged by $<$40.62% on PowerMan, with a minimum of 1.57% (66.43s $\rightarrow$ 67.47s) for a Human-in-the-Loop task, and a maximum of 40.62% (32ms $\rightarrow$ 45ms) for a SDN task. We also confirm PowerMan's utility at large scale with simulations, and find that for a PowerMan with 120K servers, the round-trip

time (RTT) for management applications is ∼40ms.

- Cost comparisons with other technologies show that, apart from saving infrastructure modification costs, PowerMan can be constructed with low equipment cost ($1/2 \sim 1/3$ of the cost of related designs at the same scale). PowerMan is also power-efficient in operation: its power consumption is 6%∼9% of other technologies.

Caveat: PowerMan is suitable for many management applications given its performance characteristics. However, we acknowledge that, for some applications, delivering control messages with low latency is crucial: fine-grained load balancing [62, 82] and flow scheduling [20, 30] need to configure data plane on millisecond time scale. *PowerMan alone is not suitable for such applications.* For them, we suggest dual-homing the controller with access to both the data plane network and PowerMan network. Latency-sensitive traffic can use the fast data plane, while PowerMan can serve other management applications. We believe PowerMan is also valuable as a back-up/diagnostic network to fall-back on in case of failures.

2 Background

2.1 Power System in Datacenters

The power system [35, 44] is the most fundamental system in a datacenter. A typical DCPS is shown in Figure 2, and it is composed of:

- The main switch board (MSB) directs electricity from one or more sources of supply to several smaller regions of usage. It feeds into all loads in the datacenter.
- The uninterruptible power supply (UPS) provides consistent power to critical loads without interruption. It contains energy storage, which supplies power to the load when the utility power is down.
- A power distribution unit (PDU) is an electrical distribution device, and it can be free-standing or rack-mounted. The PDU houses circuit breakers that are used to create multiple branch circuits from a single feeder circuit, and can also contain transformers, surge protection devices, and power monitoring/controls.
- A power supply unit (PSU) rectifies AC power from the connected PDU to DC power. For reliability, critical servers and switches are usually equipped with two PSUs in case of failure.

DCPS is often classified as belonging to "Tier I-IV" [71] depending on the power distribution, UPS, redundancy, etc. [15, 18] For example, in a Tier-III DCPS [35, 44], each critical load device has two power distribution paths (including redundancy components), and the power system in Figure 2 is replicated for each PSU. In what follows, for clarity, our design of PLC networking is limited to the primary power system depicted in Figure 2 by default, and we discuss how all tiers of DCPS can adopt PowerMan in §4.4.

2.2 Power Line Communication

PLC uses electrical wires to simultaneously carry high frequency data signals and 50∼60Hz AC power transmission. PLC has been widely used in power systems for protection, telemetering, and industrial control applications [39, 40, 45, 83] with data rate of a few Kbps.

In recent years, we witness a rapid growth of PLC appliances for home networking, due to its ubiquity (home power systems provide sockets in every room) and ease of deployment (no new wire needed). The home networking market drives PLC technology to reach higher bandwidth, in order to support popular use cases such as broadband Internet access, video streaming, gaming, etc. Through standardization efforts from the US HomePlug Powerline Alliance and European Home System Consortium, vendors have converged to use Orthogonal Frequency Division Multiplexing (OFDM) [61] as the modulation scheme in PHY layer, and CSMA/CA [33] as the MAC layer protocol. Adopting the HomePlug protocols (HomePlug 1.0 [56]/AV [24]/AV2 [81]), PLC modems and adapters can form a communication network providing Ethernet connectivity to TVs, gaming console, and PCs. Currently, many vendors offer PLC modems with up to 1200Mbps PHY layer bit-rate [8, 16, 17], and up to 64 devices in one PLC network [13, 16, 17].

In academia, there have been continuous efforts in PHY [60, 69] and MAC [74, 75, 76, 77, 78] layers for PLC to achieve higher throughput and lower latency. Orthogonal to prior work, we focus on the application of PLC in the context of datacenter MN. We believe PLC is a suitable candidate for MN as a built-in communication network in DCPS for the following reasons:

- Power system is the last-to-fail system in datacenters, and is independent of the data plane network. A MN within the power system therefore can survive data plane failures, and is ready for immediate diagnosis and recovery.
- Power system reaches every device in datacenters, providing full visibility for management applications.
- PLC reuses the wires in the power system, and there is no need to change the existing room plans, ceiling height, and rack dimensions. This compatibility with existing datacenter designs greatly reduces the deployment cost.

In the following, leveraging the technology advances in household PLC appliances, we are motivated to: 1) understand performance characteristics of PLC networks (§3), 2) expand PLC network from home-scale to datacenter-scale (§4), and 3) evaluate our design for datacenter MN using PLC (§6).

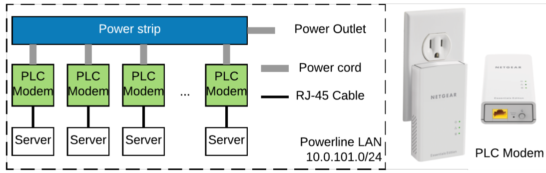


Figure 3: COTS PLC Testbed & PLC Modem

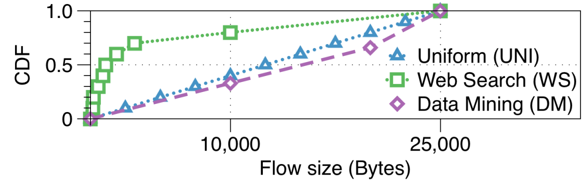


Figure 4: Flow Size Distributions

3 Building a PLC Testbed

In this section, we first build a small-scale PLC testbed. Then, we perform a series of experiments and measurements to learn the communication characteristics of the PLC channel in a datacenter environment.

3.1 Building a PLC Testbed

We describe the devices we use and how they are connected to build the testbed.

Server & switch: We use Huawei FusionServer RH1288 with Intel E5-2630 and 64GB memory (1 Rack Unit). Each server has a NetXtreme BCM5719 Ethernet Network interface card (NIC) with 4×1 GbE ports. The servers are all connected to a Gigabit Ethernet switch via their first Ethernet interface (Eth0).

PLC modem: We obtained 16 Netgear Powerline 1000 (PL1000) PLC modems (US$ 30.3 per piece) via local home appliance vendors. As in Figure 3, each modem has one built-in power plug and one RJ-45 port for Ethernet connection. The max power consumption of PL1000 is 3.73 watts (0.49 watts in standby mode). It is compatible with HomePlug AV protocols. For OFDM, it uses frequencies in the range from 2 MHz to 86 MHz.

PDU: The rack-mounted servers and Ethernet switch are plugged into the in-rack PDU with no empty sockets. We use a separate Thomson TM-EC6 8-socket power extension cord for PLC modems.

Interconnection & wiring: We connect the PLC modems to the power extension cord, and then plug them into the power outlets on the in-rack PDU. Each server is connected to one PLC modem via its second Ethernet interface (Eth1).

In summary, as shown in Figure 3, we build a PLC testbed using commodity components. Each server is both connected to an electrical ToR Ethernet switch via Eth0, and to a PLC modem via Eth1. These modems are connected via a power strip, forming a PLC network.

Through building the testbed, we identify the first difficulty for practical deployment of PLC networking in datacenters: wiring. This is because each of these external PLC modems requires an additional power socket slot and a network cable, resulting in 2× socket count on the in-rack PDU and 1.5× space for cabling. As the current rack design does not anticipate the usage of PLC devices, we find it difficult to organize the additional cables, and the PLC modems have to be attached to a power extension cord. We will address this in §4.1.

Pattern	CDF	AFCT-us	99% FCT-us	Thruput-Mbps	Pkt Loss%
1-to-1	DM	3887 (233)	8631 (327)	48.15 (484.56)	0.00% (0.00%)
1-to-5	DM	12914 (686)	29552 (1146)	35.39 (791.49)	0.13% (0.00%)
5-to-1	DM	16429 (606)	43210 (944)	33.52 (931.06)	0.12% (0.00%)
1-to-1	UNI	3972 (223)	8686 (331)	25.00 (444.11)	0.00% (0.00%)
1-to-5	UNI	11590 (618)	26798 (1143)	30.48 (763.02)	0.13% (0.00%)
5-to-1	UNI	15728 (532)	39639 (870)	31.13 (928.74)	0.13% (0.00%)
1-to-1	WS	2895 (187)	7900 (321)	13.11 (202.57)	0.00% (0.00%)
1-to-5	WS	9234 (337)	31049 (1117)	13.98 (522.98)	0.23% (0.00%)
5-to-1	WS	11021 (296)	36435 (618)	17.00 (635.87)	0.17% (0.00%)

Table 1: Measurements of Synthetic Traffic on PLC Testbed. The results of a gigabit electrical network are shown in the parentheses.

3.2 Testbed Experiments

Next, we measure its performance using both synthetic traffic and real management applications.

3.2.1 Scalability

We first investigate how many PLC modems can coexist in a PLC network. We add PLC modems to the power strip one by one (IP addresses and subnet masks are assigned beforehand), and then monitor the indicator lights on the modems for successful connections. Finally, we verify the connection on the servers via `ping` utility. We observe that the network can accommodate at most 6 PLC modems. When there are more than 6 modems in the network, the first 6 modems are connected.

3.2.2 Experiments with Production Traces

Setting: For the flow size, we adopt 2 realistic flow size distributions used in prior work [21, 22, 25, 43]: one from a web search cluster [21] and the other from a data mining cluster [43], respectively. We also include a uniform distribution for reference. All distributions, shown in Figure 4 are capped at 25KB, as we are mainly interested in management applications, which tend to have shorter flow sizes.

We use the following traffic patterns:

- **1-to-N:** This pattern occurs in management applications where a master pushes configurations to slaves.
- **N-to-1:** This pattern occurs in monitoring applications where a server collects statistics from clients.

Among the 6 connected servers, we create traffic patterns using a traffic generator [6], which is a client/server application for generating user-defined traffic. The server listens for incoming requests on the specified ports, and replies with a flow with the requested size for each request. The client connects to a list of servers, and generates requests to randomly chosen servers. For each

request, it samples from the input request size and fanout distributions to determine the request size and how many flows to generate in parallel for the request. All packets use the same default priority.

In each experiment, we use a different combination of patterns and distributions, and each client generates 25K requests. We measure the flow completion time (FCT), throughput, and packet loss rate for each flow, and Table 1 summarizes the results. The average round-trip time (RTT, grouped by traffic patterns) is shown in Figure 5. We then repeat the experiments using the Gigabit electrical network (via Eth0), and the results are shown in the parentheses in Table 1.

Results: We make the following observations:

- **Latency:** The average FCT on PLC testbed is around 2 order-of-magnitude (OoM) larger than that on the electrical network. We observe the same trend for 99th percentile FCT. For RTT, the smallest one (2.2ms, from 1-to-1 pattern) is also around 2 OoM larger (∼20us on the electrical network).
- **Throughput:**The advertised 1000Mbps bandwidth is in fact the maximum PHY bit-rate, and we cannot obtain more than 50Mbps TCP throughput on the testbed, which matches field-tested results [23]. The throughput is about 1 OoM less than that on the electrical network.
- **Packet loss rate:** The packet loss rates are less than 0.5% for PLC testbed across all cases, while the electrical network shows near-zero packet loss rates.

Implications: As expected, the PLC testbed we constructed shows much lower throughput and longer latency compared to the Gigabit electrical network. This is because PLC is an "extremely harsh environment" [60] for the high-bandwidth, high-frequency communication signals, as critical channel parameters (e.g., noise, impedance, and attenuation) are highly unpredictable and varied with time, frequency and location [69]. As a result, the PLC network is clearly inappropriate for time-critical tasks (e.g., fine-grained load balancing [62, 82] and flow scheduling [20, 30]).

However, the PLC testbed is shown to deliver <10ms average FCT and >10Mbps throughput for N-to-1/1-to-N patterns, which are common for management applications. Thus, for latency-insensitive management tasks (e.g., device installation, bring-up/restart, configuration, monitoring, diagnostics, etc), the PLC network remains attractive, due to its other benefits such as survivability in case of data plane failure, compatibility with existing datacenter design, and economy.

Therefore, we proceed to evaluate the end-to-end application performance of the PLC testbed with latency-insensitive management tasks.

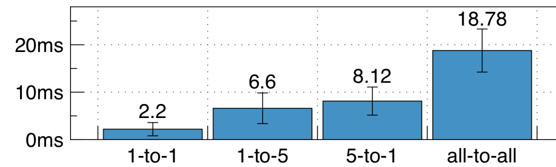


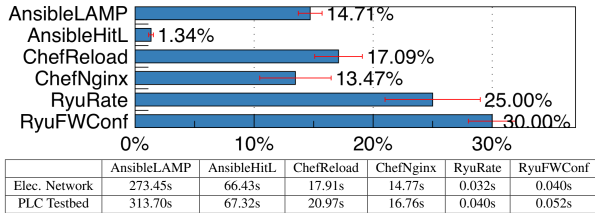
Figure 5: RTT measurements on PLC testbed

3.2.3 Experiments with Management Applications

Setting: Based on the communication model, we choose two management platforms for on-premise datacenters and cloud virtual clusters: push-based Ansible [1]), and pull-based Chef [3]. As SDN is an important class of applications, we also include a SDN controller, Ryu [12]. We perform tasks with real usage scenarios.

- **Ansible [1]** is an automation engine for clusters. Ansible is push-based and agentless: from the master node, it manages slave nodes through SSH connections. We deploy Ansible 1.7.2 on our testbed, and perform one automated task and one Human-in-the-Loop (HitL) task:
  - **AnsibleLAMP**: An automated LAMP deployment with two web severs, two load balancers, and two database servers. The playbook is based on Ansible official examples [2].
  - **AnsibleHitL**: A HitL setting with an operator checking configurations of servers. Via Ansible ad-hoc commands [7], in each experiment, the operator sequentially executes `df`, `route`, and `lsmod` on all servers.
- **Chef [3]** is an automation platform for cluster management. Chef is a pull-based: clients poll a centralized master periodically for updates. On our testbed, we install a Chef Server 12.11 in standalone mode on one of the servers, and the rest are installed with Chef Client 12.17. We perform two automated tasks described by Chef cookbooks.
  - **ChefReload**: This cookbook [4] automatically reloads the Apache service on all servers.
  - **ChefNginx**: This cookbook [10] automatically distributes the install file (889KB), installs, and configures nginx [9] 1.10.2 on all servers.
- **Ryu [12]** is a SDN framework. It can be integrated with OpenStack Neutron [5] for SDN applications. We installed OpenVSwitch 2.5.1 [63] on all servers and a Ryu 3.26 controller on one of them. We run two tasks in official documentation [14]:
  - **RyuRate**: We use `curl` to query the Ryu controller via its RESTful API, and the controller replies with the current rates of all ports.
  - **RyuFWConf**: We add a firewall rule via RESTful API, and the Ryu controller replies with the result.

We run the above 6 management tasks (each for 10 times) and measure their completion times with millisecond precision on both the PLC network and Gigabit



	AnsibleLAMP	AnsibleHitL	ChefReload	ChefNginx	RyuRate	RyuFWConf
Elec. Network	273.45s	66.43s	17.91s	14.77s	0.032s	0.040s
PLC Testbed	313.70s	67.32s	20.97s	16.76s	0.040s	0.052s

Figure 6: Management Applications on PLC testbed

electrical network. We use percentage increase in completion time as the metric: for a task, denote its completion time on Gigabit electrical network as T_e and on PLC testbed T_p, the metric is defined as: $\frac{T_p - T_e}{T_e} \times 100\%$.

Results: In Figure 6, we observe encouraging application performance delivered by the PLC network. Overall, we find that using PLC results in less than 30% increase in completion time compared to the Gigabit electrical network for all the tasks, and this increase is mainly due to the latency introduced by the PLC network. In the best case, we notice that, for AnsibleHitL task, the PLC network performs almost the same as the electrical network (only 1.34% longer). This is because human response time is the main contributor of latency in this task. In the worst case, the completion time of RyuFWConf is increased by 30%, which is because it performs only HTTP query/response and network latency contributes the most to the completion time. In summary, our results of end-to-end application performance on PLC network is promising for latency-insensitive management tasks.

3.3 Lessons Learnt

We conclude: 1) It is possible to use commodity PLC modems to form a PLC network that provides Ethernet connectivity for all connected servers. 2) PLC performance is promising for management applications: it provides <10ms average FCT and >10Mbps throughput for N-to-1/1-to-N patterns, which are common for management applications; the management tasks also have similar user experience. 3) This PLC network, however, cannot be directly used in real datacenters due to the deployability (wiring) and scalability problems.

Therefore, we are motivated to tackle the wiring and scalability issues, so that the PLC technology can be deployed in real datacenters.

4 PowerMan Design

To tackle the wiring and scalability issues, we design PowerMan. To ensure deployability, our guiding principle is to respect the existing datacenter designs, and preserve the floor plan, room design, rack dimensions, and power line wiring. To this end, PowerMan only replaces two types of components in existing DCPS: PSU and PDU.

4.1 Power Supply Unit (PSU)

In the PLC testbed, each server needs two network cables: one for data plane connectivity, and one for PLC modem to access MN. It also needs two power sockets: one for its PSU, and one for the PLC modem. Thus, a PLC MN requires 1.5× cable space and 2× power sockets per rack. Given a rack hosts 20∼40 servers [65, 67], it is infeasible to accommodate this additional wiring with existing rack design. To address the problem, we design a novel PSU for rack-mounted servers and switches in datacenters (Figure 7). The key idea is to increase the integration level in the PSU to reduce external wiring.

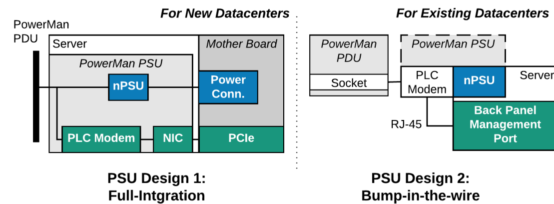


Figure 7: PowerMan PSU

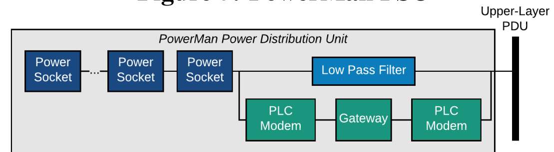


Figure 8: PowerMan PDU

We present two designs of PowerMan PSU to fit different deployment scenarios: Full-Integration and Bump-in-the-Wire. Full-Integration is designed for new installation of datacenters, as the datacenter operator has the freedom to customize the hardware configuration of each server/switch. We combine a normal PSU module (nPSU in Figure 7), a PLC modem module, and a network interface module in the PowerMan PSU. It connects to the mainboard of the server via a PCIe interface, and appears as another NIC to the OS, which allows users to use familiar networking stack to access the PLC network.

The Bump-in-the-Wire design is for incremental deployment in existing datacenters, and it leverages the integrated NIC on the mainboard of rack-mounted servers, which is exposed as the management port on the server back panel. The PLC modem attaches to the PSU externally, and acts as a "bump" in the power cable from the PDU socket to the PSU. The power to the server is fed into its PSU through the PLC modem via a bypass circuit. The PLC modem connects to the management port via a RJ-45 network cable, so that the integrated NIC can access the PLC network. This network cable travels a short distance from the power port to the management port on the back panel, and thus does not tangle with other in-rack cables.

Via PowerMan PSU, a server can connect to a PLC MN without complicated wiring and additional power sockets, thus is compatible with the design and dimensions of the current racks.

4.2 Power Distribution Unit (PDU)

The scale of PLC network on our testbed is limited to 6 nodes. We refer the PLC network within the PDU as PLC LAN (PLAN). Manufacturers of more advanced models claim that the scale can be as large as 64 nodes [13, 16, 17]. However, this is still too small for production datacenters [65, 67]. The main reason for such limited scalability is that these devices are designed for home-use, where network size is not the main concern.

To scale, we design a novel rack PDU (Figure 8). The key idea is to remove cross-PDU PLC signal interference but maintains network connectivity. PowerMan PDU achieves this with two main components, a low pass filter (LPF) and a PLC gateway.

We keep the circuit of a normal PDU, and add a LPF between the circuit and the external power line. Since the OFDM frequencies used in the PLC modem is $\geq$ 1.8MHz [60] and the AC power frequency is 50$\sim$60Hz, a LPF with appropriate cut-off frequency (between 60Hz to 1.8MHz) can greatly attenuate the outgoing and incoming high frequency PLC signals, thus effectively eliminating the interference from/to other PLANs.

While the PLC signals are mostly eliminated across the LPF, the network connectivity is preserved using a PLC gateway. The PLC gateway consists of a packet-forwarding hardware gateway and two PLC modems. One modem is connected to the PLC network inside the PDU, and the other is connected to the PLC network on the external, upper-layer PDU. The PLC gateway is therefore connecting the PDU's PLAN and the upper-layer PDU's PLAN by forwarding packets between them, with no PLC signal interference.

PowerMan PDU replaces the rack PDU and retains the same cable and socket count. It acts as a switch for the PLC network devices on the same rack.

4.3 Interconnection & Scalability

With the new PowerMan PDUs developed, we can now connect them and scale the PLC network to support real datacenters. We leverage DCPS to interconnect the PLC devices. Since PDUs in DCPS are connected in a tree topology (Figure 2), we also choose to use the same topology to scale. Other topologies (e.g. ring, mesh, hypercube, etc.) requires changing the wiring of the power system. Take ring topology as an example, each PDU connects to more than one other PDUs, requiring an additional power cable for each PDU. Other topologies also requires a different power allocation scheme, both inside the PDU and across PDUs.

As shown in Figure 9, we construct the PowerMan PDUs into a $(k-1)$-ary tree topology, where k is the number of PLC devices supported in a PLAN. For our current PLC modems, k=6; up to k=64 have been reported for other COTS PLC modem models [13]. With height h, this topology can connect $(k-1)^h$ PowerMan PSUs. With a tree height of h=3 and k=64, 250K PSUs can be connected.

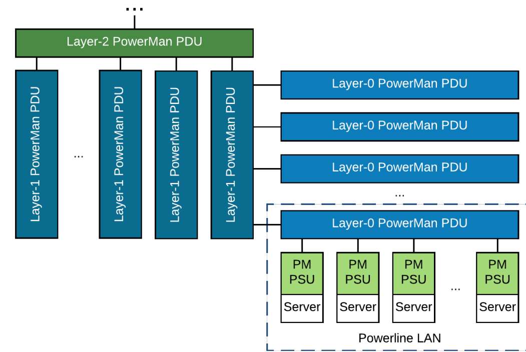


Figure 9: Scaling PLC with PowerMan

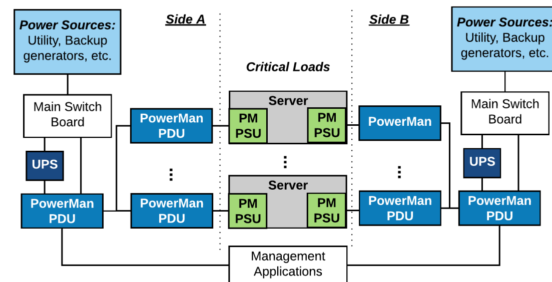


Figure 10: PowerMan Fault-Tolerance: Example of Tier-III DCPS with AB Dual-Bus [44]

4.4 Fault-tolerance

PowerMan leverages the redundancy in existing DCPS to achieve high availability. As mentioned in §2, DCPS can be classified into 4 tiers [71], and all can be integrated with PowerMan.

- **Tier-I** DCPS have a single path for power distribution without redundant components, and PowerMan can be integrated as in Figure 9.
- **Tier-II** adds redundant components to this design (N + 1), improving availability, and PowerMan can be integrated into the main distribution path as for Tier-I DCPS, the PDUs in the redundant components should also be replaced with PowerMan PDUs.
- **Tier-III** datacenters have one active and one alternate distribution path for utilities. Each path has redundant components and are concurrently maintainable, providing redundancy during maintenance. PowerMan can be integrated into both distribution paths. As an example, Figure 10 showcases how PowerMan can be integrated with Tier-III DCPS [44]. This architecture is configured with two sides, A and B. Each side can include multiple UPSs, and either side can handle 100% load. If one side has a problem, the load

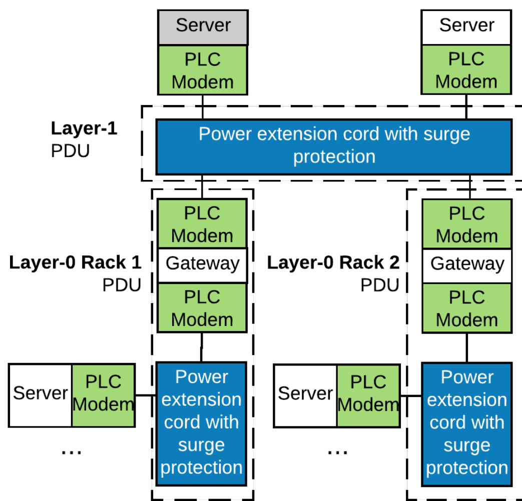


Figure 11: Two-Layer PowerMan Prototype

is automatically switched to the other side. Automatic load transfer switches can reside upstream of the UPS for maintenance isolation purpose. This design ensures a high level of system availability even during maintenance or component failure. PowerMan can therefore be replicated in both sides; the controller node where management applications are located should also connect to the root PowerMan PDUs on both sides, so that when failure (either PowerMan or DCPS) in either side happens, management applications can still access the servers and switches.

- **Tier-IV** DCPS have two simultaneously active power distribution paths, redundant components in each path, and are supposed to tolerate any single equipment failure without impacting the load. PowerMan can be integrated in the same way as Tier-III.

Embedded in DCPS, PowerMan share the redundancy and availability mechanisms, thus is expected survive even partial power outages in Tier-II (or higher) DCPS.

5 Prototype Implementation

We implement a PowerMan prototype to validate the design, and its schematic is shown in Figure 11. We have not yet constructed a PowerMan PSU that can be fit into our rack-mounted servers, but its functionality can be emulated using the same setting as §3.1: each server connects to a PLC modem via one of its NIC ports (Eth1).

PDU has two components: a LPF and a PLC gateway. For the LPF, instead of implementing LPF circuits and installing them on the power extension cords, we identify that power extension cords with surge protection can serve as low cost alternatives[2]. This is because surge protector removes voltage spikes and high frequency noise. We note that the use of surge protector as LPF is only for prototyping, and real deployment of PowerMan should use properly designed LPF in the PDU. We obtained Targus SmartSurge 6 power extension cords from local vendors, and our testing shows that two PLC modems cannot establish connection across two such cords, which indicates that they have the correct cut-off frequency. In this way, the PLC modems can form a PLAN within the power extension cord that they are attached to, without interference of PLC signals from other PLANs.

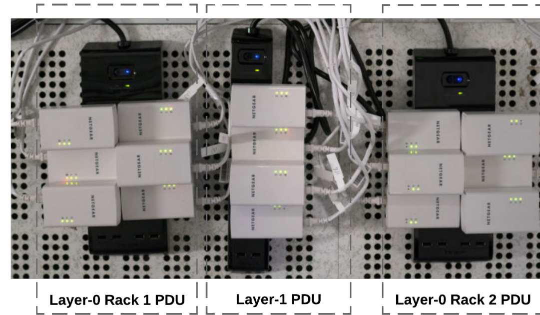


Figure 12: PLC Network Components of PowerMan prototype

Next, we implement the PLC gateway with two PLC modems and a rack-mounted server. The server connects to the two modems via Eth1 and Eth2 ports. The modems are attached to different power extension cords with surge protection. Therefore, their signals are isolated, and can only propagate within their own PLANs. With routing rules correctly configured, the server acts as a packet forwarding gateway between the two PLANs.

We construct 3 prototype PowerMan PDUs, which form a tree topology with 2 layers, as shown in Figure 11&12. In Layer-0, the prototype has two racks, and each rack forms a PLAN on its own PDU. The two Layer-0 racks are connected to a Layer-1 PLAN via their PLC gateways. In addition to the gateways of the two racks, we connect another two servers to act as gateways on the Layer-1 PLAN. The routing tables and IP addresses are properly configured in all the servers and gateways, so that each server can reach all the other servers on this PowerMan PLC network.

6 Evaluation

In this section, we evaluate three aspects of PowerMan: performance, reliability, and cost.

Summary of results:

- Experiments with production traces show <24ms average FCT and >10Mbps throughput for 1-to-N/N-to-1 traffic patterns.

[2]This choice is inspired by the product FAQ [13] from the vendor of our PLC modem. The FAQ advices against the usage of surge protectors with the PLC modems, because surge protector may remove high frequency signals.

- Experiments with real management applications demonstrate that, compared to the Gigabit Etherent, the completion times of all tasks are only prolonged by <40.62% on PowerMan.
- By simulating a year of operation, PowerMan is shown to achieve >99.9977% availability (leveraging the redundancy in DCPS) at the scale of 250K servers.
- Apart from saving infrastructure modification costs, PowerMan can be constructed with low initial cost (1/2∼1/3 of the cost of other technologies at the same scale), with 6%∼9% operating power usage.

6.1 Performance

6.1.1 Prototype Experiments

On the PowerMan prototype, we perform the same set of experiments as in §3.2.

Experiments with Production Traces: In addition to the setting in §3.2.2, our experiments here include another parameter: distance, which refers to the number of PLC gateways (i.e., hops) between the servers and clients. For example, for 1-to-5 pattern with distance=1, a client in Rack 1 will only send requests to the traffic generator server hosted in gateways in Layer-1 PLAN[3]. To understand this parameter in real PowerMan deployments, for a controller node connected to the root with tree height h=3, its distance to all PSUs is merely 2. We summarize the results from the experiments on the prototype in Table 13. We make the following observations.

- **Latency:** Compared to Table 1, we see on average 3.04ms increase in FCT if distance increases by 1, and 4.13ms if distance increases by 2. This corresponds to our RTT measurements on the prototype in Figure 14: when distance increases from 0 to 1, the RTT increases on average 2.19ms, and 2.92ms from 1 to 2.
- **Throughput:** Increasing distance by 1 (2) decreases the throughput by 3.27Mbps (6.80Mbps) on average. Still, the prototype provides >10Mbps for 1-to-N/N-to-1 patterns.
- **Packet loss:** interestingly, increasing distance lowers the packet loss rate: 1 (2) increase in distance decreases the packet loss rate by 0.05% (0.05%) on average. This is because the inter-PLAN flows converge at the gateway, and from there, are forwarded to their destinations. This store-and-forward behavior for flows across PLAN results in lower packet loss rate compared to the flows within a PLAN running CSMA/CA.

In summary, PowerMan prototype demonstrates <24ms average FCT and >10Mbps throughput for common management application traffic patterns (1-to-N/N-to-1) for distance=2. This indicates that, a PowerMan with tree height h=3 can support management applications with

[3]The setting for the results in Table 1 can be considered as distance=0 (within the same PLAN).

Pattern	CDF	Distance	AFCT (us)	99% FCT (us)	Thruput (Mbps)	Pkt Loss%
1-to-1	DM	1	7963	15671	32.54	0.00%
1-to-1	DM	2	13856	26245	31.35	0.01%
1-to-5	DM	1	14736	30747	27.52	0.04%
1-to-5	DM	2	19701	39326	24.55	0.03%
5-to-1	DM	1	17418	38063	31.93	0.04%
5-to-1	DM	2	23046	48150	23.89	0.02%
1-to-1	UNI	1	7529	16575	13.19	0.01%
1-to-1	UNI	2	11841	24255	8.39	0.01%
1-to-5	UNI	1	14231	31086	26.22	0.06%
1-to-5	UNI	2	19715	41289	20.46	0.03%
5-to-1	UNI	1	17148	40939	28.29	0.05%
5-to-1	UNI	2	21833	47630	22.25	0.03%
1-to-1	WS	1	5825	15648	6.52	0.02%
1-to-1	WS	2	9601	25792	3.96	0.05%
1-to-5	WS	1	11574	33556	12.19	0.14%
1-to-5	WS	2	14657	38995	10.56	0.07%
5-to-1	WS	1	11783	33270	15.62	0.06%
5-to-1	WS	2	15561	40009	12.13	0.04%

Figure 13: Measurements of trace-based experiments on PowerMan prototype

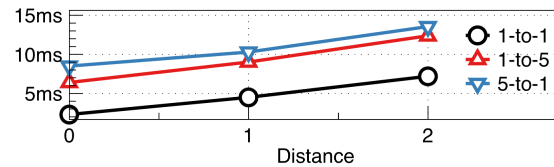
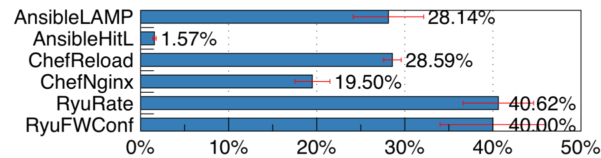


Figure 14: RTT on PowerMan prototype



	AnsibleLAMP	AnsibleHitL	ChefReload	ChefNginx	RyuRate	RyuFWConf
Elec. Network	273.45s	66.43s	17.91s	14.77s	0.032s	0.040s
PowerMan	350.39s	67.47s	23.03s	17.65s	0.045s	0.056s

Figure 15: Management Applications on PowerMan prototype

reasonable latency[4] and throughput.

Experiments with Real Management Applications: Next, we evaluate the end-to-end applications performance. We perform the tasks in §3.2.2 again on both PowerMan prototype and the Gigabit electrical network. We scale the set of tasks in §3.2.3 so that they can cover all 10 servers in the testbed. For example, the AnsibleLAMP task now configures 4 web servers, 2 load balancers, and 4 database servers. We assign one of the gateway server in Layer-1 PLAN as the master node for Ansible, Chef, and Ryu, which is the darkened gateway in Figure 11. We plot the results in Figure 15.

As expected, due to the need of traversing one PLC gateway, the completion times increase for all the tasks. Among them, for AnsibleHitL, the PLC network performs almost the same with the electrical network (only 1.57% slower) as distance increases.Also, as explained in §3.2.2, network latency dominates the completion times of the two Ryu tasks, so their metrics increase the most, i.e., 40.62% and 40% respectively. Furthermore, using PowerMan results in <30% increase in comple-

[4]We consider soft real-time constraints for interactive systems, e.g. 300ms [28, 72], are reasonable latency targets.

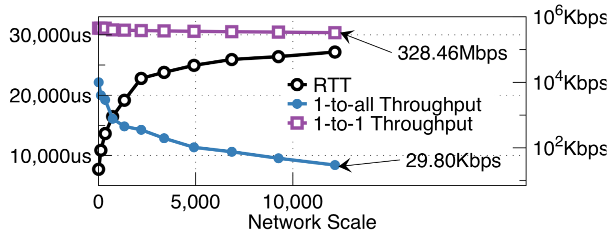


Figure 16: Large Scale Simulations: RTT and throughput

tion times for Chef tasks. Overall, adding one more tier in PowerMan prototype compared to §3's testbed results in less than 5% increase in task completion time.

6.1.2 Large-scale Simulations

The current prototype is still too small to reveal PowerMan's performance in actual deployments. Using ns-3 [50] simulator, we perform simulations at the scale of real datacenters [65, 67] to infer the user experience of PowerMan in actual deployments.

Setting: Since each PowerMan PDU corresponds to a PLAN that uses CSMA/CA (§2.2), we simulate PLAN using the CSMA network implementation in ns-3 with parameters in [73, 79]. We interconnect PLANs with point-to-point links, which corresponds to the PLC gateways in our design. We assume that a controller connects to the root of the PowerMan tree topology with a 1Gbps network interface. We fix the tree height h=3, so the distance from the controller to every PSU is 2. We first tune the parameters to fit the results in Figure 5&14, so that the RTT within a PLAN is 8ms and the latency across a point-to-point link is 3ms. Then we run the simulations for different scale of the network (number of servers) from 125 (k=6) to 12167 (k=24).

Results: We create 1-to-1 and 1-to-N patterns from the controller using TCP connections, and measure the RTT and throughput per-server for different network scales. The results are plotted in Figure 16. For 1-to-1 connection from the controller to a server, we observe consistent throughput >328.64Mbps. For 1-to-N pattern, we create connections from the controller to all servers, and see that the per server throughput quickly drops as more and more servers shares the out-going bandwidth of the controller. At maximum network scale (12167), the per-server throughput is 29.8Kbps. For latency, we observe that the average RTT is smaller than 40ms even when network scales beyond 10^5 servers. This is as expected as the overall distance is only 2.

6.2 Availability

We use availability to characterize the system reliability of PowerMan, which is the percentage of reachable servers. The key component in PowerMan PDU and PSU is the PLC modem, so we model availability of the entire system at the resolution of an PLC modem. We use a Poisson process [27] to characterize the failure process of

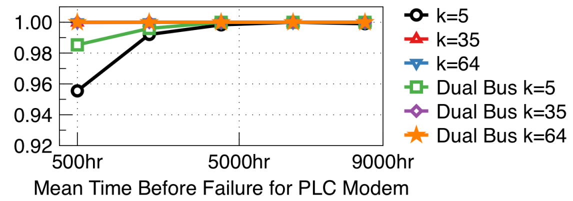


Figure 17: Availability of PowerMan

a PLC modem. The Mean Time Before Failure (MTBF) is the key metric in this model, and it is a common measure of reliability of a hardware component [27, 53]. A higher MTBF means the component is more reliable. Our PLC testbed and prototype have been running for 2 months without failure (1440 hours); using this as reference, we vary the MTBF of PLC modem from 500 to 9000 hours. The MBTF of packet forwarding gateway is assumed to be 3000 hours [37].

We implement an event-driven failure simulation, modeling the entire network of PLC modems in PowerMan. In each run, we vary the scale of network (by increasing k from 5 to 64, h=3), the MTBF of PLC modem (from 1000 to 9000), and simulate a year of PowerMan operation with the failure model describe above. We plot the average availability of PowerMan in Figure 17. We observe that PowerMan is highly available at large scale: For k=64 (network scale is 250K), the availability is 99.9943% (using the least reliable modem with MBTF=500hrs). High availability provides consistent global visibility to management applications, allowing them to perform monitoring and diagnostic tasks.

In Figure 17, we also plot the availability of a PowerMan in a DCPS with Dual Bus redundancy as shown in Figure 10. In this setting, we have PLC networks replicated in both sides, and the controller attaches to the roots of both trees. We can see that, by integrating with the redundant power systems, PowerMan can achieve higher availability for varying network scales.

Since PowerMan is embedded in DCPS, servicing/replacing components is similar to that in a typical DCPS. Tier II-IV DCPSs are designed with redundancy (§4.4), so when parts of the system fail, the operations can continue, as back-up units will take over. In the meantime, faulty components can be repaired/replaced. PowerMan adopts the same recovery strategy.

6.3 Cost Comparisons

Next we compare the construction, equipment, and operational costs of PowerMan and other related designs that can be used as out-of-band MNs. The comparison is done at the same scale of 16000 servers. We compare with these proposals for datacenters: 3D-Beamforming (3DBF) [84], Firefly [48], Diamond [36], and Fat-Tree [19]. We emphasize that this is not a direct comparison: these designs are complete datacenter networks with both data plane and in-band control plane, and we

Components	FatTree	3D-Beamforming	Firefly	Diamond	PowerMan
NIC (k$)	80	80	80	240	80
Switch (k$)	2080	2080	416	832	0
Wireless (k$)	0	192	2400	1920	0
Cable (k$)	80	80	0	32	0
PLC Modem (k$)	0	0	0	0	787
Gateway (k$)	0	0	0	0	351
Total (k$)	2240	2432	2896	3024	1218

Table 2: Comparison of Equipment Costs

use them as proxy for comparing different technology that can be used to construct out-of-band MNs: 60GHz WiFi, FSO, and electrical packet switching.

Construction cost: Wireless designs (3DBF, Firefly, & Diamond; as well as other WiFi and FSO designs [41, 47]) have various requirements on the datacenter interior designs. For example, reflective surfaces (static [36, 48, 84] or mechanically controlled [41]) must be installed for connectivity. In addition,3DBF has ceiling height requirements [84], which may incur room modifications in deployment. Furthermore, Diamond also requires the spacing between racks. This limits the number of racks per room, and Diamond deployments may need more rooms to hold the same number of servers.

In contrast, PowerMan leverages the wiring in existing DCPS to achieve scalable connectivity for MN, and only replaces the PDUs and PSUs in the DCPS. Thus, constructing PowerMan should incur no cost in modifications of room design or floor plan, which greatly reduces the cost of deployment compared to the other proposals.

Equipment cost: Next, we compare the equipment costs in Table 2, and we explain the assumptions as follows. For PowerMan, we assume PSU uses Design 1, which incurs no cable cost. The tree topology of PowerMan is configured as h=3,k=27. Each PLC modem is $30[5]. Each Gateway is $500. For other designs, we consider the cost of NICs on the server, switches, wireless radios and cables. We adopt the conservative estimates in [36], and make the following cost assumptions: each wireless radio component costs $60 [85], each 40-port switch costs $1040, each NIC port costs $5 [46], each FSO device port costs $150 [48], and an average cost of $1 per meter for cabling [48] and $1 per square meter of absorbing paper. We assume the reflectors used [36, 48, 84] have negligible cost as equipments.

Overall, PowerMan can be constructed with $1/2 \sim 1/3$ of the cost of other proposals at the same scale, confirming PowerMan as a cost-effective option for MN.

Power consumption: Power consumption is an important component of operational cost. In Figure 18, we compare the operational power consumption of different designs. We assume each NIC consumes 5 Watts (W) [36], each PLC modem 3.73W (§3.1), each switch 170W [36], and each gateway 300W[6]. For wireless designs, each FSO component in Firefly consumes 3W [48] and the WiFi module in Diamond 60W [36].

[5] We use retail price here. The per-unit price are dependent on many factors: quantity, availability, distance, etc. With large quantity, the price tend to decrease.

[6] We use a rack-mounted server as the packet-forwarding hardware in the prototype, thus the high power usage. This can be reduced with a typical packet-forwarding device.

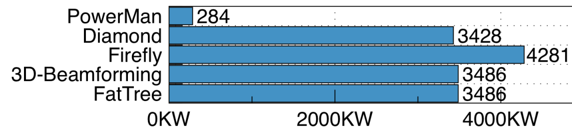


Figure 18: Comparison of Power Consumption

In general, PowerMan consumes much lower power at the same scale, using 6%$\sim$9% power of other designs. This is because PowerMan is mostly composed of PLC modems with lower power usage.

7 Discussion

In this section, we discuss the limitations of PowerMan and experiences from constructing and operating the PLC testbed and PowerMan prototype.

Interference in DCPS: The major contributor to the loss of performance (§3.2) is high-frequency noise from sources of interference in DCPS, including lighting, cooling, mechanical system, etc. As pointed out in [60], PLC is an extremely harsh environment for the high-bandwidth, high-frequency communication signals, as critical channel parameters (e.g. noise, impedance, and attenuation) are highly unpredictable [69]. Given the severity of the interference, in the design of PowerMan, we aim to limit the PLC signal within each PDU (i.e. within a PLC LAN (PLAN)) which reduces the signals exposure to interference, as the signal only travels short distances within the PDU. The LPF in each layer also removes high frequency noises from non-PLC sources.

Low Throughput Alternatives to PLC: Our experiments and simulations (6) exhibit low throughput for various traffic patterns, and as we have discussed, the reasons include noise, signal attenuation, MAC layer overhead, etc. Due to the low throughput of PowerMan, it is natural to consider using low cost, low bandwidth WiFi or Ethernet devices as alternatives. Compared to low cost alternatives, we believe PLC is advantageous in the three goals we outlined for an out-of-band MN (§1): survivability, deployability, and scalability.

- *Survivability:* PLC can leverage the robustness in existing power systems. Power system is the last-to-fail system in datacenters, and is independent of the data plane network. Embedded in DCPS, PowerMan can survive data plane failures, or even power system failures in Tier-II to IV datacenters, and is ready for immediate diagnosis and recovery. Other low cost alternatives do not share this quality. For example, a separate WiFi network requires additional monitoring and management systems to achieve the same level of robustness of power system.

- *Deployability:* PowerMan reuses wiring in existing DCPS, thus there is no need to change ceiling height, and rack dimensions. This compatibility with existing datacenter designs greatly reduces the deployment cost. In contrast, WiFi-based solutions require changes in the rack dimensions to accommodate antennae of servers and access points. Ethernet-based solutions require additional rack space and cabling.
- *Scalability:* PowerMan reaches every device in the datacenter, as it reuses wiring in DCPS. Ethernet-based solutions with the same topology as PowerMan can reach the same port count, but at the cost of much more cabling. Like PLC, WiFi also suffers from the interference, which is more difficult to manage than that in a wired network. PowerMan is able to use signal filters on the border of two PLANs to eliminate interference between them. Such is not so easy in a wireless network, as there is no clear border between two broadcast domains. For WiFi-based solutions, handling interference requires careful planning of antennae direction, AP radio power, location, and channel selection. At the scale of a modern datacenter, the management of WiFi-based solution is challenging.

DC datacenters: Many modern datacenters are using DC power [31, 55, 68]. Our design can also work on such DCPS, because PowerMan is a design that utilizes power lines, which is the same in both DC and AC power systems. The carrier frequencies in PLC devices (assuming compliance with HomePlug standards) come from OFDM circuitries, and are not the 50-60Hz AC power.

Security Concerns: Datacenter MN is a high value target, and a MN using PLC may be vulnerable to on-premise attacks. PowerMan can adopt security mechanisms on MAC, network, transport, application layers. For example, in the PLC MAC layer, HomePlug 1.0 [56] supports 56-bit DES encryption, and later versions (HomePlug AV/AV2 [24, 81]) support 128-bit AES.

Cooling: Even with intensive experiments on PowerMan prototype, we have not yet witnessed any overheating issues for PLC modems. This is because: 1) the PLC modems have low power profile, and 2) the PLC modems are placed outside of the servers. Bump-in-the-Wire PSU design may benefit from the same reasons; but it is still important to investigate the heat dissipation of the Full-Integration design inside a rack-mounted server or switch as future work.

8 Related works

We summarize the related work in three broad categories: datacenter management, alternative datacenter networking architectures, and PLC networking.

Datacenter Management: There is vast literature on the management and control planes of datacenter networks [20, 42, 49, 52, 70, 85]. They often assume that the management traffic can be delivered, and PowerMan complements these works with an out-of-band MN that offers necessary latency and bandwidth, while being survivable, scalable, and deployable.

Datacenter Networking Architectures: Datacenter networks in production usually use the Clos network [19, 43, 54, 65, 67] to achieve high bisection bandwidth. Using flexible networking technology such as optical switching [32, 34, 38, 57, 59, 64, 80], FSO [41, 48], and 60GHz wireless radios [36, 47], dynamic network topologies are proposed to mitigate traffic hotspots and changing demands. We differ from them in our technology choice. In terms of datacenter MN, Angora [85] proposed using 60GHz wireless radio to construct a datacenter ”facility network”, which is a MN but with much stricter latency requirements. In contrast, PowerMan is the first attempt to employ PLC in the datacenter MN setting, and as our cost comparisons (§6.3) suggest, PowerMan has lower initial cost and operating power consumption than the other technologies at the same scale.

PLC Networking: In PLC PHY [60, 69] and MAC [51, 74, 75, 76, 77, 78] layer, many efforts have been made to improve the bandwidth, reliability, and latency [83]. In comparison, PowerMan focuses on the application of PLC in MN, exploring networking (§3.1) and scalability (§4.3) for datacenter management. PowerMan can benefit from all PHY and MAC layer optimizations (e.g. parameter setting, dynamic bandwidth allocation scheme), as they improve the PLANs in PowerMan.

9 Conclusion

This paper has introduced PLC as an out-of-band management channel for datacenters. We build a small-scale PLC testbed, and demonstrate the potential of PLC with deployment of actual management applications. In the process, we identified the wiring and scalability issues which prevent deployment of PLC in datacenters. To tackle these problems, we design PowerMan, a datacenter MN using PLC that can be implemented using commercially available PLC devices. We build a PowerMan prototype on a small testbed of 12 servers. Using experiments and large-scale simulations, we evaluate its performance, reliability, and cost-effectiveness.

For future work, we plan to 1) investigate custom PLC devices with optimized PHY/MAC layers to improve latency, throughput, scalability, and reliability; 2) integrate PSU with single-board computer, so as to provide isolation from local OS-related failures.

Acknowledgements: This work is supported in part by Hong Kong RGC ECS-26200014, GRF-16203715, GRF-613113, CRF-C703615G, & China 973 Program No.2014CB340303. We thank the anonymous NSDI reviewers and our shepherd Shyam Gollakota for their constructive feedback and suggestions.

References

[1] Ansible. https://www.ansible.com/. (Accessed on 01/08/2017).

[2] ansible/ansible-examples. https://github.com/ansible/ansible-examples/. (Accessed on 01/19/2017).

[3] Chef. https://www.chef.io/chef/. (Accessed on 01/08/2017).

[4] chef-web-docs/resource_examples.rst at master chef/chef-web-docs. https://github.com/chef/chef-web-docs/blob/master/chef_master/source/resource_examples.rst. (Accessed on 01/19/2017).

[5] Configuration openstack havana with ryu. https://github.com/osrg/ryu/wiki/configuration_openstack_havana_with_ryu. (Accessed on 01/08/2017).

[6] datacenter/empirical-traffic-gen: Simple client-server application for generating user-defined traffic patterns. https://github.com/datacenter/empirical-traffic-gen. (Accessed on 01/08/2017).

[7] Introduction to ad-hoc commands ansible documentation. http://docs.ansible.com/ansible/intro_adhoc.html. (Accessed on 01/20/2017).

[8] Netgear pl1200. https://www.netgear.com/home/products/networking/powerline/PL1200.aspx?cid=wmt_netgear_organic. (Accessed on 01/24/2017).

[9] Nginx. https://www.nginx.com/. (Accessed on 01/19/2017).

[10] nginx cookbook - chef supermarket. https://supermarket.chef.io/cookbooks/nginx. (Accessed on 01/19/2017).

[11] Openstack docs: Network design. http://docs.openstack.org/ops-guide/arch-network-design.html. (Accessed on 01/07/2017).

[12] osrg/ryu: Ryu component-based software defined networking framework. https://github.com/osrg/ryu. (Accessed on 01/08/2017).

[13] Product faq powerline adapters. http://kb.netgear.com/20233/Product-FAQ-Powerline-Adapters?cid=wmt_netgear_organic. (Accessed on 01/15/2017).

[14] Ryubook.pdf. https://osrg.github.io/ryu-book/en/Ryubook.pdf. (Accessed on 01/19/2017).

[15] Tia-942 telecommunications infrastructure set. https://global.ihs.com/tia_telecom_infrastructure.cfm?RID=Z56&MID=5280. (Accessed on 09/25/2017).

[16] Tp-link av1200 gigabit passthrough powerline starter kit. http://www.tp-link.com/ph/products/details/cat-18_TL-PA8010P-KIT.html. (Accessed on 01/24/2017).

[17] Trendnet powerline 1200 av2 adapter kit. https://www.trendnet.com/products/powerline-1200/TPL-420E2K. (Accessed on 01/24/2017).

[18] Data center site infrastructure tier standard: Topology. *Uptime Institute* (2012).

[19] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *ACM SIGCOMM* (2008).

[20] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI* (2010).

[21] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center tcp (dctcp). *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 63–74.

[22] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM* (2013).

[23] Alliance, H. Homeplug_av2_whitepaper_20130909.pdf. https://www.codico.com/fxdata/codico/prod/media/Datenblaetter/AKT/HomePlug_AV2_whitepaper_20130909.pdf. (Accessed on 02/10/2018).

[24] Alliance, H. Homeplug av specification. *Version 1*, 2006.12 (2007), 16.

[25] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND SUN, W. Information-agnostic flow scheduling for commodity data centers. In *USENIX NSDI* (2015).

[26] BANERJEE, S., BHATTACHARJEE, B., AND KOMMAREDDY, C. *Scalable application layer multicast*, vol. 32. ACM, 2002.

[27] BARLOW, R. E., AND PROSCHAN, F. *Mathematical theory of reliability*. SIAM, 1996.

[28] BARROSO, L., DEAN, J., AND HOEZLE, U. Web search for a planet: the architecture of the google cluster. *IEEE Micro 23*, 2 (2003), 22–28.

[29] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture 8*, 3 (2013), 1–154.

[30] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: Fine grained traffic engineering for data centers. In *CoNEXT* (2010).

[31] BORS, D. Data center power system design debate: Ac or dc? http://www.ecmweb.com/power-quality-archive/data-center-power-system-design-debate-ac-or-dc. (Accessed on 02/10/2018).

[32] CHEN, K., SINGLA, A., SINGH, A., RAMACHANDRAN, K., XU, L., ZHANG, Y., WEN, X., AND CHEN, Y. Osa: An optical switching architecture for data center networks with unprecedented flexibility. In *USENIX NSDI* (2012).

[33] CHEN, K.-C. Medium access control of wireless lans for mobile computing. *IEEE Network 8*, 5 (1994), 50–63.

[34] CHEN, L., CHEN, K., ZHU, Z., YU, M., PORTER, G., QIAO, C., AND ZHONG, S. Enabling wide-spread communications on optical fabric with megaswitch. In *NSDI* (2017).

[35] CISCO. Data center power and cooling. http://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/unified-computing/white_paper_c11-680202.pdf. (Accessed on 01/24/2017).

[36] CUI, Y., XIAO, S., WANG, X., YANG, Z., ZHU, C., LI, X., YANG, L., AND GE, N. Diamond: Nesting the data center network with wireless rings in 3d space. In *USENIX NSDI* (2016).

[37] DEAN, J. Software engineering advice from building large-scale distributed systems.

[38] FARRINGTON, N., PORTER, G., RADHAKRISHNAN, S., BAZZAZ, H. H., SUBRAMANYA, V., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *ACM SIGCOMM* (2010).

[39] FERREIRA, H. C., GROVE, H., HOOIJEN, O., AND VINCK, A. H. Power line communications: an overview. In *AFRICON, 1996., IEEE AFRICON 4th* (1996), vol. 2, IEEE, pp. 558–563.

[40] GALLI, S., SCAGLIONE, A., AND WANG, Z. For the grid and through the grid: The role of power line communications in the smart grid. *Proceedings of the IEEE 99*, 6 (2011), 998–1027.

[41] GHOBADI, M., MAHAJAN, R., PHANISHAYEE, A., DEVANUR, N., KULKARNI, J., RANADE, G., BLANCHE, P.-A., RASTEGARFAR, H., GLICK, M., AND KILPER, D. Projector: Agile reconfigurable data center interconnect. In *ACM SIGCOMM* (2016).

[42] GREENBERG, A., HAMILTON, J., MALTZ, D. A., AND PATEL, P. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review 39*, 1 (2008), 68–73.

[43] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A scalable and flexible data center network. In *ACM SIGCOMM* (2009).

[44] GRID, T. G. Power equipment and data center design. https://www.thegreengrid.org/en/resources/library-and-tools/382-Power-Equipment-and-Data-Center-Design. (Accessed on 01/24/2017).

[45] GUNGOR, V. C., AND LAMBERT, F. C. A survey on communication networks for electric system automation. *Computer Networks 50*, 7 (2006), 877–897.

[46] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., AND LU, S. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM* (2009).

[47] HALPERIN, D., KANDULA, S., PADHYE, J., BAHL, P., AND WETHERALL, D. Augmenting data center networks with multi-gigabit wireless links. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 38–49.

[48] HAMEDAZIMI, N., QAZI, Z., GUPTA, H., SEKAR, V., DAS, S. R., LONGTIN, J. P., SHAH, H., AND TANWERY, A. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *ACM SIGCOMM* (2014).

[49] HASSAS YEGANEH, S., AND GANJALI, Y. Kandoo: a framework for efficient and scalable offloading of control applications. In *ACM HotNets* (2012).

[50] HENDERSON, T. R., LACAGE, M., RILEY, G. F., DOWELL, C., AND KOPENA, J. Network simulations with the ns-3 simulator. *SIGCOMM demonstration 15* (2008), 17.

[51] HENRI, S., VLACHOU, C., HERZEN, J., AND THIRAN, P. Empower hybrid networks: Exploiting multiple paths over wireless and electrical mediums. In *ACM CoNEXT* (2016).

[52] ISARD, M. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review 41*, 2 (2007), 60–67.

[53] KALBFLEISCH, J. D., AND PRENTICE, R. L. *The statistical analysis of failure time data*, vol. 360. John Wiley & Sons, 2011.

[54] KASSING, S., VALADARSKY, A., SHAHAF, G., SCHAPIRA, M., AND SINGLA, A. Beyond fat-trees without antennae, mirrors, and disco-balls. In *ACM SIGCOMM* (2017).

[55] KASSNER, M. P. Dc distribution is not just for the giants. http://www.datacenterdynamics.com/content-tracks/design-build/dc-distribution-is-not-just-for-the-giants/95037.fullarticle. (Accessed on 02/10/2018).

[56] LEE, M., NEWMAN, R. E., LATCHMAN, H. A., KATAR, S., AND YONGE, L. Homeplug 1.0 powerline communication lans: protocol description and performance results. *International Journal of Communication Systems 16*, 5 (2003), 447–473.

[57] LIU, Y. J., GAO, P. X., WONG, B., AND KESHAV, S. Quartz: a new design element for low-latency dcns. In *ACM SIGCOMM* (2014).

[58] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: Enabling innovation in campus networks. *ACM Computer Communication Review* (2008).

[59] MELLETTE, W. M., MCGUINNESS, R., ROY, A., FORENCICH, A., PAPEN, G., SNOEREN, A. C., AND PORTER, G. Rotornet: A scalable, low-complexity, optical datacenter network. In *ACM SIGCOMM* (2017).

[60] MENG, H., CHEN, S., GUAN, Y., LAW, C., SO, P., GUNAWAN, E., AND LIE, T. Modeling of transfer characteristics for the broadband power line communication channel. *IEEE Transactions on Power delivery 19*, 3 (2004), 1057–1064.

[61] NEE, R. V., AND PRASAD, R. *OFDM for wireless multimedia communications*. Artech House, Inc., 2000.

[62] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM* (2014).

[63] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E. J., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., ET AL. The design and implementation of open vswitch. In *USENIX NSDI* (2015).

[64] PORTER, G., STRONG, R., FARRINGTON, N., FORENCICH, A., SUN, P.-C., ROSING, T., FAINMAN, Y., PAPEN, G., AND VAHDAT, A. Integrating microsecond circuit switching into the data center. In *ACM SIGCOMM* (2013).

[65] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *ACM SIGCOMM* (2015).

[66] SCHWARTZ, M. Carrier-wave telephony over power lines: Early history [history of communications]. *IEEE Communications Magazine 47*, 1 (2009), 14–18.

[67] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., ET AL. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *ACM SIGCOMM* (2015).

[68] STARK, J. 380v dc power: Shaping the future of data center energy efficiency. http://www.datacenterknowledge.com/archives/2015/06/25/380v-dc-power-shaping-future-data-center-energy-efficiency. (Accessed on 02/10/2018).

[69] TANG, L., SO, P., GUNAWAN, E., CHEN, S., LIE, T., AND GUAN, Y. Characterization of in-house power distribution lines for high-speed data transmission. In *Proc. 5th Int. Power Engineering Conf.(IPEC 2001)* (2001), pp. 7–12.

[70] TOOTOONCHIAN, A., AND GANJALI, Y. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking* (2010), pp. 3–3.

[71] TURNER IV, W. P., PE, J., SEADER, P., AND BRILL, K. Tier classifications define site infrastructure performance. *Uptime Institute* (2006).

[72] VAMANAN, B., HASAN, J., AND VIJAYKUMAR, T. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review* (2012).

[73] VLACHOU, C. plc-click-elements/plcstats.h. https://github.com/christinavl/plc-click-elements/blob/master/PLCStats.h. (Accessed on 02/10/2018).

[74] VLACHOU, C., BANCHS, A., HERZEN, J., AND THIRAN, P. Analyzing and boosting the performance of power-line communication networks. In *ACM CoNEXT* (2014).

[75] VLACHOU, C., BANCHS, A., HERZEN, J., AND THIRAN, P. On the mac for power-line communications: Modeling assumptions and performance tradeoffs. In *IEEE ICNP* (2014).

[76] VLACHOU, C., BANCHS, A., HERZEN, J., AND THIRAN, P. Performance analysis of mac for power-line communications. *ACM SIGMETRICS Performance Evaluation Review 42*, 1 (2014), 585–586.

[77] VLACHOU, C., BANCHS, A., SALVADOR, P., HERZEN, J., AND THIRAN, P. Analysis and enhancement of csma/ca with deferral in power-line communications. *IEEE Journal on Selected Areas in Communications* (2016).

[78] VLACHOU, C., HERZEN, J., AND THIRAN, P. Fairness of mac protocols: Ieee 1901 vs. 802.11. In *Power Line Communications and Its Applications (ISPLC), 2013 17th IEEE International Symposium on* (2013), IEEE, pp. 58–63.

[79] VLACHOU, C., HERZEN, J., AND THIRAN, P. Simulator and experimental framework for the mac of power-line communications. EPFL-REPORT-205770.

[80] WANG, G., ANDERSEN, D., KAMINSKY, M., PAPAGIANNAKI, K., NG, T., KOZUCH, M., AND RYAN, M. c-Through: Part-time optics in data centers. In *ACM SIGCOMM* (2010).

[81] YONGE, L., ABAD, J., AFKHAMIE, K., GUERRIERI, L., KATAR, S., LIOE, H., PAGANI, P., RIVA, R., SCHNEIDER, D. M., AND SCHWAGER, A. An overview of the homeplug av2 technology. *Journal of Electrical and Computer Engineering 2013* (2013).

[82] ZATS, D., DAS, T., MOHAN, P., BORTHAKUR, D., AND KATZ, R. Detail: reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 139–150.

[83] ZHAO, Z., CHEN, I., ET AL. Moving homeplug to industrial applications with power-line communication network.

[84] ZHOU, X., ZHANG, Z., ZHU, Y., LI, Y., KUMAR, S., VAHDAT, A., ZHAO, B. Y., AND ZHENG, H. Mirror mirror on the ceiling: Flexible wireless links for data centers. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 443–454.

[85] ZHU, Y., ZHOU, X., ZHANG, Z., ZHOU, L., VAHDAT, A., ZHAO, B. Y., AND ZHENG, H. Cutting the cord: a robust wireless facilities network for data centers. In *ACM MobiCom* (2014).

Appendix

Management Traffic Optimizations

As discussed in §3.2.2, two common traffic patterns of management application is 1-to-N (e.g. configuration tasks) and N-to-1 (e.g. monitoring tasks). The experiments in §3.2&6.1 shows that such patterns perform poorly on PowerMan. In the following, we propose application-layer traffic optimizations to reduce the completion times of these two patterns on PowerMan. We assume the controller is located at the root of the tree.

Accelerating 1-To-N Pattern

As shown in Figure 16, PowerMan has low per-server bandwidth at large scale. This is because the controller node needs to maintain connection to all servers, so the per-server bandwidth is constrained by the interface capacity of the controller. Low per-server bandwidth can prolong completion times of configuration tasks.

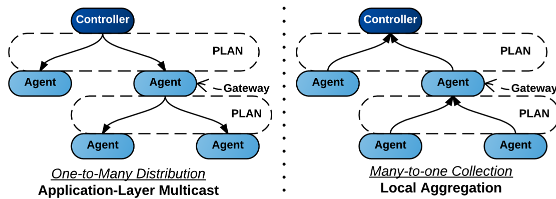


Figure 19: Accelerating Management Application Traffic Patterns

We propose to construct an application-layer multicast (ALM) overlay network [26] in PowerMan for management tasks with 1-to-N distribution pattern, where the gateway in each PDU act as a distribution agent in corresponding PLAN. As shown in Figure 19, the distribution from controller to all server is divided into multiple distributions within different PLANs.

We evaluate the performance of ALM. We use the production traces in §3.2.2. For baseline performance, we create 1-to-N traffic patterns using the traffic generator with different numbers of receivers. For ALM, we modify the traffic generator to include an implementation of ALM agent, and enable the agents in the gateways. We collect the FCTs and the results are plotted in Figure 20. We observe that ALM reduces the FCT for 1-to-N pattern, and the performance gap increases with the number of total receivers. For 10 receivers, the FCT is reduced by 15.38% on average. The main reason is that the total traffic volume is reduced with ALM, as copies of the flow are created by the agent in each gateway, which reduces the traffic volume at higher layers.

Accelerating N-to-1 Pattern

Information collection tasks include monitoring, diagnostics, and measurement, which exhibit N-to-1 pattern

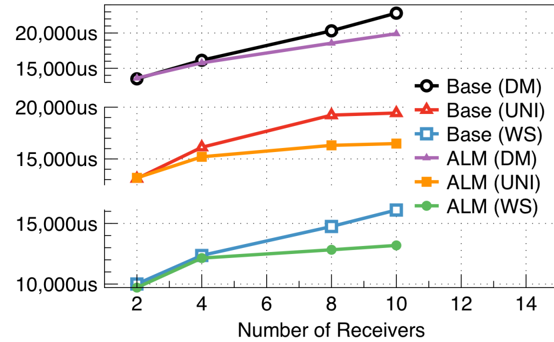


Figure 20: Accelerate 1-to-N Pattern with Application-Layer Multicast: Average FCT

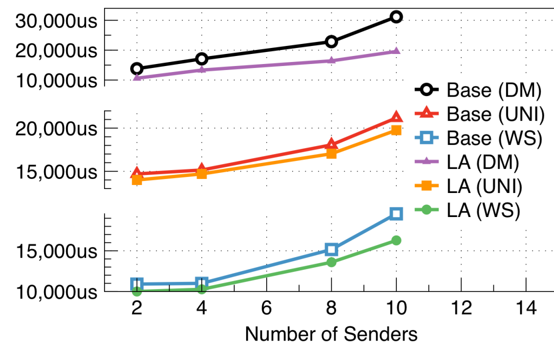


Figure 21: Accelerate N-to-1 Pattern with Local Aggregation: Average FCT

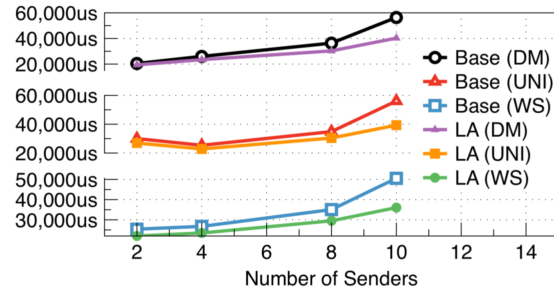


Figure 22: Accelerate N-to-1 Pattern with Local Aggregation: 99 percentile FCT

from the servers to the controller. As the dual of ALM, we propose local aggregation (LA) at each layer (PLAN) of the tree. An agent at each gateway collects the information from all servers/agents in its PLAN, and then sends the aggregated information to the agent in upper-layer gateway.

We then evaluate the performance of LA. For baseline, we create N-to-1 patterns using the same flow size distributions as above. For LA, we implemented a LA agent for the traffic generator, and enable them on all gateways. We collect the FCTs and plot the average in Figure 21 with respect to the number of senders. We can see that, although LA in general outperforms baseline, the performance gap is smaller than that of Figure 20. This is because the total traffic volume is not reduced with LA. However, LA on PowerMan effectively reduces the num-

ber of contending flows at the controller from N (total number of servers) to $k-1$ (number of nodes in a PLAN). This can be observed in Figure 22, which summarizes the tail latencies (99th percentile FCTs). Tail latencies capture the worst performing flows whose completion times are prolonged by events such as packet loss, reordering, frame collision. Using LA to reduce the number of contending flows at the receiver decreases the occurrences of tail latency events, which improves the completion times. Finally, a further optimization is to compress local information before sending. Compression of locally collected information can reduce total traffic volume, and it would be beneficial if the computation overhead on the gateway is acceptable.

NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion

Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, Martin Vechev
ETH Zürich

netcomplete.ethz.ch

Abstract

Network operators often need to adapt the configuration of a network in order to comply with changing routing policies. Evolving existing configurations, however, is a complex task as local changes can have unforeseen global effects. Not surprisingly, this often leads to mistakes that result in network downtimes.

We present NetComplete, a system that assists operators in modifying existing network-wide configurations to comply with new routing policies. NetComplete takes as input configurations with "holes" that identify the parameters to be completed and "autocompletes" these with concrete values. The use of a partial configuration addresses two important challenges inherent to existing synthesis solutions: *(i)* it allows the operators to precisely control how configurations should be changed; and *(ii)* it allows the synthesizer to leverage the existing configurations to gain performance. To scale, NetComplete relies on powerful techniques such as counter-example guided inductive synthesis (for link-state protocols) and partial evaluation (for path-vector protocols).

We implemented NetComplete and showed that it can autocomplete configurations using static routes, OSPF, and BGP. Our implementation also scales to realistic networks and complex routing policies. Among others, it is able to synthesize configurations for networks with up to 200 routers within few minutes.

1 Introduction

In a world where more and more critical services converge on IP, even slight network downtimes can cause large financial or reputational losses. This strategic importance contrasts with the fact that managing a network is surprisingly hard and brittle. Out of high-level requirements, network operators have to come up (often manually) with low-level configurations specifying the behavior of hundreds of devices running complex distributed protocols. A single misconfiguration can bring down the network infrastructure, or worse, a piece of the Internet in case of BGP-related misconfigurations. Every few months downtimes involving major players such as NYSE [1], Google [2], Facebook [3], or United Airlines [4] make the news. Actually, studies show that human-induced misconfigurations, *not* physical failures, explain the majority of downtimes [5].

To address these challenges, recently there has been an increased interest in configuration verification [6, 7, 8, 9, 10, 11, 12, 13] and synthesis [14, 15, 16, 17, 18, 19, 20]. Configuration synthesis in particular promises to alleviate most of the operator's burdens by deriving *correct* configurations out of high-level objectives.

Challenges in network synthesis While promising, network operators can still be reluctant to use existing synthesis systems for at least three reasons: *(i) interpretability*: the synthesizer can produce configurations that differ wildly from manually provided ones, making it hard to understand what the resulting configuration does. Moreover, small policy changes can cause the synthesized configuration (or configuration templates in the case of PropaneAT [16]) to change radically; *(ii) protocol coverage*: existing systems [15, 16] are restricted to producing BGP-only configurations, while most networks rely on multiple routing protocols (e.g., to leverage OSPF's fast-convergence capabilities); and *(iii) scalability*: recent synthesizers such as SyNET [20] handle multiple protocols but do not scale to realistic networks.

NetComplete We present a system, NetComplete, which addresses the above challenges with partial synthesis. Rather than synthesizing a new configuration from scratch, NetComplete allows network operators to express their intent by sketching parts of the existing configuration that should remain intact (capturing a high-level insight) and "holes" represented with symbolic values which the synthesizer should instantiate (e.g., OSPF weights, BGP import/export policies).

NetComplete then autocompletes these "holes" such that the resulting configuration leads to a network that exhibits the required behavior. Our approach supports a practically relevant scenario as few operators ever start from scratch but rather modify their existing configurations (e.g., OSPF weights) to handle new routing requirements. This evolving approach also has the benefit of better explainability as large parts of the existing configuration are preserved in the newly synthesized configuration. Further, because we focus on synthesizing parts of the configuration, there is an opportunity to scale the synthesizer to realistic networks. This opportunity arises even though NetComplete is quite expressive: it handles static routes, OSPF, and BGP[1] as well as a variety of essential routing requirements such as waypointing, failure-resilience, load-balancing, and traffic isolation.

NetComplete reduces the autocompletion problem to a constraint satisfaction problem that it solves with SMT solvers (e.g., Z3 [21]). The main challenge is that a naive encoding of the problem leads to complex constraints that cannot be solved in reasonable time (e.g., within a day). To scale, NetComplete relies on two key insights: *(i)* partial evaluation along with *(ii)* network-specific heuristics to efficiently navigate the search space. Specifically, it speeds up BGP synthesis by propagating symbolic announcements through partial BGP policies allowing it to eliminate many variables. For OSPF, NetComplete is 100x faster than a naive encoding via a new counter-example guided inductive synthesis algorithm. Overall, NetComplete autocompletes configurations for networks with up to 200 routers in few minutes.

Contributions Our main contributions are:

- A new approach to network-wide configuration synthesis based on autocompletion of partial configurations. It enables operators to evolve existing configurations so they match new requirements.
- A scalable synthesis procedure based on SMT constraints which relies on partial evaluation techniques along with domain-specific heuristics and counter-example guided inductive synthesis.
- An end-to-end implementation of our approach in a system called NetComplete which outputs actual Cisco configurations.
- A comprehensive evaluation of NetComplete using a variety of real-world topologies and complex requirements. Our results demonstrate that NetComplete can effectively autocomplete partial configurations for large networks with up to 200 routers within few minutes.

[1] We plan to add support for more protocols and mechanisms in future work, including MPLS and route redistribution.

2 Motivating Scenarios

In this section, we motivate the need for NetComplete through *three practical use cases* rooted within existing network management practices. These use cases are difficult or practically impossible to solve today.

Scenario 1: *Evolving configurations preserving existing semantics*. Existing configurations typically embed deep knowledge of semantics and design guidelines. For instance, operators often use specific OSPF weights to identify primary/backup links, and specific BGP local-preferences or communities to identify their peers. This (often unwritten) semantic helps them reason about the network-wide configuration. At the same time, these rules also reduce the operators flexibility as it can complexify the implementation of new routing requirements, e.g., by requiring the modification of multiple weights instead of one.

NetComplete allows operators to communicate such semantics as constraints on the configuration sketch and let the synthesizer find a valid network-wide configuration that adheres to the operators style.

Scenario 2: *Simplifying federated or constrained management*. Network configurations are often maintained by multiple teams of operators [22, 23], each responsible for some parts (e.g., edge vs core) or functionalities. Coordinating changes in these federated configurations tends to be challenging as multiple teams need to come together. With NetComplete, the operators can easily explore whether there is a way to implement the policy locally, for instance, without adapting the BGP configuration (i.e., by restricting changes to the OSPF configuration). Similar requirements appear in heterogeneous networks where not all routers support all protocols (e.g., due to licensing issues or device capabilities).

NetComplete allows operators to simply communicate such constraints as part of the sketch and let the synthesizer find a multi-protocol configuration.

Scenario 3: *Configuration Refactoring and Network Merging*. Configurations evolve over time and this increases their complexity. Design decisions that made sense in the past may no longer do, requiring refactoring. Other examples calling for large refactoring include merging and acquisitions; e.g., when a company buys another one and wishes to integrate their networks [24].

NetComplete helps operators to refactor configurations by enabling them to morph entire pieces of their existing configurations, e.g., to adopt the configuration guidelines of one network and let the synthesizer compute and propagate the changes network-wide.

3 Overview

We now show how given a network topology, high-level routing requirements, and a partial configuration, NetComplete autocompletes the partial configuration to a correct network-wide configuration. First, we present a small running example and define NetComplete's inputs. We then present the key synthesis steps to produce the output configuration before explaining the more complex steps in detail in the following sections.

3.1 Running Example

In Fig. 2, we show how a network operator would use NetComplete to synthesize a network-wide configuration that enforces routing requirements. We consider that the Autonomous System (AS) of the operator's network is *AS*500 and consists of four routers: *A*, *B*, *C*, and *D*. This network is further connected to one customer peer *AS*100 and three external peers: *AS*200, *AS*300, and *AS*400.

High-level Routing Policy The policy for our example is given in Fig. 1. Rule (1) disallows transit traffic between external peers; e.g., *AS*200 cannot send traffic to *AS*300 through the network. Rule (2) defines how the customer peer accesses prefixes announced by external peers: *AS*300 is most preferred, followed by *AS*400, and then *AS*200. Traffic to *AS*200 may exit via *B* or *C*, where *B* is preferred. Rules (3) and (4) capture traffic engineering requirements. Note that this policy can be formalized in a high-level SDN-like language, such as Propane [15], Genesis [18], Frenetic [25], or SyNET [20].

Rule 1 No transit between *AS*200, *AS*300, and *AS*400;
Rule 2 Traffic from the customer peer *AS*100 to the external peers prefers exit routers in order: *AS*300, *AS*400, *AS*200 via *B*, *AS*200 via *C*;
Rule 3 Traffic from *AS*100 to *AS*300 is load-balanced along $A \to C$ and $A \to D \to C$; if both paths are unavailable, then the path $A \to B \to C$ is used;
Rule 4 Traffic from *AS*100 to *AS*400 must follow the path $A \to B \to C$.

Figure 1: High-level policy for our running example

3.2 NetComplete Inputs

NetComplete takes three inputs: (i) network topology, (ii) routing requirements, and (iii) a configuration sketch.

(1) Network Topology The network topology is given via a graph over the set of routers (*A*, *B*, *C*, and *D*) and external peers (*AS*100, *AS*200, *AS*300, and *AS*400). An edge represents a physical link that connects two nodes.

(2) Routing Requirements We now describe the type of requirements supported by NetComplete. We start with some basic notation. A *routing path* is of the form: $P ::= Src \to R_1 \to \cdots \to R_n \to Dst$, where *Src* and *Dst* are source and destination routers, respectively, and $R_1, \ldots, R_n$ are router identifiers. We use a wildcard notation to denote sets of simple paths, i.e., paths without repeated nodes. For example, $Src \to * \to Dst$ denotes all simple paths from *Src* to *Dst*.

NetComplete supports positive and negative requirements. Positive requirements have the form

$$Req ::= (P, \cdots, P) \mid (P = \cdots = P) \mid Req \gg Req$$

where *P* is a routing path. All routing paths that appear in a requirement must have identical source and destination. The semantics of requirements is as follows:

An *any-path* requirement $(P_1, \ldots, P_k)$ is satisfied if the traffic from the source to the destination is forwarded along *any* available path in $\{P_1, \ldots, P_k\}$. The requirement is not-applicable if all paths $P_1, \ldots, P_k$ are unavailable. We remark that any-path requirements are used to ensure failure-resilience. We will refer to any-path requirements (P) consisting of a single path *P* as *simple* requirements.

An *ECMP* requirement $(P_1 = \cdots = P_k)$ is satisfied if the traffic from *Src* to *Dst* is load-balanced among all available paths in the set $\{P_1, \ldots, P_k\}$. The requirement is not-applicable if all paths $P_1, \ldots, P_k$ are unavailable. We remark that ECMP requirements are useful to capture load-balancing.

An *ordered* requirement $Req_1 \gg Req_2$ defines a *preference* over requirements. This requirement is satisfied if the most preferred applicable requirement is satisfied, and it is not-applicable if both requirements are not-applicable. For example:

$$(AS100 \to A \to B \to C \to AS300) \\ \gg (AS100 \to A \to C \to AS300)$$

is satisfied if traffic from *AS*100 to *AS*300 is forwarded along this path if it is available:

$$AS100 \to A \to B \to C \to AS300$$

Otherwise traffic is forwarded along the path:

$$AS100 \to A \to C \to AS300$$

NetComplete also supports negative requirements of the form $!\{P_1, \ldots, P_k\}$, where $\{P_1, \ldots, P_k\}$ is a set of routing paths. This requirement is satisfied if traffic is not forwarded along any path in this set. Negative requirements are useful to express traffic isolation.

The requirements for our running example are given in Fig. 2b. We interpret sets of paths, such as $AS100 \to * \to AS300$, as any-path requirements. Policy rules 1, 2, 3, and 4, given Fig. 1, are specified as requirements 1, $2-7$, 8, and 9, respectively. We use a natural assignment

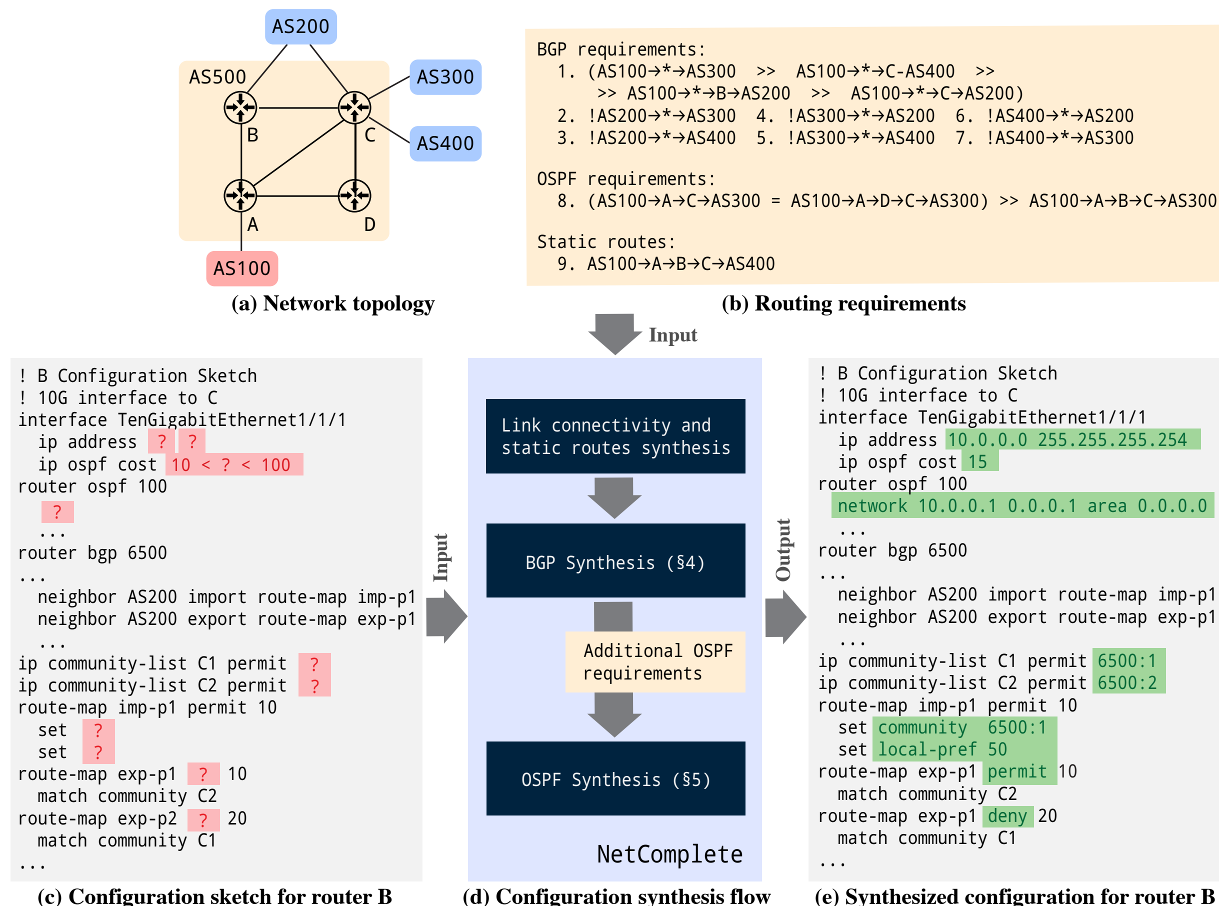


Figure 2: Overview of NetComplete. The inputs are: **(a)** network topology, **(b)** routing requirements, and **(c)** a configuration sketch. The output is a configuration for each router; for the configuration of router B see **(e)**.

of requirements to protocols. For example, requirements $1-7$ pertain to external peers and they are assigned to the Border Gateway Protocol (BGP). Requirement 8 pertains to traffic engineering within the network and is assigned to the Open-Shortest Path First (OSPF) protocol, which forwards traffic along the shortest path. Note that requirements 8 and 9 cannot be both enforced by OSPF. To enforce requirement 9, the cost of $A \to B \to C$ must be lower than that of $A \to C$ and $A \to D \to C$. However, this would also divert traffic from $AS100$ to $AS300$ to be forwarded along routers $A \to B \to C$, which would violate requirement 8. To this end, requirement 9 is enforced using a static route.

We remark that the requirements above can be specified manually by the operator, or using existing systems [15, 16, 18, 25] that compile high-level policies to forwarding paths.

(3) Configuration Sketch Configuration sketches are router configurations where some of the parameters are left symbolic. To specify symbolic values, the operator tags parts of the configurations with a question mark symbol ? (instead of writing concrete values). The symbol ? represents: *(i)* specific attributes (e.g., OSPF link cost, BGP local preferences[2]); or *(ii)* entire import / export policies, *e.g.*, `match ?, action ?`.

As an example, we depict the sketch of router B's configuration in Fig. 2c. We remark that operators can write additional constraints to restrict how NetComplete instantiates symbolic parameters. For example, the symbolic OSPF link cost in the sketch of router B is constrained to values between 10 and 100.

This sketching language enables NetComplete to be used in different scenarios. For example, changes can be restricted to certain parts of the network [Scenario 2]. By leaving most of the configurations symbolic, an operator can explore a large range of possible configurations that implement a given set of requirements [Scenarios 1 and 3]. Moreover, an operator can also provide a fully concrete configuration to verify its correctness.

[2]Except BGP AS numbers, which are assigned based on higher-level considerations that are not captured in the requirements.

3.3 Configuration Synthesis

NetComplete synthesizes a network-wide configuration that enforces the requirements in three steps.

First, it synthesizes the sessions between routers that have a physical link between them and may be necessary to enforce the routing requirements. Further, it configures any static routes defined in the requirements. For example, for requirement $AS100 \to A \to B \to C \to AS400$, NetComplete establishes a session between $A-B$ and $B-C$, and configures static routes at A and B.

Second, NetComplete synthesizes router-level BGP configurations based on the BGP routing requirements. To this end, NetComplete computes a propagation graph that captures which BGP announcements are exchanged between the routers and in what order they must be selected. NetComplete then synthesizes BGP configurations that enforce the constructed propagation graph. We explain this step in detail in §4. Note that BGP may select routes based on path costs (computed by OSPF). Therefore, whenever this is necessary to enforce the requirements, the BGP synthesizer outputs additional OSPF requirements to be enforced by the OSPF synthesizer.

Third, NetComplete synthesizes OSPF costs that enforce all OSPF requirements. This is a well-known hard problem that is difficult to scale to large networks. We solve the problem in §5 via a novel counter-example guided inductive synthesis algorithm.

If all synthesis steps succeed, NetComplete outputs a configuration that is guaranteed to enforce the requirements. Otherwise, a counter-example is returned to indicate that the requirements cannot be enforced for the given inputs. Based on this counter-example, the network operator can modify the partial configuration (by making more parameters symbolic) or adapt the requirements. We present a detailed evaluation of NetComplete with practical topologies and requirements in §6.

4 BGP Synthesis

We now present NetComplete's BGP synthesizer which takes as input BGP requirements and computes router-level BGP policies. It also outputs a set of OSPF requirements (to be fed to NetComplete's OSPF synthesizer) if the BGP requirements cannot be enforced by BGP policies alone. In the following, we first overview the BGP protocol (§4.1), then present the construction of a BGP propagation graph which defines correct propagation of BGP announcements (§4.2). We illustrate NetComplete's BGP sketches in §4.3 and propagation of (symbolic) announcements over them in §4.4. Finally, we describe our BGP synthesis procedure (§4.5).

Name	**Description**
`Prefix`	A value that represents a set of destination IPs that belong to the same traffic class
`LocalPref`	A positive integer that indicates the degree of preference for one route over the other routes
`Origin`	The origin of the announcement: `IGP`, `EGP`, or `Incomplete`
`MED`	(Multi-Exit Discriminator) A positive integer that indicates which of the multiple routes received from the same AS is selected
`ASPath`	The AS path to reach the destination
`ASPathLen`	The length of the AS path to the destination
`NextHop`	The router to which to forward packets
`Communities`	A list of tags carried with the announcement.

Figure 3: BGP attributes supported by NetComplete.

4.1 BGP Protocol

The BGP protocol is used to exchange information between ASes. An AS sends announcements to its neighboring ASes to inform them that it can carry traffic to prefixes (i.e., sets of IP addresses). Announcements are also exchanged within an AS to disseminate routing information among routers. Note that operators may partition their network into multiple ASes to use BGP to enforce routing requirements within the network. We refer to the ASes under the operator's control as private and to the remaining as public ASes.

Announcements have attributes, which are used to select a single *best* route out of (possibly) multiple routes to the same prefix; see Fig. 3. A router processes each received announcement using import filters, which may drop the announcement or modify its attributes. Then, the router selects the best route according to a local BGP policy, processes it using export filters, and forwards the result to its neighboring routers.

Each router uses the following preferences when selecting the best route:

1. Prefer higher `LocalPref`;
2. Prefer shorter `ASPathLen`;
3. Prefer lower origin type: `IGP` $<$ `EGP` $<$ `Incomplete`;
4. Prefer lower `MED`;
5. Prefer announcements from external routers;
6. Prefer lower `IGPCost`, calculated by the network's Internal Gateway Protocol (IGP), such as OSPF.

We assume prefixes in announcements do not overlap (as we can use known techniques [8] to ensure this).

4.2 BGP Propagation Graph

We present how NetComplete builds, for each prefix, a propagation graph that defines a correct enforcement of the BGP routing requirements for that prefix. In more

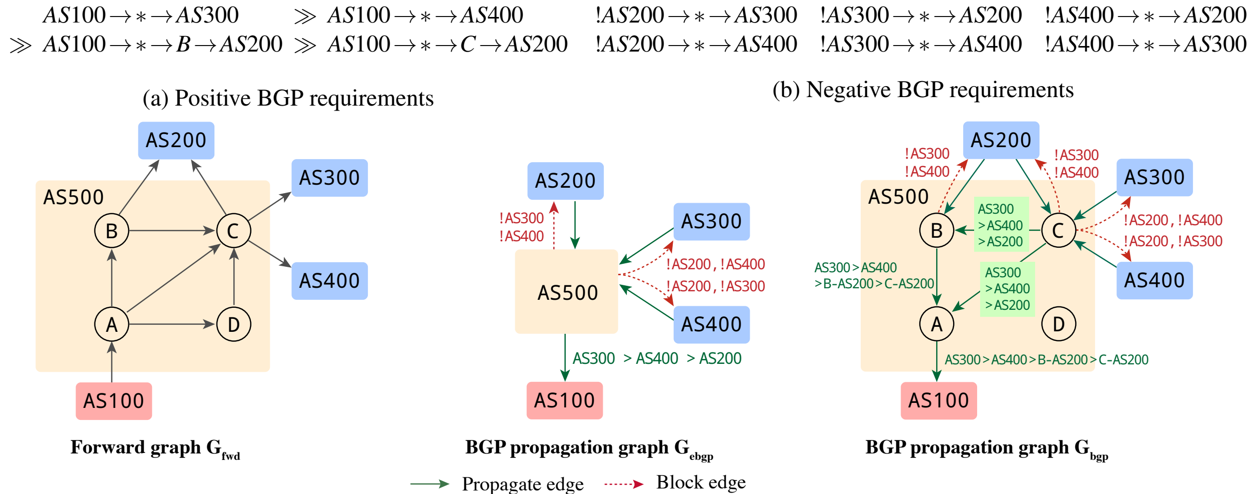


Figure 4: Deriving a BGP propagation graph from BGP requirements and a network topology.

details, NetComplete first constructs a graph G_{ebgp} that only considers announcements learned over eBGP. Then, it refines G_{ebgp} into G_{bgp}, which also defines how announcements are propagated internally (using iBGP). In Fig. 4, we illustrate the steps on our running example.

Construct eBGP propagation graph The graph G_{ebgp} contains one node for each private/public AS. For our example, G_{ebgp} has one private AS, $AS500$, and four public ones $AS100, \ldots, AS400$; see Fig. 4.

The graph G_{ebgp} has two kinds of labeled edges: *propagate* and *block* edges, labeled with the preference order over announcements and, respectively, announcements that must be dropped.

To add propagate edges, NetComplete traverses each positive BGP requirement backwards and appends edges along the traversed ASes. For example, for the requirement $AS100 \to * \to AS300$, NetComplete traverses three ASes and adds the propagate edges $AS300 \to AS500$ and $AS500 \to AS100$. While adding these edges, NetComplete tracks the set of announcements that must be propagated along them and labels the edges with the preference order based on the requirements.

To add block edges, NetComplete traverses each negative requirement and adds block edges to enforce it. For example, for the requirement $!AS200 \to * \to AS300$, it adds the block edge $AS500 \to AS200$, label with $!AS300$, to enforce the requirement.

Once G_{ebgp} is fully constructed, NetComplete checks if preferences over announcements are consistent. To illustrate, suppose $AS1$ must select announcements from $AS2$, and $AS2$ must select from $AS3$. Then, the preferences over announcements labeled along the edges $AS3 \to AS2$ and $AS2 \to AS1$ must match.

Construct iBGP propagation graph Next, NetComplete refines G_{ebgp} into a detailed propagation graph, G_{bgp}, that also accounts for iBGP.

First, for each private AS in G_{ebgp}, NetComplete adds to G_{bgp} all BGP-enabled routers within that AS. For our example, NetComplete adds the routers A, B, C, and D.

Second, NetComplete connects the neighbor routers between ASes that have an edge in G_{ebgp}. For example, for edge $AS200 \to AS500$ in G_{ebgp}, NetComplete adds the edges $AS200 \to B$ and $AS200 \to C$ to G_{bgp}.

Finally, NetComplete extends the paths learned via eBGP. Note that in iBGP routers will not export routes learned from another iBGP router.[3] Similar to G_{ebgp}, nodes in G_{bgp} are labeled with the preferences over announcements and NetComplete check if the preferences over announcements are consistent.

4.3 BGP Policies

We now present the semantics of BGP policies. A BGP policy applies on a set of announcements and has a match expression followed by zero or more actions. The match expression is a boolean formula over the announcement's attributes. If the match expression holds for the input announcement, then the actions are executed which modify the announcement's attributes or drop the announcement. For example, the following policy:

```
1 BGPPolicy
2   match next-hop AS200
3   set local-pref 10
```

matches an announcement whose `NextHop` attribute is set to $AS200$ and sets the value of attribute `LocalPref` to 10.

[3] While NetComplete does not support Route Reflectors, we plan to add support for them as their functionality is similar to eBGP.

```
1 AttributesSketch
2   match next-hop AS200
3   set local-pref ? < 50
```

(a) Attributes sketch S_{attr}

```
1 AbstractSketch
2   match ?
3   set ?
```

(b) Abstract sketch S_{abs}

Figure 5: Example of two BGP policy sketches.

Sketching BGP Policies NetComplete allows the network operator to define the policy sketch at three levels of details; *(i)* everything is concrete (no holes), *(ii)* define the types of matches and actions but leave the specific values empty (see Fig. 5a), *(iii)* or leave the matches and actions as holes (see Fig. 5b). We formalize the encoding of such sketches using SMT constraints in Appendix B. In §4.5, we show how NetComplete synthesizes BGP policy and instantiates the symbolic values in the given sketch to enforce the BGP propagation graph.

4.4 Processing Symbolic Announcements

We present how NetComplete processes symbolic BGP announcements passing through the various BGP policy sketches in the network. Given an announcement A, we write $\texttt{attr}_A$ to denote the attribute $\texttt{attr}$ of A. For instance, $\texttt{LocalPref}_A$ returns A's local preference. Each announcement attribute either has a concrete value, if its value is fixed by the partial configuration, or a symbolic value, if a correct concrete value is yet to be discovered by the BGP synthesizer.

We represent announcements symbolically as their attribute values are constrained by the BGP policies, which are yet to be synthesized by NetComplete. Each announcement A is represented with symbolic variables $\texttt{Prefix}_A, \ldots, \texttt{NextHop}_A$. The set of possible attribute values of A is captured by a conjunction of constraints over these variables. For example, the constraint

$$(\texttt{NextHop}_A = AS200) \land (0 < \texttt{LocalPref}_A < 50)$$

captures all announcements whose next hop is $AS200$ and local preference is a positive integer smaller than 50.

In addition to the attributes listed in Fig. 3 we introduce two boolean variables: $\texttt{Permitted}_A$, which indicates whether the announcement A is dropped, and $\texttt{eBGP}_A$, which indicates whether A is sent via eBGP or iBGP.

Processing Announcements with Policy Sketches A BGP policy sketch takes as input a symbolic announcement A_{in} (a set of constraints over A_{in}'s attributes) and outputs another symbolic announcement A_{out}. To compute the set of possible output announcements for a given input announcement, we take the conjunction of the BGP sketch constraints with the constraint that captures the set of possible concrete input announcements.

To illustrate this step, consider the input announcement $\texttt{NextHop}_{A_{in}} = AS200$ and the BGP sketch in Fig. 5a. Since the $\texttt{NextHop}$ attribute is concrete and equal to $AS200$, NetComplete knows that the input announcement would match this policy. Therefore, NetComplete captures the set of possible output announcements with the constraint:

$$(\texttt{LocalPref}_{A_{out}} = Var_1) \land (0 < Var_1 < 50) \\ \land (\texttt{NextHop}_{A_{out}} = \texttt{NextHop}_{A_{in}}) \land \cdots$$

Namely, the local preference of the output announcement is set to the value of Var_1, which is constrained to positive values below 50 (to be synthesized by NetComplete), and all remaining attributes are identical to those in the input announcement (captured with equality constraints, such as $\texttt{NextHop}_{A_{out}} = \texttt{NextHop}_{A_{in}}$).

As another example, consider the input announcement $\texttt{NextHop}_{A_{in}} = Var_1$ where the $\texttt{NextHop}$ attribute is symbolic. When evaluating this announcement with the BGP sketch in Fig. 5a, NetComplete captures the set of possible output announcements with the following constraint:

$$\begin{aligned} &\textbf{if } Var_1 = AS200 \\ &\quad \textbf{then } (\texttt{LocalPref}_{A_{out}} = Var_2) \land (0 < Var_2 < 50) \\ &\qquad\quad \land (\texttt{NextHop}_{A_{out}} = \texttt{NextHop}_{A_{in}}) \land \cdots \\ &\quad \textbf{else } (\texttt{NextHop}_{A_{out}} = \texttt{NextHop}_{A_{in}}) \land \cdots \end{aligned}$$

This constraint is more complex because the result of the match expression depends on the symbolic next hop (Var_1). If the next hop is $AS200$, then the local preference is set to Var_2 and all remaining attributes remain unchanged. Otherwise, all attributes in the output announcement A_{out} are identical to those in the input announcement A_{in}.

Encoding Selection of Announcements When a BGP router receives different announcements for the same prefix, it uses the preference ordering to select the best route; see §4.1. We encode the selection process into two SMT predicates: $PrefNoIGP(A_1, A_2)$ and $Pref(A_1, A_2)$. The predicate $PrefNoIGP(A_1, A_2)$ holds if and only if A_1 is preferred over A_2 without considering the IGP costs of A_1 and A_2. While the predicate $Pref(A_1, A_2)$ holds if and only if A_1 is preferred over A_2 with considering the IGP costs of A_1 and A_2. We show the encoding of these predicates in Appendix A.

4.5 BGP Policy Synthesis

We now describe how NetComplete synthesizes BGP policies from requirements and policy sketches.

Encoding Requirements Suppose that a router receives multiple announcements $A_1, \ldots, A_n$ to the same prefix. The BGP propagation graph identifies a preference with which the announcements must be selected by the router.

Suppose the router must select announcements A_1, A_2, and A_3 in the order $A_1 \gg A_2 \gg A_3$. We encode this requirement with the following constraint:

$$Pref(A_1,A_2) \land Pref(A_2,A_3) \land \big(\forall i \in [4,..,n].Pref(A_3,A_i)\big)$$

Note that simpler requirements that do not stipulate a particular order are a special case. For example, if a requirement stipulates that an announcement A_k is selected as the best route, the above constraint becomes:

$$\forall i \in [1,n].\ k \neq i \implies Pref(A_k,A_i)$$

Overall Synthesis Algorithm Putting all pieces together, the complete algorithm employed by NetComplete to synthesize concrete BGP policies is as follows:

Step 1 (§4.2): Construct a BGP propagation graph G_{bgp} from the given requirements and network topology.

Step 2 (§4.3): Encode the routers' BGP policy sketches. The result is a constraint φ_S over variables $\overline{S}$. Each concrete instantiation of the variables S identifies concrete BGP policies.

Step 3 (§4.4): Declare symbolic variables $\overline{A}$ to represent all announcements propagated through the BGP propagation graph.
Propagate all symbolic announcements through the policy sketches. The result is an SMT constraint $\varphi_{announcements}$ over the variables $\overline{S}$ and $\overline{A}$.

Step 4 (Synthesis without additional OSPF requirements): Encode the route selection process and the requirements with the selection predicate *PrefNoIGP*, resulting in SMT constraints φ_{select} and φ_{req} over the variables $\overline{A}$. If a model of $\varphi_{select} \land \varphi_{req}$ exists, then derive concrete BGP policies and return; otherwise, go to Step 5.

Step 5 (Synthesis with additional OSPF requirements): Find the unsatisfiable core of $\varphi_{select} \land \varphi_{req}$ and derive a set S of pairs (A_1, A_2) of announcements that *cannot* be correctly selected without considering their IGP costs. Modify the constraint to:

$$\Big(\bigwedge_{(A_1,A_2)\in S} \texttt{IGPCost}_{A_1} < \texttt{IGPCost}_{A_2}\Big) \Rightarrow \varphi_{select} \land \varphi_{req}$$

If a model of this constraint exists, then derive BGP policies, create OSPF requirements from the set S, and return; otherwise, return that the requirements cannot be satisfied.

5 OSPF Synthesis

We now present NetComplete's OSPF synthesizer. OSPF is a Dijskstra-based routing protocol that forwards traffic along the shortest path, where path costs are computed based on the OSPF cost attached to each link.

OSPF Requirement:

$$\begin{aligned}&(AS100 \to A \to C \to AS300\\&\quad = AS100 \to A \to D \to C \to AS300)\\&\gg AS100 \to A \to B \to C \to AS300\end{aligned}$$

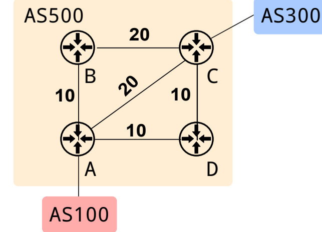


Figure 6: Example of correct assignment of link costs with respect to OSPF requirements.

NetComplete features a new *counter-example guided inductive synthesis* (CEGIS) [26] algorithm for OSPF that, given a set of OSPF requirements and a network topology, outputs OSPF link costs that enforce the requirements. Our algorithm can be tailored to support other Dijkstra-based routing protocols, such as IS-IS [27].

5.1 SMT Encoding

We phrase the OSPF synthesis problem as a constraint solving problem as follows. For any link that connects two nodes R to R' we introduce an integer variable $C_{R,R'}$ to represent the cost of link $R \to R'$. The cost of a path is given by the sum of the link costs along that path. For example, the cost of $AS100 \to A \to B \to C \to AS300$, denoted by $Cost(A \to B \to C)$, is $C_{A,B} + C_{B,C}$. We also denote the (finite) set of all simple paths between two nodes R and R' with $Paths(R,R')$. We can encode that the path $P = AS100 \to A \to C \to AS300$ has the lowest cost among all other simple paths from $AS100$ to $AS300$ via:

$$\forall X \in Paths(AS100,AS300) \setminus \{P\}.\ Cost(A \to C) < Cost(X)$$

We can directly use this method to encode the enforcement of OSPF requirements; see Fig. 6. For our example requirements, we obtain:

$$\begin{aligned}&Cost(A \to C) = Cost(A \to D \to C)\\&\quad \land \big(Cost(A \to C) < Cost(A \to B \to C)\big)\\&\quad \land \big(\forall X \in Paths(AS100,AS300) \setminus S.\\&\qquad\quad Cost(A \to B \to C) < Cost(X)\big), \text{ where}\\&S = \{A \to C, A \to D \to C, A \to B \to C\}\end{aligned}$$

This constraint captures that: *(i)* $AS100 \to A \to C \to AS300$ and $AS100 \to A \to D \to C \to AS300$ must have equal costs, *(ii)* path $AS100 \to A \to C \to AS300$ has lower cost than $AS100 \to A \to B \to C \to AS300$, and *(iii)* all other paths have higher cost than

$AS100 \rightarrow A \rightarrow B \rightarrow C \rightarrow AS300$. Note, in this example, $Paths(AS100, AS300) = S$. Therefore, we have $Paths(AS100, AS300) \setminus S = \emptyset$ and condition *(iii)* vacuously holds.

Naive OSPF Synthesis A naive synthesis solution is to encode all requirements with constraints, as described above, and to then use a constraint solver to discover a model that identifies correct link costs. Unfortunately, phrasing the OSPF synthesis problem directly into SMT does not scale to large networks; cf. [20]. The main issue is the for-all ($\forall$) quantifier in the constraints used to encode that a path has a lower cost among all other simple paths with the same source and destination.

5.2 Counter-Example Guided Inductive Synthesis for OSPF

We now present our new counter-example guided inductive synthesis (CEGIS) algorithm for OSPF. CEGIS is a contemporary approach to synthesis, where a correct solution is iteratively learned from counter-examples [26]. CEGIS algorithms tend to work quite well in practice because often a small number of counter-examples (that is, few iterations) is sufficient to discover a correct solution.

The OSPF synthesis problem amounts to finding a model of logical constraints of the form:

$$\exists \overline{C}.\ \textsc{EncodeOSPF}(\overline{C}, r, Paths(r))$$

where $\overline{C}$ is the set of variables that represent link costs, r is an OSPF requirement, $Paths(r)$ is the set of all paths from the source *Src* and destination *Dst* provided in the requirement r, and $\textsc{EncodeOSPF}(\overline{C}, r, Paths(r))$ returns a logical formula that encodes the requirement's satisfaction (as described in §5.1). Finding a model of this formula directly using a constraint solver is difficult due to the large number of paths in $Paths(r)$. To avoid this quantifier, CEGIS restricts the constraint to a (small) set of paths $S = \{P_1, \dots, P_n\} \subseteq Paths(r)$. The resulting constraint is:

$$\exists \overline{C}.\ \textsc{EncodeOSPF}(\overline{C}, r, S)$$

which is easier to solve by existing constraint solvers. A model of this constraint identifies link costs that imply that the requirement holds over the paths in S. However, it may not hold over all paths in $Paths(r)$. The idea of CEGIS is to check the requirement over all paths and to obtain a concrete counter-example that violates it, if one exists; we remark that the step of checking is usually efficient. The set S is then iteratively expanded with counter-examples until a correct solution is found.

Algorithm We show the main steps of our CEGIS algorithm in Alg. 1. For each requirement $r \in Reqs$, the algorithm declares a set S_r (line 3). The algorithm then iteratively repeats the following steps. For each requirement $r \in Reqs$, the algorithm samples b paths from the source to the destination of the requirement r and adds these to S_r (line 7). It then encodes the requirement's satisfaction with respect to S_r (line 8) and conjoins the result to φ (line 9). If the resulting constraint φ is unsatisfiable, it means the requirements cannot be satisfied and the algorithm returns $\bot$ to indicate this. Otherwise, it obtains a model M of the constraints φ (line 12), which defines a concrete value for each link cost variable.

The algorithm then checks whether these costs defined by M enforce the requirements *Reqs* (over all paths). If the requirements are satisfied, the algorithm returns $M(\overline{C})$ (line 14), i.e. it returns the values associated to the link cost variables $\overline{C}$. Otherwise, it obtains a concrete counter-example as a pair $(r, path)$ of a path *path* that violates a requirement r, and expands the set S_r with *path* (line 16). This ensures that the counter-example is avoided in the next iteraion. Further, to reach a solution faster, the algorithm samples additional b paths for each requirement r and adds them to S_r. These steps are repeated until a solution is found or the requirements are deemed unsatisfiable.

Algorithm 1: CEGIS algorithm for synthesizing OSPF link costs with respect to OSPF requirements.

```
Input: OSPF requirements Reqs = ⋃_i r_i, link cost
       variables C̄, bound b
Output: OSPF link costs
1  begin
2    for r ∈ Reqs do
3      S_r = ∅
4    while true do
5      φ = true
6      for r ∈ Reqs do
7        S_r ← S_r ∪ SamplePaths(r, b)
8        φ_r ← EncodeOSPF(C̄, r, S_r)
9        φ ← φ ∧ φ_r
10     if Unsat(φ) then
11       return ⊥
12     M ← Model(φ)
13     if CheckReqs(M, Reqs) then
14       return M(C̄)
15     (r, path) ← CounterExample(M, Reqs)
16     S_r ← S_r ∪ {path}
```

6 Implementation and Evaluation

We implemented NetComplete in around $10K$ lines of Python code using SMT-LIB v2 [28] and Z3 [21]. Our implementation is based on the theories of linear in-

Network	Req.	2 requirements				8 requirements				16 requirements			
		50% symbolic		100% symbolic		50% symbolic		100% symbolic		50% symbolic		100% symbolic	
size	type	CEGIS	Naive	CEGIS	Naive	CEGIS	Naive	CEGIS	Naive	CEGIS	Naive	CEGIS	Naive
Small	Simple	0.41	0.93	0.43	1.04	1.66	2.42	1.67	2.73	3.33	6.95	3.39	8.02
	Any-path	0.62	2.00	0.67	2.38	2.31	12.27	2.38	14.48	4.63	7.22	4.76	8.58
	ECMP	0.48	0.84	0.53	0.94	1.72	5.02	1.77	5.76	3.44	3.16	3.48	3.61
	Ordered	0.55	0.54	0.68	0.64	2.90	2.93	5.49	3.50	4.76	5.07	7.93	6.05
Medium	Simple	0.79	790.04	0.81	1554.81	3.06	19613.55	3.10	20.60	6.17	3238.46	6.18	6039.24
	Any-path	1.27	1677.30	1.28	4208.68	4.89	18758.02	4.94	66.10	9.70	107.13	9.83	122.68
	ECMP	0.85	567.02	0.86	1370.70	3.16	5643.60	3.24	22272.88	6.34	45.32	6.39	51.61
	Ordered	1.76	450.64	2.81	732.60	30.83	2942.83	33.60	8636.21	31.08	49.43	43.63	58.54
Large	Simple	1.78	$>$ 24h	1.85	$>$ 24h	7.35	$>$ 24h	7.40	$>$ 24h	13.90	$>$ 24h	14.03	$>$ 24h
	Any-path	4.23	$>$ 24h	4.33	$>$ 24h	16.59	$>$ 24h	16.89	$>$ 24h	32.61	$>$ 24h	33.01	$>$ 24h
	ECMP	1.83	$>$ 24h	1.89	$>$ 24h	7.07	$>$ 24h	7.14	$>$ 24h	13.37	$>$ 24h	13.52	$>$ 24h
	Ordered	6.90	$>$ 24h	15.00	$>$ 24h	33.81	$>$ 24h	44.72	$>$ 24h	249.48	$>$ 24h	1155.19	$>$ 24h

Figure 7: Using Counter-Example Guided Inductive Synthesis (CEGIS) to synthesize OSPF weights is *considerably* faster than a naive OSPF algorithm which aims to solve all constraints at once.

teger arithmetic and quantifier-free uninterpreted functions. Our prototype takes as input partial configurations (combining OSPF, BGP, and static routes) and outputs completed ones. We support standard Cisco commands for setting OSPF costs and BGP policies and can easily extend our code base to support other languages.

In the following, we show that our NetComplete implementation is practical and scales to realistic networks. Specifically, we measure: (i) NetComplete OSPF and BGP synthesis times in growing network topologies; (ii) the impact of having more or less symbolic variables in the sketches; and (iii) how NetComplete compares against competing approaches such as SyNET [20].

6.1 Methodology and datasets

Topologies We sample 15 network topologies from Topology Zoo [29] that we classify according to their size: *small* (from 32 to 34 routers), *medium* (from 68 to 74 routers), and *large* (from 145 to 197 routers). We select 5 topologies per category.

Requirements We generate four types of routing requirements (simple, any-path, ECMP, and ordered) in each topology. Each requirement is defined between a randomly selected source *Src* and destination *Dst* pair. For simple path requirements, we choose a random feasible path from *Src* to *Dst*. For the other requirements, we first choose two paths P_1 and P_2 from *Src* to *Dst* and then we construct (P_1, P_2) for any-path requirements, $(P_1 = P_2)$ for ECMP, or $P_1 \gg P_2$ for ordered requirements. For each topology, we generate multiple sets of requirements of size 2, 8, and 16. We generate all four types of requirements for the OSPF evaluation, and only generate simple and ordered path requirements for the BGP evaluation. Indeed, any-path and ECMP requirements are typically internal requirements and are therefore typically enforced by IGP protocols.

Sketches We construct configuration sketches for each topology from a fully concrete configuration (which we synthesize using NetComplete) for which we randomly make a given percentage of the variables symbolic. For instance, to generate partial OSPF (resp. BGP) configurations that are 50% symbolic, we randomly make 50% of the edges (resp. BGP import/export policies) in the synthesized concrete configurations symbolic.

Validation We validate that our synthesized configurations comply with the corresponding requirements in an emulated environment composed of Cisco routers [30].

6.2 Results

We now present our results focusing first on OSPF synthesis, before considering BGP synthesis, and finishing with a comparison with SyNET. We run all our experiments on a server with 128GB of RAM and a 12-core dual-processors running at 2.3GHz. Unless indicated, we report averaged results over 5 runs and across topologies of the same class.

OSPF Synthesis We first illustrate the effectiveness of synthesizing OSPF configuration using our CEGIS algorithm versus a naive algorithm in which the entire $\exists\forall\varphi$ constraint is directly fed to the solver. We then evaluate how sketches affect synthesis time.

Our results are reported in Fig. 7 and convey four important insights. *First*, CEGIS significantly outperforms naive OSPF synthesis, especially in large networks where naive synthesis does not even terminate within a day. *Second*, we see that the synthesis time is proportional to the topology size and the number of requirements. Indeed, the number of symbolic variables

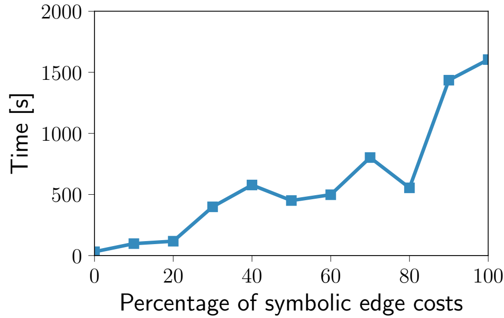


Figure 8: NetComplete synthesizes ordered path requirements faster when the configuration sketch provides more concrete values for edge costs.

Topo	Req. type	Total	16 reqs. Min % Eval	Max % Eval
Small	Simple	58578	9.62%	18.76%
	Ordered	37662	16.75%	18.76%
Medium	Simple	98683	7.27%	13.54%
	Ordered	58924	10.02%	22.81%
Large	Simple	83832	11.93%	14.57%
	Ordered	29565	22.56%	25.07%

Figure 9: Number of generated symbolic variables. Thanks to partial evaluation, NetComplete is able to evaluate between 7% and 25% of the symbolic variables—making BGP synthesis significantly faster.

is equal to the number of symbolic edge costs, while the number of constraints is proportional to the requirements size and the number of available paths. *Third*, ordered path requirements take more time to synthesize than the other requirements. This is expected as such requirements specify a strict sequence of paths making the search space more sparse. *Fourth*, the use of more concrete values significantly reduces the synthesis time, especially for ordered path requirements with reductions up to 70%. We further illustrate this behavior in Fig. 8 which depicts the times required to synthesize 16 ordered path requirements for the large networks as a function of the percentage of symbolic values. We see that NetComplete indeed leverages the concrete variables and the reduced search space to synthesize configurations faster.

BGP Synthesis We now evaluate the effectiveness of the BGP synthesizer and how it leverages partial evaluation to concretize up to 25% of the symbolic variables and therefore speed up the overall synthesis time.

In Fig. 9, we show the average number of generated symbolic variables for each group (see Appendix C for detailed numbers). We see that the number of generated symbolic variables is not directly related to the topology size and the number of requirements: the number of variables for medium topologies can exceed the ones of larger topologies. For BGP, the number of variables indeed depends on: (i) the number of routers (and their connectivity) in the computed propagation graph; (ii) the complexity of the configuration sketch; and (iii) the effectiveness of partial evaluation.

Regarding partial evaluation, we observe that NetComplete manages to evaluate between 7% and 25% of the generated symbolic variables (Fig. 9), which makes BGP synthesis proportionally faster. Indeed, in Fig. 10, we show how the BGP synthesis time evolves linearly as a function of the number of symbolic variables. We also see that NetComplete always manages to synthesize BGP configurations in less than 14min.

Comparison to SyNET We now compare the synthesis time of NetComplete to SyNET. Specifically, we compare NetComplete and SyNET running times for the worst-case scenario reported in [20] involving 10 requirements defined in topologies with 49 and 64 routers. Since SyNET defines requirements in terms of the number of traffic classes and not forwarding paths as NetComplete, we first translate each traffic class to a set of simple path requirements. To ensure a fair comparison, we provide NetComplete with entirely symbolic sketch since SyNET does not accept sketches.

Our results (Fig. 11) shows that NetComplete is at least 600× faster than SyNET and is able to synthesize configurations for larger topologies that SyNET timed out on. This speed up stems from two factors. First, NetComplete does not use an SMT solver for the requirements that it can solve directly (such as synthesizing static routes). Second, NetComplete relies on domain-specific heuristics (CEGIS and partial evaluation) to reduce the search space, while SyNET relies on the generic optimizations of the underlying SMT solver.

7 Related Work

Intent-based Networking and SDN Languages The importance of relying on high-level abstractions in network management has received considerable attention, specifically in the context of Software-Defined Networking (SDN) [18, 25, 31, 32, 33, 34, 35, 36, 37]. This influence goes beyond academic with two of the largest SDN controllers (ONOS and OpenDayLight) now providing declarative network management [38, 39].

Our work brings programmability to traditional networks, by enabling operators to enforce policies expressed in high-level SDN-like languages such as Genesis [18] or Frenetic [25]. Our work, therefore, complements the above initiatives and enables them to be used

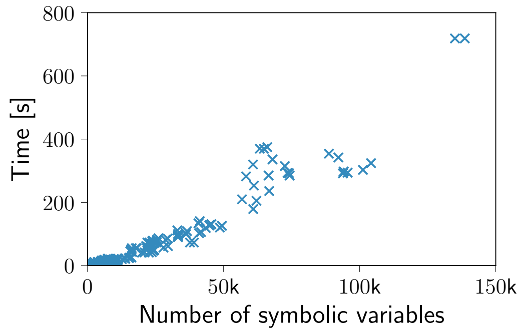


Figure 10: BGP synthesis time grows linearly with respect to the number of symbolic variables.

# Rtrs	Protocol	SyNET	NetComplete
49	Static	14m11s	0.05s
	Static + OSPF	5h22m56s	2m1s
	Static + OSPF + BGP	timeout (> 24h)	44m2s
64	Static	49m22s	0.06s
	Static + OSPF	21h13m16s	2m22s
	Static + OSPF + BGP	timeout (> 24h)	6h6m30s

Figure 11: NetComplete is $> 600\times$ faster than [20].

beyond OpenFlow or P4-enabled networks.

Network Verification Network verification approaches are used to check a configuration with respect to requirements. FSR [40] encodes BGP preferences using routing algebra [41] and verifies safety properties (e.g., BGP stability) using SMT solvers. Batfish [6] encodes routing protocols in Datalog and uses Datalog solvers to check conformance with routing requirements. Bagpipe [9] and Minesweeper [8] formalize BGP and present an analyzer for BGP configurations. In contrast, NetComplete focuses on synthesis, which subsumes verification.

Network Configuration Synthesis Recently, multiple works have aimed at synthesizing configurations out of high-level requirements [14, 15, 16, 20].

ConfigAssure [14] supports requirements expressed using first-order constraints. As shown in our evaluation, a direct encoding of routing computations into constraints goes beyond what existing solvers can handle.

Route Shepherd [41, 42] takes a partial specification of BGP preferences and derives constraints over link costs that capture the absence of BGP instability. In contrast, NetComplete models the derivation of BGP preferences and also synthesizes the BGP configuration.

Propane [15] and PropaneAT [16] produce BGP configurations out of high-level requirements. Having the freedom to output any configuration enables these systems to use templates and, in turn, to scale to large networks. In contrast, NetComplete supports partial configurations for multiple protocols (OSPF, BGP, and static routes), which prevents us from leveraging specific templates. While NetComplete pays for this flexibility in terms scalability, it is still fast, synthesizing configurations within seconds.

SyNET [20] is another network-wide configuration synthesizer supporting multiple protocols. It differs from NetComplete in two ways. First, SyNET supports any protocol that can be specified in stratified Datalog [43], while NetComplete supports specific protocols (OSPF, BGP). Since BGP cannot be fully captured in stratified Datalog, SyNET supports a simplified BGP though, while NetComplete supports it fully. Second, SyNET uses a generic synthesis procedure for Datalog, while NetComplete uses custom procedures for each protocol. Consequently, SyNET does not scale to large networks and is orders of magnitude slower than NetComplete.

Synthesizers such as NetEgg [19, 44] and NetGen [17] target SDN environments and aim to derive controller programs (instead of configurations) out of requirements. While their goal is similar to ours, our target is different (distributed protocols vs. centralized controller).

Program Synthesis Our work also relates to program synthesis. In particular, we showed a novel instantiation of counter-example guided inductive synthesis (CEGIS) [26] for synthesizing weights in OSPF. CEGIS is a general concept that has become popular in the program synthesis community. A key challenge in using it is finding effective ways to specialize it (e.g., efficient representation of the hypothesis space, interaction with the SMT solver) to the particular application domain (e.g., networking and the OSPF protocol in our case).

8 Conclusion

We presented NetComplete, the first scalable network-wide configuration synthesizer to support multiple protocols and a partial sketch of the desired configuration.

NetComplete features a new BGP synthesis procedure that supports BGP configuration sketches and partial computations over symbolic announcements. It also introduces an efficient synthesis procedure for the widely-used OSPF protocol. This procedure is based on counter-example guided inductive synthesis and achieves significant speedups (> 100x) over existing solutions.

Finally, we presented a comprehensive set of experimental results, which demonstrate that NetComplete can autocomplete configurations for large networks with up to 200 routers within few minutes.

Acknowledgments

We are grateful to our shepherd, Vyas Sekar, and the anonymous reviewers for their constructive feedback.

References

[1] Stock trading closed on NYSE after glitch caused major outage. https://www.theguardian.com/business/live/2015/jul/08/new-york-stock-exchange-wall-street.

[2] Google routing blunder sent Japan's Internet dark on Friday. https://www.theregister.co.uk/2017/08/27/google_routing_blunder_sent_japans_internet_dark/, 2017.

[3] Facebook, Tinder, Instagram suffer widespread issues. http://mashable.com/2015/01/27/facebook-tinder-instagram-issues/.

[4] United Airlines jets grounded by computer router glitch. http://www.bbc.com/news/technology-33449693.

[5] Juniper Networks. Whats Behind Network Downtime? Proactive Steps to Reduce Human Error and Improve Availability of Networks. Technical report, May 2008.

[6] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015.

[7] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference SIGCOMM '16*, 2016.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *Proceedings of the 2017 ACM SIGCOMM Conference SIGCOMM '17*, 2017.

[9] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA'16)*, 2016.

[10] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, 2012.

[11] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, 2013.

[12] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference SIGCOMM '16*, 2016.

[13] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking Beliefs in Dynamic Networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015.

[14] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative Infrastructure Configuration Synthesis and Debugging. *Journal of Network and Systems Management*, 16(3):235–258, 2008.

[15] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference SIGCOMM '16*, 2016.

[16] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Network Configuration Synthesis with Abstract Topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '17*, 2017.

[17] Shambwaditya Saha, Santhosh Prabhu, and P Madhusudan. NetGen: Synthesizing Data-plane Configurations for Network Policies. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research SOSR '15*, 2015.

[18] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages POPL '17*, 2017.

[19] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. Scenario-based Programming for SDN Policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies CoNEXT '15*, 2015.

[20] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-Wide Configuration Synthesis. In *Proceedings of the 29th International Conference on Computer Aided Verification CAV '17*. Springer, 2017.

[21] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS '08*, 2008.

[22] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A Network-State Management Service. In *Proceedings of the 2016 ACM SIGCOMM Conference SIGCOMM '15*, 2015.

[23] Nanxi Kang, Ori Rottenstreich, Sanjay G Rao, and Jennifer Rexford. Alpaca: Compact Network Policies With Attribute-Encoded Addresses. *IEEE/ACM Transactions on Networking*, 2017.

[24] G Gonzalo et al. *Network Mergers and Migrations: Junos Design and Implementation*, volume 45. John Wiley & Sons, 2011.

[25] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming ICFP '11*, 2011.

[26] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems ASPLOS XII*, 2006.

[27] R. W. Callon. RFC 1195: Use of OSI IS-IS for Routing in TCP/IP and Dual Environments, 1990.

[28] C. Barrett et al. The SMT-LIB Standard: Version 2.0, 2010.

[29] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Alistair Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 2011.

[30] Graphical Network Simulator-3 (GNS3). `https://www.gns3.com/`.

[31] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the 2015 ACM SIGCOMM Conference SIGCOMM '15*, 2015.

[32] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software Defined Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, 2013.

[33] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Proceedings of the 2016 ACM SIGCOMM Conference SIGCOMM '13*, 2013.

[34] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014.

[35] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN Symposium on Principles of Programming Languages POPL '14*, 2014.

[36] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A Coalgebraic Decision Procedure for NetKAT. In *Proceedings of the 42st ACM SIGPLAN Symposium on Principles of Programming Languages POPL '15*, 2015.

[37] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM Conference on Emerging Networking Experiments and Technologies CoNEXT '14*, 2014.

[38] Open Network Operating System (ONOS) Intent Framework. `https://wiki.onosproject.org/display/ONOS/The+Intent+Framework`.

[39] OpenDayLight (ODL) Group-Based Policy. `https://wiki.opendaylight.org/view/Group_Policy:Main`.

[40] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott. FSR: Formal Analysis and Implementation Toolkit for Safe Interdomain Routing. *IEEE/ACM Transactions on Networking*, 20(6):1814–1827, 2012.

[41] Alexander J., T. Gurney, Anduo Wang Limin Jia, and Boon Thau Loo. Partial Specification of Routing Configurations. In *Workshop on Rigorous Protocol Engineering*, 2011.

[42] Alexander J.T. Gurney, Xianglong Han, Yang Li, and Boon Thau Loo. Route Shepherd: Stability Hints for the Control Plane. In *Proceedings of the 2016 ACM SIGCOMM Conference SIGCOMM '12*, 2012.

[43] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[44] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. NetEgg: Programming Network Policies by Examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks HotNets '14*, 2014.

$PrefNoIGP(A_1, A_2) \Leftrightarrow$

```
// 0) A1 is received and A2 is dropped
```
$(\texttt{permitted}_{A_1} \wedge \neg\texttt{permitted}_{A_2})$
```
// 1) Higher local preference
```
$\vee(\texttt{permitted}_{A_1} \wedge \texttt{permitted}_{A_2} \wedge (\texttt{LocalPref}_{A_1} > \texttt{LocalPref}_{A_2}))$
```
// 2) Lower AS path length
```
$\vee(\texttt{permitted}_{A_1} \wedge \texttt{permitted}_{A_2} \wedge (\texttt{LocalPref}_{A_1} = \texttt{LocalPref}_{A_2})$
$\wedge (\texttt{AsPathLen}_{A_1} > \texttt{AsPathLen}_{A_2}))$

$\vdots$
```
// 5) Prefer routes learned over eBGP
```
$\vee(\texttt{permitted}_{A_1} \wedge \texttt{permitted}_{A_2} \wedge (\texttt{LocalPref}_{A_1} = \texttt{LocalPref}_{A_2})$
$\cdots \wedge (\texttt{eBGP}_{A_1} \wedge \neg\texttt{eBGP}_{A_2}))$

$Pref(A_1, A_2) \Leftrightarrow PrefNoIGP(A_1, A_2) \vee$
```
// 6) Lower IGP cost
```
$(\texttt{permitted}_{A_1} \wedge \texttt{permitted}_{A_2} \wedge (\texttt{LocalPref}_{A_1} = \texttt{LocalPref}_{A_2})$
$\cdots \wedge (\texttt{IGPCost}_{A_1} < \texttt{IGPCost}_{A_2}))$

Figure 12: SMT encoding of the preference over BGP announcements with the same prefix

A SMT Encoding of the BGP Selection Process

Encoding Selection of Announcements When a BGP router receives different announcements for the same prefix, it uses the preference ordering to select the best route. We encode the logic used by routers to select the best route in Fig. 12. The predicate $PrefNoIGP(A_1, A_2)$ holds if and only if announcement A_1 is preferred over A_2 without considering the IGP costs of A_1 and A_2. The constraint is a disjunction over the different cases defined by the BGP selection process. First, if A_2 is dropped, then A_1 is selected as a best route. Second, if both announcements are permitted, the router selects A_1 over A_2 if A_1's local preference is lower than that of A_2. Analogously, the constraint encodes cases $3-5$ described in §4.4.

In addition, we define the constraint $Pref(A_1, A_2)$ that also compares the announcements' IGP costs. The constraint $Pref(A_1, A_2)$ holds if announcement A_1 is preferred over A_2 without considering IGP costs (i.e., $PrefNoIGP(A_1, A_2)$ holds) or the IGP cost of A_1 is lower than that of A_2.

B SMT Encoding of BGP sketches

We illustrate the encoding of BGP sketches using SMT constraints. Consider the following BGP sketch:

```
1 AttributesSketch
2   match next-hop AS200
3   set local-pref ? < 50
```

This sketch would match any announcement that has the value $AS200$ set for the next hop attribute. If an announcement is matched, this policy would set the local preference of the output announcement to a value that is yet to be synthesized by the BGP synthesizer. As defined by the sketch, this local preference value must be smaller than 50. Note that this BGP policy does not change the remaining attributes (as there are no further actions).

We encode this BGP sketch as follows:

$$\begin{aligned} &\textbf{if } \texttt{NextHop}_{A_{in}} = \text{AS200} \\ &\quad \textbf{then } ((\texttt{LocalPref}_{A_{out}} = Var_1) \wedge (0 < Var_1 < 50) \\ &\qquad\qquad \wedge (\forall X \in Attrs \setminus \{\texttt{LocalPref}\}.\, X_{A_{out}} = X_{A_{in}})) \\ &\quad \textbf{else } \quad \forall X \in Attrs.\, X_{A_{out}} = X_{A_{in}} \end{aligned}$$

where $Attrs = \{\texttt{NextHop}, \ldots\}$ and Var_1 are fresh variables

Here, the variable Var_1 represents the local preference value that will be set by the BGP policy. A_{in} represents the input announcement (before it is processed by the BGP policy) and A_{out} the output one. The constraint formalizes that only input announcements with next hop equal to $AS200$ are matched. For matched announcements, the *then* constraint encodes that the output announcement has local preference set to Var_1, which is a value smaller than 50, and all remaining attributes are identical to those in the input announcement (and thus remain unchanged). Finally, the *else* constraint ensures that if an announcement is not matched (its local preference is not $AS200$), then all attributes remain unchanged.

C Symbolic Variables in BGP Synthesis

In Fig. 13, we show the number of generated symbolic variables when synthesizing BGP configurations for each topology we used in our data set. We observe that the number of generated variables depends on the number of routers (and their connectivity) in the computed propagation graph, the complexity of the configuration sketch, and on the effectiveness of partial evaluation.

Size	Topo	Req. type	2 reqs. Total	2 reqs. 100%	2 reqs. 50%	8 reqs. Total	8 reqs. 100%	8 reqs. 50%	16 reqs. Total	16 reqs. 100%	16 reqs. 50%
Small	Arnes	simple	1997	488	482	13565	2466	2397	68355	7205	7126
		order	962	224	200	11759	2394	2352	35993	6541	6380
	Bics	simple	2045	515	503	17960	2592	2553	51890	6930	6760
		order	758	206	197	10313	2462	2423	40581	7302	7223
	Canerie	simple	764	210	198	16451	2569	2478	55787	6394	6233
		order	1238	316	300	10775	2457	2424	40243	6981	6981
	CrlNet	simple	653	179	170	13112	2078	2078	41693	5476	5288
		order	854	228	210	6728	1634	1562	27607	5204	5055
	Renater	simple	2717	669	651	18983	2884	2884	75167	7189	7104
		order	758	206	188	13283	2642	2558	43884	7333	7251
Medium	Columbus	simple	2057	556	540	13400	2520	2421	52672	7159	7010
		order	854	228	219	6728	1634	1595	28545	6510	6361
	Esnet	simple	2324	543	536	28403	4206	4110	105879	10298	10125
		order	1526	382	370	13295	2888	2849	51135	9885	9805
	Latnet	simple	5111	1052	1043	44012	4921	4825	149394	10778	10590
		order	3530	837	834	27626	3990	3903	104606	10482	10482
	Sinet	simple	1139	293	281	33905	4339	4230	114648	10466	10278
		order	2966	712	703	29552	4978	4933	77823	11276	11203
	Uninett2011	simple	2705	696	678	24275	3317	3224	70821	8752	8585
		order	1610	397	388	14333	3011	2933	32511	7122	7122
Large	Cogentco	simple	3293	837	828	22565	4420	4348	85708	11441	11371
		order	1046	272	272	7115	1819	1743	29726	6982	6821
	Colt	simple	3578	866	845	47795	6087	6024	85997	12524	12362
		order	662	184	184	9992	2544	2475	33887	7819	7737
	GtsCe	simple	3566	861	861	29627	4948	4855	67705	9450	9356
		order	854	228	210	8621	2060	2023	31073	7011	7011
	TataNld	simple	2348	624	608	20330	3861	3822	75380	10575	10405
		order	662	184	181	6650	1680	1602	31424	7393	7316
	UsCarrier	simple	1460	412	391	19643	3776	3737	104371	12202	12017
		order	758	206	199	5438	1445	1415	21715	5440	5361

Figure 13: The number of symbolic variables generated for each topology and the number of partially evaluated variables when the configuration sketch is 100% and 50% symbolic.

Automatically Correcting Networks with *NEAt*

Wenxuan Zhou, Jason Croft, Bingzhe Liu, Elaine Ang, Matthew Caesar
University of Illinois at Urbana-Champaign
{*wzhou10, croft1, bingzhe, ranang2, caesar*}@*illinois.edu*

Abstract

Configuring and maintaining an enterprise network is a challenging and error-prone process. Administrators often need to consider security policies from a variety of sources such as regulatory requirements, industry standards, and mitigating attack vectors. Erroneous configuration or network application could violet crucial policies, and result in costly data breaches and intrusions. Relying on humans to discover and troubleshoot violations is slow and prone to error, considering the speed at which new attack vectors propagate and the increasing network dynamics, partly an effect of SDN.

To address this problem, we present *NEAt*, a system analogous to a smartphone's autocorrect feature that enables on-the-fly repair to policy-violating updates. It does so by modifying the forwarding behavior of updates to automatically repair violations of policies such as reachability, service chaining, and segmentation. *NEAt* takes as input a set of administrator-defined high-level policies, and formulates these policies as directed graphs. Sitting between an SDN controller and the forwarding devices, *NEAt* intercepts updates proposed by SDN applications. If an update violates a policy, *NEAt* transforms the update into one that complies with the policy. Unlike domain-specific languages or synthesis platforms, *NEAt* allows enterprise networks to leverage the advanced functionality of SDN applications while simultaneously achieving strong, automated enforcement of general policies. Based on a prototype implementation and experimentation using Mininet and operation trace of a large enterprise network we demonstrate that *NEAt* achieves promising performance in real-time bug-fixing.

1 Introduction

Modern enterprise networks must comply with highly stringent security demands that merge together demands from a variety of sources and standards. As a result, network administrators must carefully design and maintain their networks to follow these policies, by mapping out device contexts and access to sensitive resources, assessing risk, and installing access control policies that effectively mitigate that risk. However, mistakes and errors in implementing the policies can result in costly data breaches, segmentation violations, and infiltrations. Through 2020, Gartner predicts 99% of firewall breaches will be caused by misconfigurations [1, 2].

While discovering and troubleshooting these bugs is essential to maintaining network security, doing so is notoriously hard. Relying on humans to configure and maintain the network configuration is not only prone to mistakes, but slow. Given the sophistication and speed at which new attack vectors propagate, manually updating and testing new configurations leaves the network in a vulnerable state until the attack vector is fully secured. Further, maintaining a security posture in the presence of software-defined networking (SDN) is even more challenging. While SDN enables new functionality, application designers may not be aware of the policy or security requirements of the networks on which their applications will be deployed. Worse yet, SDN applications written in general-purpose languages such as Java or Python can be arbitrarily complex. Requiring applications to implement and modify their behavior to support a broad spectrum of policies needed across a broad spectrum of networks presents an almost insurmountable challenge.

To this end, we present *NEAt*, a transparent layer to automatically repair policy-violating updates in real-time. *NEAt* secures the network with a mechanism similar to a smartphone's autocorrect feature, which enables on-the-fly repair to policy violating updates and ensures the network is always in a state consistent with policy. Unlike prior work on update synthesis, *NEAt* maintains backward compatibility and flexibility to run general SDN application code. To do this, *NEAt* does not synthesize network state from scratch, but rather *influences* updates from an existing SDN application toward a correct specification. In particular, *NEAt* enforces a concrete definition of correctness by influencing and constraining dynamically arriving network instructions. To formulate those correctness criteria, we construct a set of *policy graphs* to represent humans' correctness intent, which is based on the observation that important error conditions can be caught by a concise set of boundary conditions. *NEAt* sits between an SDN controller and the forwarding devices, and intercepts the updates proposed by the running SDN applications. If the update violates an administrator's defined policy, such as reachability or segmentation, *NEAt* transforms the update into

one that complies with the policy.

A key challenge is discovering update repairs in real-time. In *NEAt*, we build on prior work on verification to efficiently model packet forwarding behavior as a set of Equivalence Classes (ECs) [19, 30]. Upon receiving an update from an SDN controller, *NEAt* computes the set of affected ECs and checks for a violation in the same manner as [19]. To repair the violation, we cast the problem as an optimization problem, to find the minimum number of changes (added or deleted edges) to repair the violating EC's forwarding graph. To rapidly compute repairs on arbitrarily large networks, we exploit two optimization techniques, *topology limitation* which "slices" away irrelevant part of the network, and *graph compression*, to compress both an EC's forwarding graph and the topology. Then we solve the optimization problem on the sliced and compressed graphs.

Furthermore, as *NEAt* repairs policy-violating updates, stateful applications — without knowledge of the violating or repaired updates — will diverge from the underlying network state. To address this problem, applications can interactively propose updates to *NEAt* and receive notifications of repairs with minor modifications to application code. Thus, applications can remain unmodified and leverage *NEAt* transparently in a *pass-through* mode, with a risk of state divergence, or propose updates in an *interactive* mode.

A preliminary evaluation of our prototype shows promising results. On topologies with up to 125 switches and 250 hosts, *NEAt* can discover repairs in under one second for applications with non-overlapping rules, and under two seconds for applications with more complex dependencies. Furthermore, we find *NEAt* can verify and repair updates on realistic data planes. On a large enterprise network with 1M forwarding rules, *NEAt* discovered and repaired 28 loop violations. Simulations on this data set show *NEAt* can verify and repair reachability and loop freedom policies in under a second.

2 Motivation

Enterprise network policies must compose together requirements from a variety of demands to mitigate risk for attack vectors and limit access to sensitive resources. As a result, network administrators must take into account complex, composed policies configuring or updating a network. This is a slow and often error-prone process for a human operator. The operator may introduce errors translating the demands into high-level policies, or translating the policies into low-level routing configurations. While tools [17, 19] exist to automatically discover misconfigurations in real-time, they offer the operator no guidance on how to repair the misconfiguration beyond the type of correctness property that is violated. Rather, these tools block updates from introducing violations into the data plane state, at the cost of functionality.

Instead, a system to automatically repair updates, ensuring the network always remains consistent with the administrator's policy, can relieve a slow and error-prone process from the configuration process. If an update violates a given property in the network, a *repair* should fix the cause of the violation while maintaining the original purpose of the update. We argue a minimal change is best, to repair the update with the least number of added or removed edges. Furthermore, such a system should improve upon a manual effort with transparency in both architecture and performance. A system that requires hours or days to verify and repair a network is not useful if the process can be completed manually in just a few minutes. It should also not require modifying existing applications or redesigning infrastructure.

Efficiently discovering repairs is not a trivial addition on top of data plane verification tools, such as [17, 19]. Due to the size of the network and data plane state, performance is a key challenge in repairing policy violations in real-time. Consider a naive approach built on top of VeriFlow that separates the forwarding behavior into Equivalence Classes (ECs) of packets. All packets within an EC are forwarded in precisely the same manner. Each EC defines a *configuration graph* that captures the the forwarding packets for packets within the EC. The number of ECs is dependent on the number of devices and forwarding rules in the network, and the time to discover a repair is dependent on the number of ECs and the number of edges in the network. A brute force approach might discover repairs by testing all permutations of edge additions and removals to an EC's configuration graph. A repair that requires only adding edges, from 10 possible unused topology edges, would need to explore 10! (~3.6M) permutations. If the violating property can be checked in just 1ms, each EC could take up to 10 minutes to find a repair.

3 Design

NEAt operates between the controller and switches, intercepting and verifying updates against a set of correctness properties specified by a network operator. *NEAt* takes these properties as input in the form of a directed graph called a *policy graph* (① in Figure 1). Policy graphs can express properties including reachability, segmentation, and waypointing, as described in §4.

To verify updates conform to the operator's intended policies, each update is applied to a model of the data plane state and checked using *NEAt*'s verification engine. a policy violation, the correction engine transforms the update or existing data plane state to satisfy the violated policy. This ensures only updates conforming to

the network policies are sent onto the network.

NEAt can integrate with the existing SDN control infrastructure in two ways. It can serve as a transparent layer in a *pass-through mode*, or can interact with controller applications in an *interactive mode* with minor changes to the applications.

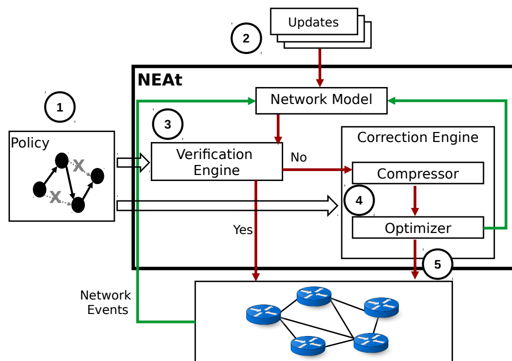


Figure 1: *System architecture of NEAt.*

3.1 Verification and Repair

To verify updates and efficiently reason about the current data plane states, *NEAt* builds on our previous work in data plane verification [19,30] that separates the forwarding behavior into Equivalence Classes (ECs) of packets. From each EC, we can extract a *configuration graph* that defines the forwarding behavior for packets within the EC. A repair for a given EC must then explore additions or deletions of links in the configuration graph. Finding a link addition requires examining the *topology graph* defined by the edges in the physical topology. To efficiently discover repairs, we propose two optimization techniques to compress the configuration and topology graphs, described in §6. We refer to the outcome of these techniques as the *compressed configuration graph* and *compressed topology graph*.

With each update (②), *NEAt* applies the change to a network model, from which the ECs affected by the update are computed. Using the policy graph, *NEAt* checks each affected EC in the network model for policy violations using the verification engine (③). If the update does not introduce any violations, it is sent onto the network. However, if it does introduce a violation, the configuration graph and topology graph are compressed and passed to the correction engine (④). The optimizer returns a set of edges to be added or removed to the EC's configuration graph, which are then applied to the network model, converted to OpenFlow rules, and sent to the forwarding devices (⑤).

3.2 Interaction Modes

To prevent applications from diverging from the underlying network state, *NEAt* exposes two integration modes: *pass-through* and *interactive*.

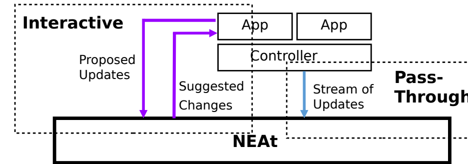


Figure 2: *Interaction modes of NEAt.*

In pass-through mode, *NEAt* acts as a transparent layer that sits between the controller and forwarding devices. This mode enforces network policies without modification to controller or SDN applications. Both these applications and the controller are unaware of *NEAt*. *NEAt* intercepts updates from the controller, as well as updates from the network about link and switch state, and passes them to the verification and correction engines. The corrected updates abide by correct network policies, and are directly applied to the network.

It's possible for applications to have a different view with *NEAt* about the current network state in pass-through mode. But this does not violate consistency, since *NEAt* acts as an arbitrator during rule insertion, and will diligently verify and correct updates regardless of the application's intention. The original intention of an application is well preserved if the application is written with full knowledge of, and in accordance with, the network-wide correctness criteria. Otherwise, *NEAt* may sacrifice application correctness for the benefit of enforcing correct network policies.

Interactive mode enables applications to leverage *NEAt*'s verification and repair process by checking proposed updates. An application passes to *NEAt* a set of updates, which are checked against the current network model. If the updates introduce a violation, *NEAt* returns a set of repaired updates, which the application can accept or reject. If the application accepts the changes, it can send them onto the network and update its state, ensuring the application and network state are consistent. If the application rejects the changes, it can propose another set of updates to *NEAt*. Interactive mode requires modifications to applications to update its state with the accepted changes.

NEAt maintains consistency between the interaction modes, allowing applications and the controller to both simultaneously benefit from *NEAt*'s automated repair. For example, one application can use *NEAt*'s API while another remains unmodified, allowing its updates to be checked by *NEAt* in pass-through mode.

4 Policy as Graphs

Many existing tools reason about individual network paths [18, 19]. While this approach has proven effective for network verification, synthesizing network state changes requires viewing the entire network as a whole

(i.e., a graph), as changes that repair one path may influence the correctness of other paths. In addition, expressing network correctness conditions as a graph instead of a collection of paths enables dealing with a richer set of policies, for instance, path consistency and load balancing. Based on this intuition, *NEAt* takes as input a set of intended policies, and formulates these policies as directed graphs called *policy graphs*.

A *policy graph* is defined on a packet header pattern, for example, ip dst 10.0.1.0/24, port 443. Each node on a policy graph is a traffic footprint matching a particular packet header pattern at a certain network location, e.g., a switch, a routing table, an ACL table. Edges are marked with labels denoting different types of reachability constraints. For example, the graph in Figure 3 requires that at least m paths exist from node A to B when $m > 0$, each bounded by n hops, or no path exists from A to B when $m = 0$. For simplicity, packet header patterns are not depicted. Next, we show how to represent several commonly-used network policies as policy graphs.

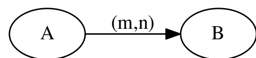


Figure 3: *Policy edge*

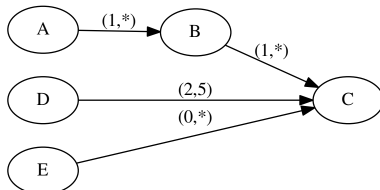


Figure 4: *Policy graph*

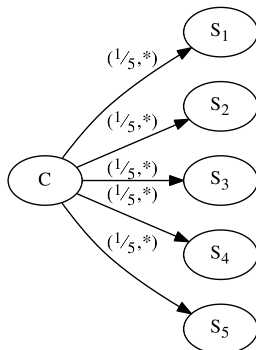


Figure 5: *Load balancing policy*

Reachability This policy requires that at least one path exists from one node to another. It is expressed with $m = 1$ and n unspecified (shown as "*"), for example, the edges $A \to B$ and $B \to C$ in Figure 4.

Bounded path length This policy defines the maximum number of hops between two reachable nodes. The path length is specified using n. For instance, the path from D to C in Figure 4 is bounded by 5 hops.

Shortest path This policy can be viewed as a special case of the bounded path length policy, where each path is bounded by the length of the corresponding shortest path in the topology. Therefore, it can be encoded in a similar way as bounded path length policy.

Multipath This policy requires multiple paths exist from one node to another. It is expressed by assigning m an integer larger than one. As shown in Figure 4, there should be at least two paths from D to C ($m = 2$).

Isolation This policy prevents one node from reaching another, which is expressed by specifying $m = 0$ on the edge connecting the two isolated nodes.

Service chaining The policy defines a set of *waypoints* that one flow must traverse in sequence. It is represented by concatenating edges on a policy graph. For example, in Figure 4, traffic from node A should traverse a waypoint B before reaching C.

Load balancing This policy requires distributing traffic from a source to a pool of servers according to a specified distribution. In our policy model, it is expressed by assigning m a fractional value . As an example, Figure 5 denotes a policy that requires traffic from client C to be distributed evenly among five servers.

In summary, a policy graph is able to express both qualitative and quantitative reachability constraints.

5 Repair Algorithm

In this section, we present *NEAt*'s core algorithm for repairing violations at runtime constrained by a given policy graph. First, we introduce the network model and give an overview of the algorithm. Next, we describe our formulation of the repair problem for basic reachability policies as an integer linear programming (ILP) problem. We then generalize this approach to repair the wider range of policies discussed in 4.

Network Model As described in §3, upon intercepting an update, *NEAt* constructs a network graph model for each affected EC that captures the configure forwarding behavior for all packets within the EC. This directed *configuration graph* ℓ_c, along with a topology graph T and policy graph $\wp$ serve as inputs to the repair algorithm.

Each node in these graphs represents a host or a networking device, and each edge between a pair of nodes defines reachability between them. The policy graph $\wp$, as discussed in 4, is a directed graph constructed from a set of conflict-free policies that represents the expected behavior of the whole network and hence should not be violated at runtime. Policy conflict freedom can be guaranteed by tools like PGA [23], which is out of the scope of this paper. A topology graph T is an undirected graph that represents the physical topology of the network.

Algorithm Overview When the verification engine discovers a violated EC, the algorithm is executed. Its goal is to repair the detected violations optimally, i.e., with the minimum number of changes to the original configuration. *NEAt* formulates the problem as an optimization problem: we aim to add or delete the minimum number of edges on ℓ_c so that the modified ℓ_c complies with $\wp_c$. $\wp_c$ is a subgraph of $\wp$ that is relevant to EC c. Note that the added edges are constrained within the topology graph T. We solve the optimization problem using ILP.

Subsection §5.1 describes the repair algorithm for basic reachability policies, and subsection §5.2 enhances the basic algorithm to cope with the entire set of policies in §4. We complete the section with our repair algorithm for forwarding loops (§5.3). Table 1 summarizes the key notations used in this section and the next section §6.

Symbol	*Description*
ℓ_c	The configuration graph for EC c.
$\wp$	The policy graph.
T	The topology graph.
(i,j)	The edge from node i to node j.
ρ_{ij}	The paths between node i and node j.
C_i^c	The cluster of node i for equivalence class c.
c_i	The compressed node i for C_i^c.
E_a	The set of all edges in graph a.
$N(E_a)$	Number of all edges in graph a.
$NB_a(i)$	The set of neighbors of node i in graph a.

Table 1: Key notations in problem formulation.

5.1 Repair Basic Reachability

We start with the basic case where $\wp_c$ contains only reachability constraints. Our integer program has a set of binary decision variables $x_{i,j,p,q}$ and $x_{i,j}$ where

$$x_{i,j,p,q}, (i,j) \in E_T, (p,q) \in E_{\wp_c} \tag{1}$$

$$x_{i,j}, (i,j) \in E_T \tag{2}$$

E_T and $E_{\wp_c}$ denote the set of all edges in T and $\wp_c$ respectively. Variable $x_{i,j,p,q}$ defines the mapping between a physical edge and a policy graph edge. It is one if a directed edge (i,j) is mapped to policy edge (p,q) for the current EC c, i.e., the flow from p to q will be forwarded through edge (i,j) from i to j. Variable $x_{i,j}$ defines whether edge (i,j) is used for forwarding this EC's traffic regardless of which flow uses it. Edge (i,j) in T is selected if any flow (p,q) is forwarded through (i,j) (Equation 3). Similarly, for the other direction (j,i), we have Equation 4. No physical link can be selected to forward traffic for the same EC on both directions (Equation 5) to avoid loops.

$$\forall(i,j) \qquad x_{i,j} \geq \sum_{(p,q)\in E_{\wp_c}} \frac{x_{i,j,p,q}}{N(E_{\wp})} \tag{3}$$

$$\forall(j,i) \qquad x_{j,i} \geq \sum_{(p,q)\in E_{\wp_c}} \frac{x_{j,i,p,q}}{N(E_{\wp})} \tag{4}$$

$$\forall(j,i) \qquad x_{i,j} + x_{j,i} \leq 1 \tag{5}$$

Equations 6-8 are the flow conservation equations for policy level reachability (p,q). $\forall(p,q), \forall i \in T$:

$$\begin{cases} \sum_{j\in NB_T(i)} x_{i,j,p,q} = 1 \\ \sum_{j\in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \tag{6}$$

$$\begin{cases} \sum_{j\in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j\in NB_T(i)} x_{j,i,p,q} = 1 \end{cases} \quad \text{if } i = q \tag{7}$$

$$\left\{ \sum_{j\in NB_T(i)} (x_{i,j,p,q} - x_{j,i,p,q}) = 0 \right. \quad \text{otherwise} \tag{8}$$

The optimization objective is to minimize the number of changes (additions and deletions) on the original configuration graph ℓ_c.

$$\min \Big(\sum_{(i,j)\notin E_{\ell_c}} x_{i,j} - \sum_{(i,j)\in E_{\ell_c}} x_{i,j} \Big) \tag{9}$$

5.2 Generalizing the Algorithm

To support generalized reachability policies in §4, we encode several additional constraints into the ILP.

Isolation We introduce a special *DROP* node. If two nodes are required to be isolated, i.e., the nodes are connected with a $(0,*)$ edge in the policy graph, we change the way flow conservation equations are defined. In particular, we replace Equation 7 with Equations 10 and 11 below in the flow conservation equations. That is, a flow from p to q should sink at *DROP* before reaching q.

$$\begin{cases} \sum_{j\in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j\in NB_T(i)} x_{j,i,p,q} = 1 \end{cases} \quad \text{if } i = DROP \tag{10}$$

$$\left\{ \sum_{j\in NB_T(i)} x_{i,j,p,q} = 0 \right. \quad \text{if } i = q \tag{11}$$

Service Chaining With service chaining, or waypointing, we enhance our flow conservation equations with Equation 12. It extends the definition beyond individual reachability segments (policy graph edges), by taking into account dependencies between policy edges. The resulting mapping is guaranteed to satisfy chaining of reachability requirements. For instance, if a policy node i is required to reach q through p, because of this equation, node i in the configuration graph is not allowed to carry flow from p to q. Without this equation, p might be skipped on the path from i to q.

$$\begin{cases} \sum_{j\in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j\in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i \in \wp_c \text{ and } (\exists\rho_{i,p} \text{or } \exists\rho_{q,i}) \tag{12}$$

Bounded or Equal Path Length/Shortest Path If a path length bound n is specified for a policy edge (p,q), then a new constraint is added (Equation 13):

$$\sum_{(i,j)\in E_T} (x_{i,j,p,q} + x_{j,i,p,q}) \leq n \tag{13}$$

Multipath If at least m link-disjoint paths are required for flow (p,q), then the flow conservation equations 6 and 7 are updated as Equation 14 and 15 respectively. Multipath requirements are enforced throughout the distance between two end nodes by Equation 8.

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} \geq m \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \qquad (14)$$

$$\begin{cases} \sum_{j \in NB_T(i)} x_{i,j,p,q} = 0 \\ \sum_{j \in NB_T(i)} x_{j,i,p,q} \geq m \end{cases} \quad \text{if } i = q \qquad (15)$$

Load Balancing As discussed in §4, policy edges within a load balancing policy are denoted with a decimal path count. Correspondingly, in our optimization problem, variables that map physical edges to policy edges are also decimal values between zero and one, instead of binary values. In addition, we introduce a new equation (Equation 16) to capture how flow distribution propagates.

$$\prod_{x_{i,j,p,q} \neq 0} x_{i,j,p,q} = m \qquad (16)$$

For example, consider the network in Figure 6, where there are two layers of load balancing between client C and servers $S1$ $S5$. If the policy in Figure 5 is required, the solutions for variables $(x_{i,j})$ are shown in Figure 6.

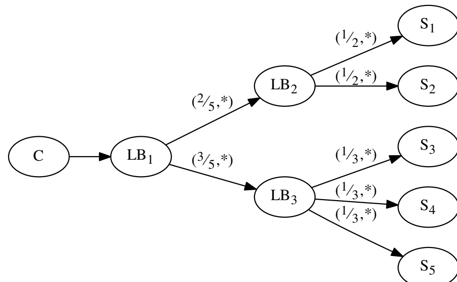


Figure 6: *Load Balancing Configuration.*

5.3 Repairing Loops

The preceding repair algorithm operates on a *loop-free* configuration graph. As such, we first check for and remove loops from each configuration graph before compressing and repairing violations of any other property type. Our objective for repairing loops is to minimize change to the network, with a preference to affect few equivalence classes as possible, as well as removed the minimal number of rules. Thus, our algorithm will remove a forwarding rule matching packets destined to 10.0.0.1/32 over one for 10.0.0.0/8. Since loops are repaired first, and *NEAt* will later check reachability properties on each equivalence class, our loop repair algorithm does not need to consider introducing permanent reachability violations by removing rules.

Algorithm 1 presents our loop repair algorithm. $\Theta(c)$ denotes the set of all loops appearing in a configuration graph ℓ_c and $N(\Theta(c))$ the number of loops in ℓ_c. θ_i is a subgraph of ℓ_c, and $N(\theta_i) = 0$ when the subgraph contains no loops. The algorithm begins by finding and removing all intersecting edges across ℓ_c's loops. For each loop in ℓ_c that is not repaired by removing these edges, next remove an edge (i,j) where i's IP address is the destination, if such an edge exists. While θ_i still has loops, remove an edge in the loop which has the most specific match rule (e.g., longest prefix). Each edge is mapped to a specific forwarding rule at a particular switch when we compute the equivalence classes.

Algorithm 1 Loop repair

```
procedure REMOVELOOP(ℓc, Θ(c))
    # remove edges appearing in multiple loops
    remove {(i, j) | (i, j) ∈ θk ∧ (i, j) ∈ θm ∀k,m ∈ Θ(c)}
    if N(Θ(c)) = 0 then
        return ℓc
    for all θi ∈ Θ(c) do
        while N(θi) > 0 do
            # remove edges forwarded out the destination
            remove (i, j) if i is destination
        while N(θi) > 0 do
            # remove most specific forwarding rule
            remove (i, j) ∈ θi with longest prefix
    return ℓc
```

Removal of a forwarding rule is accomplished by replace it with a drop rule, to prevent a coarser match from introducing another loop. For example, if a rule matching destination IP 10.0.0.1/32 is simply deleted from a switch's forwarding table, another rule matching 10.0.0.1/31 on the same switch and forwarding to the same next hop could prevent the loop from being repaired. To conserve switch memory during in response repairs, *NEAt* checks all coarser drop rules to determine if multiple rules can be aggregated together.

6 Optimizations

While conceptually straightforward, the repair algorithm in section 5 does not scale to well. In the optimization problem formulation, the number of variables for one EC is approximately the product of the number of topology links and the number of policy graph edges, which can easily exceed 100k. In this section, we present two techniques that dramatically optimize the repair speed.

6.1 Topology Limitation

This technique aims to "slice" away irrelevant or redundant part of the network, and thus shrink the size of the optimization problem. After getting a configuration graph that violates some policies, before passing it to the optimizer, we first remove disconnected components on

the physical topology. Next, we localize the potential affected area on the topology. Fortunately, most modern networks are designed in a hierarchical structure. Examples include data centers arranged in a fattree topology, and enterprise networks divided into multiple sites joint by a backbone network. Such a structure implies certain communication pattern: communication within a subtree should stay local, for example, and communication between subtrees normally doesn't traverse other subtrees, i.e., go through a valley. In our linear programming problem, typically only a subset of the topology edges is considered mappable to a policy edge. Results in section 8 shows the effectiveness of this technique.

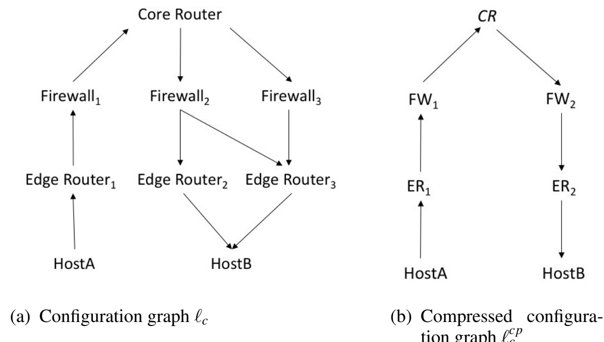
6.2 Graph Compression

Besides hierarchical structures, most large networks are designed in patterns that enforce symmetry to some extent [22] for load balancing or resilience reasons. For example, in a data center fattree topology, devices on the same layer (access, aggregate, core) are symmetrically connected to multiple devices on the neighboring layers. We exploit such regularities to compress the graphs. The key to the compression is that the compressed graphs must be equivalent to the original graphs with respect to the policies of interest. To this end, we leverage a graph pattern preserving compression [10] as the major building block of *NEAt*'s compressor (Figure 1). The algorithm compresses a labeled directed graph according to the following bisimulation relation:

Bisimulation Relation [9] We denote $G = (V, E, L)$ as a labeled directed graph. V represents a set of node and $(u, v) \in E$ represents a directed edge from node u to node v. $L(u) \in \Gamma$ represents the label of node u, where Γ is the set of labels that applied to V. In the networked system context, the labels may represent a set of similar functional networking nodes, e.g. hosts, firewalls, load balancers. For example, in Figure 7(a), we label the network nodes as *Firewall*, *Edge Router* and *Core Router* and we label the two hosts as $HostA$ and $HostB$.

A *bisimulation relation* on a graph $G = (V, E, L)$ is a binary relation $BR \subseteq V \times V$ such that for all $(u, v) \in BR$:
(a) $L(u) = L(v)$;
(b) $\forall(u, u') \in E, \exists(v, v') \in E$ such that $(u', v') \in BR$;
(c) $\forall(v, v') \in E, \exists(u, u') \in E$ such that $(u', v') \in BR$.
In Figure 7(a), $Firewall_2$ and $Firewall_3$ are *bisimular* to each other, while $Firewall_1$ is not *bisimular* to any other firewall. Because $HostB$ is solely in a *bisimular* cluster, and hence $EdgeRouter_1$ and $EdgeRouter_2$ are *bisimular* as they only has one child $HostB$. As $Firewall_2$ and $Firewall_3$ have the children that are *bisimular*, they are also *bisimular* to each other. While $Firewall_1$'s child is *Core Router*, which has a different label than *Edge Router*, $Firewall_1$ is not *bisimular* to anyone.

Bisimulation Based Compression



(a) Configuration graph ℓ_c (b) Compressed configuration graph ℓ_c^{cp}

Figure 7: *Example of compression*

Algorithm 2 presents the compression algorithm on the given graphs ℓ_c, $\wp_c$ and T, where ℓ_c and T are compressed according to $\wp_c$. Before the compression, we need to first label the nodes in ℓ_c and T according to $\wp_c$: all the nodes that are presented in $\wp_c$ are labeled uniquely. Therefore, the information in the policy graphs will not be lost after compression. We then compute *bisimulation relation* on ℓ_c using the algorithms presented in [9] and then compress the graphs based on the bisimularity. However, unlike ℓ_c and $\wp_c$, T is not a directed graph, and thus the original algorithm is not applicable. To compute T^{cp}, we first compress the parts in T that overlap with ℓ_c according to the undirected version of ℓ_c^{cp}. Then we draw edges between the non-overlapping parts and the compressed parts with their original edges in T. The time complexity of the compression algorithm is $O(|E|log|V|)$. Figure 7(b) shows the compression result on graph ℓ_c. $Firewall_2$ and $Firewall_3$ are *bisimular* and are compressed to a new clustering named FW_2. $Firewall_1$ stays by itself as FW_1.

Algorithm 2 Graph pattern preserving compression

```
procedure GRAPHCOMPRESSION(ℓc, ℘c, T)
    compute the maximum bisimulation relation BR of ℓc
    compute the clusters clusters = V/BR
    collapse the nodes in the each cluster ∈ clusters
    compute compressed ℓc^cp, T^cp
    return ℓc^cp, T^cp
```

We evaluate the compression algorithm on a simulated fattree topology and a large enterprise network. We denotes the compression rate r_c as the ratio of the number of the remaining nodes in ℓ_c^{cp} to the number of the nodes in ℓ_c. From the compression result shown in Table 2, we can conclude that the compression algorithm could result in a much smaller graph for a large-scale network.

Topology	$1 - r_c$
Fattree (6750 hosts, 1125 switches)	99.38%
Enterprise (236 routers)	88.98%

Table 2: Compression results.

Incremental Compression Further leveraging the incremental compression algorithm from [10], we incrementally maintain the compressed configuration graphs. In response to changes to the original graphs, the incremental algorithm computes the new compressed graph using the changes and the compressed graph as input, independent of the original graph. That is there is no need to decompress the graph to propagate the changes.

Repair Compressed Graphs With the compression module in place, when a violation is detected, the graphs are compressed first, then passed to the optimizer. Note that one compressed edge may represent a collection of edges in the original graph. This works fine with single-path reachability type of policies, such as reachability, isolation, service chaining. However, it will break Equation 14 and 15 for link-disjoint multipath policy. Our solution is to label the predecessors of each multipath policy destination node (E.g., q for policy edge (p,q)) differently, such that they are not compressed. In addition, T^{cp} is modeled as a weighted graph, where the weight on each edge is the number of original edges that the compressed edge represents. Multipath policy constraint Equation 14 is modified as shown Equation 17, while Equation 15 remains the same because there are never multiple edges pointing to the destination node q.

$$\begin{cases} \sum_{j \in NB_{T^{cp}}(i)} (x_{i,j,p,q} * weight_{i,j}) >= m \\ \sum_{j \in NB_{T^{cp}}(i)} x_{j,i,p,q} = 0 \end{cases} \quad \text{if } i = p \tag{17}$$

Map Back The last step is to map the result back to the original graph ℓ_c. The optimization result is a set of changes (added or deleted edges) on the compressed graph ℓ_c^{cp}. To map back to ℓ_c, a changed edge (c_i, c_j) could become a set of changed edges between the cluster C_i^c and cluster C_j^c. If an edge (c_i, c_j) is supposed to be added to ℓ_c^{cp}, then on ℓ_c, for every node i in the source cluster C_i^c, there should be an edge added from i to *one* of its neighbor node j that is in the target cluster C_j^c. It does not matter which neighbor node is chosen, because all the nodes in C_j^c are equivalent with regard to the policies, which is why they are clustered as one node. In the current design, every policy node represents a physical node, and thus a policy edge represents a one-to-one connection. However, in the future, we plan to also compress the policy graphs, enabling a policy graph node representing a cluster of nodes with similar functions. This enables policy graph edges to denote various types of connection, for example, any-to-any, one-to-many. Afterwards, those computed changes will be translated into forwarding instructions, and sent to the network devices.

Policy Perseverance Finally, we prove that the Graph-Compression algorithm (Algorithm 2) preserves the equivalence between the compressed graph G_c and the original graph G with respect to the scope of policies in section 4. As loops are repaired before the graph is compressed (§5.3), the input (G) and output (G_C) of the compression algorithm are equivalent with respect to the *loop* policy. Furthermore, on a loop-free graph, the compressed graph is proven in [10] to be equivalent to the original graph for graph pattern queries. Therefore, single-path reachability policies with bounded length and waypointing constraints are equivalent on both graphs. More specifically, let us denote $Q_r(v,u)$ as the reachability query between v and u. Intuitively, for each $Q_r(v,u)$ for G, one can show by contradiction that there exists a path from v to u in G if and only if c_v can reach c_u. Similarly, the isolation policy is preserved, as ρ_{c_v,c_u} exists iff $\rho_{v,u}$ exists. Further, as reachability is preserved, for each edge $(m,n) \in G$, there exists an edge $(c_m, c_n) \in G_c$, i.e., the length of any path is the same on G and G_c.

The load balancing and multi-path policies are more complex. We can break the load balancing policy into two requirements: *pool* and *balancing*. *Pool* in this context denotes that traffic from a single source is distributed to all of a fixed pool of nodes, which is naturally preserved as a reachability requirement. *Balancing* denotes the amount of traffic distributed to each node in the pool is equal to the amount specified by the operator. Balancing is enforced by Equation 16. Since paths are not shortened by the compression algorithm, if Equation 16 holds on G_c, it also holds on G.

We prove that this conclusion also holds for multipath policy in Appendix A. Intuitively, as the predecessors of multipath destinations in are not compressed, the link-disjoint multipath criteria is preserved after compression through the bisimulation relation back propagation and flow conservation constraints.

7 Implementation

We implemented a prototype of *NEAt* in Python. *NEAt* requires no modifications to the controller or switches. The verification engine is based on prior work [19] and we use the Gurobi Optimizer [3] within our optimization engine to solve the ILP.

NEAt's pass-through mode is implemented as a proxy between the controller and switches, listening for flow modification messages. The interactive mode is implemented as an XML-RPC API, allowing it to be compatible with applications written in any language or for any controller. In particular, *NEAt* exposes a `check()` function that accepts a set of OpenFlow flow modification messages to check against the network policy. *NEAt* updates the network model with the proposed changes, verifies the model, and searches for a set of repairs if any violations are found. The application can choose to receive the repairs as a set of OpenFlow flow modification messages or as a set of edge tuples. For example, a load

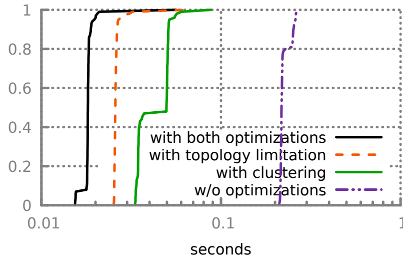


Figure 8: *Effect of optimizations*

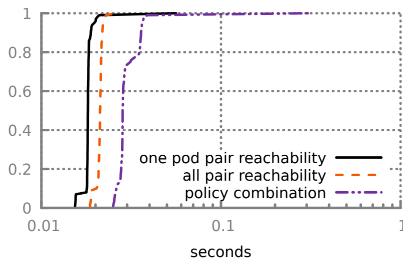


Figure 10: *Different policies.*

balancer application may wish to receive a repairs as a set of tuples (e.g., *[(s2, h1)]*) to easily re-assign a client to a particular server replica, rather than parsing an OpenFlow message from *NEAt*.

8 Evaluation

In this section, we examine the performance of repairs in *NEAt*, as well as the end-to-end latency experienced by applications. All experiments were run on a Dell Precision 5810 with a 2.6GHz Xeon E5-2697V3 CPU and 128GB RAM. We use an unmodified version of Gurobi [3] with default options in our experiments.

8.1 Repair Performance

To evaluate the feasibility and scalability of *NEAt*'s repair process, we synthesized a set of fattree topologies with various sizes, and used *NEAt* to maintain a variety of network-wide policies, including reachability, segmentation, bounded path length and multipath policies. More specifically, on each topology, under random removals of rules, we measured the repair time for each removal that caused a violation.

8.1.1 Exact matching rules

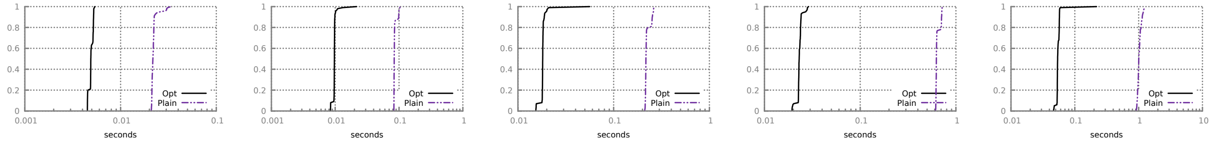
We first focus on flow-based traffic management applications, which are widely used in SDN [7, 12, 14–16]. Any forwarding rule produced by such applications at a switch matches at most one flow. In our terms, each rule only affects at most one EC.

For each fattree topology, we randomly selected a pair of pods. Suppose the desired policy is that any host in one of the pods should be able to reach every other host in both selected pods, which we will refer to as a *pod-pair reachability* policy. With random removals of rules, for those removals resulting in violations to *pod-pair reachability*, the optimization engine is triggered to perform the repair. For testing purposes, we re-verify the policy after each repair, and the check passed for all cases.

On a fattree topology with 250 hosts and 125 switches, we measured the time taken to repair *pod-pair reachability* policy by four mechanisms: (1) plain mapping, (2) mapping with topology limitation, (3) mapping with graph compression, and (4) mapping with both compression and topology limitation. Figure 8 compares the CDFs of the repair time for these four repair mechanisms: We can see the combination of graph compression and topology limitation (left most curve) brings approximately one order of magnitude speed-up over plain mapping (right most curve). Figure 9 (a-e) shows the amount of speed-up goes up as the network size scales. Even on a network with 686 hosts and 245 switches, the repair time is bounded under 0.1 second for the majority case, close to 1/20 of the repair time by plain mapping.

We next explored how *NEAt* handles a larger set of policies and a combination of different types of policies. We first assumed the desired policy being every pair of hosts should be able to reach each other , which we will refer to as an *all-pair reachability* policy. Again, on a fattree topology with 250 hosts and 125 switches, the repair time under random rule removals against this *all-pair reachability* policy was measured, as shown in Figure 10. The policy size is increased by approximately 10 times compared with *pod-pair reachability* policy, but the repair time only increases slightly.

To test a even more complex setting, next we randomly selected three pods in the fattree. Between the first two pods, hosts should be isolated from each other (*segmentation*), and between the first and third selected pods, hosts are connected by at least two path (*multipath*). For host pairs that do not fall into the previous two conditions, they are supposed to be able to reach each other (*all-pair reachability*). Both *multipath* and *all-pair reachability* are combined with a bounded path length policy, to avoid flows between pods ”go through a valley”. Note that unlike the previous pure single-path reachability policy, where repairs are all edge additions, in this case, a repair is sometimes a mix of edge additions and deletions. What's more, to satisfy multipath requirement, more additions are necessary. Due to this complexity, the repair time is increased, but still on the same order of magnitude of reachability policy cases, as shown in Figure 10. As verified by the re-checks, changes for fixing different types of policies keep other policy intact.



(a) 54 hosts, 45 switches (b) 128 hosts, 80 switches (c) 250 hosts, 125 switches (d) 432 hosts, 180 switches (e) 686 hosts, 245 switches

Figure 9: *Repair time comparison under random removals of exactly matching rules*

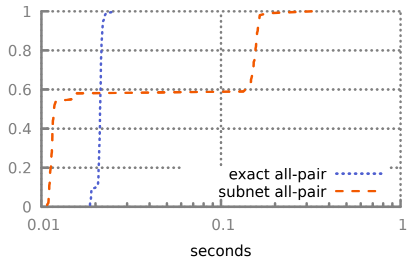


Figure 11: *Exact matching rules vs. overlapping rules*

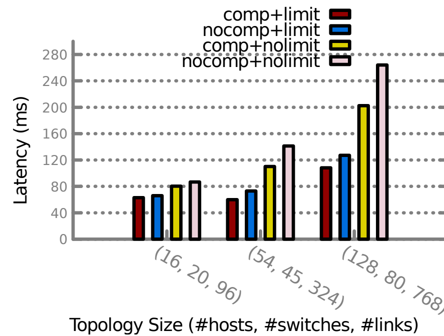


Figure 12: *Application-perceived latency of NEAt, on various fattree topologies, showing performance for a reachability policy with/without graph compression and topology limitation*

8.1.2 Overlapping rules

For networks that use wild-carded rules or longest prefix matching, the assumption in the previous subsection does not hold. One rule may affect multiple ECs, and thus potentially trigger repairs on multiple graphs. Fortunately, there is a trend to move such overlapping rules to network edge or even hosts [5, 8, 20], leaving the core with exactly matching rules. In order to study how *NEAt* performs under this less preferable but less common scenario, we assign IP addresses within the same prefix subnet to hosts within the same pod on the fattree topologies. We then aggregated rules on the switches as much as possible. For example, each core switch has only k forwarding rules, where k is the number of pods, and each rule matches on one pod's prefix. Similar to the previous experiments, we used *NEAt* to guarantee an *all-pair reachability* policy, and our engine discovered repairs for all violations. Figure 11 compares the CDFs of the repair time for overlapping rules and exact matching rules on a 250-host-125-switch fattree topology. The repair took longer compared to applications with exact match rules because of the increased number of affected ECs. With our graph compression and topology limitation techniques, optimization is able to finish under 0.4 seconds in the worst case.

8.2 End-to-End Delay

Next, we examine the application-level delay introduced by *NEAt* when using its interactive mode. We test *NEAt* on various-sized fattree topologies using Mininet [4] and the Pox controller [6]. A learning switch application and load balancer application run on top of Pox. The load balancer balances flows between the two replicas in a round-robin fashion, and we modify it to leverage *NEAt*'s API to check the assignment of clients to replicas. If *NEAt* suggests a repair, the application updates its client-to-replica mapping with one suggested by *NEAt*. While the learning switch remains unmodified and unaware of *NEAt*, its updates are transparently checked by *NEAt*. This setup demonstrates the ability of *NEAt* to interact with the controller and applications simultaneously through its two interactive modes.

The load balancer application runs on an edge switch in the fattree topology, with clients and server replicas placed in different pods. To trigger an update, a client pings the virtual IP of the load balancer. When the appropriate event handler in the load balancer is execute, it invokes *NEAt*'s `check()` function. We measure the total latency introduced by *NEAt* as the time to invoke the `check()` function and apply it to the application's state. This includes the time to verify an update (i.e, calculate equivalence classes affected by the update, compute their configuration graphs, and verify them) and repair violations in any of the affected equivalence classes.

For each topology size, we examine the total latency for a reachability policy, with and without our compression and topology limitation optimizations. Figure 12 shows the total delay experienced by the load balancer. Topology limitation has the largest speed-up of our optimizations, but when used in combination with compression of the topology and configuration graphs, *NEAt* can verify and repair an update in under 120ms.

8.3 Enterprise Network Trace Study

Finally, we examine traces from a large enterprise network, to examine *NEAt*'s performance on real forward-

ing graphs. We examine two dumps of the data plane from 2014 and 2017. These datasets containing more than one million forwarding rules across more than 200 forwarding devices. The 2014 dataset contains 27k equivalence classes, while the 2017 trace contains 285k.

8.3.1 Bugs

For each dataset, we construct loop and reachability policies and check for violations. In the 2014 dataset, *NEAt* finds nine different loops. In the 2017 dataset, *NEAt* finds 19. We examine the forwarding table and find several of these are caused by default routes with prefix 0.0.0.0/0. Only equivalence classes with more specific rules on the device are free of loops in these cases. Another cause we discover is load balancing — a device can forward packets out one of two ports, one of which will result in a path containing a loop.

8.3.2 Synthetic Updates

Next, we use the 2017 dataset to evaluate *NEAt*'s on a data plane with a realistic number of equivalence classes. First, we repair any loops in the dataset's 285k equivalence classes. We then construct synthetic updates, choosing a destination IP address and prefix length with the same probability as they appear in the dataset's forwarding rules. An update can add a rule, delete a rule, or introduce a loop. Loops are chosen from the list of those that were discovered and repaired in the first step. An update has a 10% chance of introducing one of these loops for a particular update, which may introduce loops in multiple ECs. We generated 100 updates in this manner, which affected an average of eight ECs per update.

We apply the set of random updates to different combinations of policies, including loop-freedom, reachability, and our compression and topology limitation optimizations. Since the compression and topology limitation optimizations only apply to the reachability policy, we do not test loop freedom with compression or topology limitation. Figure 13 shows a CDF of the total update time, including verification and repairs (when necessary). Of the 100 updates, 20 loops violations needed repair, as well as 24 reachability violations. Median and 98th percentile update times were 10ms and 1300ms, respectively, for a reachability policy with compression and topology limitation enabled. For a loop freedom property, median and 98th percentile update times were 35ms and 730ms, respectively. Combining these two policies, without compression or topology limitation optimizations, resulted in median and 98th percentile times of 36ms and 193 seconds. Adding our two optimizations reduced these times to 36ms and six seconds.

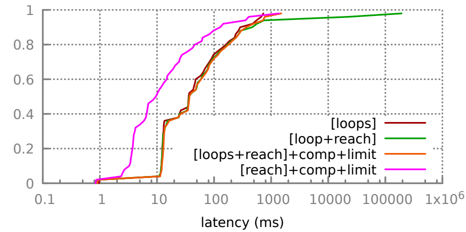


Figure 13: *Total update time for different policies and optimizations, on a model of a real-world data plane trace*

Topology Size	*NEAt*	NetGen	NetGen-C
(16, 20, 96)	5.9ms	743.2ms	513.2ms
(54, 45, 324)	7.2ms	4404.0ms	1160.8ms
(128, 80, 768)	9.0ms	16337.7ms	2056.3ms

Table 3: *Repair time of an all-pair reachability property in NEAt, NetGen [26], and NetGen using our clustering algorithm (NetGen-C) on fattree topologies. Topology size is measured as (#hosts, #switches, #edges).*

8.4 Repair vs. Synthesis

In the context of this work, we distinguish between repair and synthesis based on the scope and urgency of the change. We envision repair as a tool for temporary and immediate application on a time scale too small for human intervention. For example, the scope of *NEAt*'s repair is limited to forwarding actions after a single update to the data plane. We consider synthesis, on the other hand, to be useful for construction of longer-lived configurations and programs, without the need for a partial implementation, or changes of a larger scope.

In this section, we compare *NEAt* with NetGen [26], a tool to synthesize data plane changes using an SMT formulation. NetGen's specification language uses regular expressions to both select candidate ECs and describe changes to paths within them. In Table 3, we compare *NEAt* and NetGen under a repair scenario similar to § 8.1.1. We remove a single forwarding rule from the data plane on various-sized fattree topologies to introduce a violation, and measure the median repair time over 10 trials. For *NEAt*, we use both graph compression and topology limitation. For NetGen, we report results for the original approach operating on uncompressed configuration graphs, as well as a modified version that leverages our graph compression. We can see *NEAt* achieves performance up to two orders of magnitude faster than the modified version of NetGen, and up to four orders of magnitude faster than the unmodified version of NetGen.

In Table 4, we evaluate *NEAt* and NetGen under a synthesis scenario and generate an entire data plane from scratch. Specifically, we use *NEAt* and NetGen to "repair" an empty configuration graph and report the total time to repair an all-pair reachability policy. Since

Topology Size	*NEAt*	NetGen
(16, 20, 96)	921.7ms	7.1min
(54, 45, 324)	16.3ms	381.7min
(128, 80, 768)	2.9min	173.2hrs

Table 4: *Synthesis time of an all-pair reachability property on an empty configuration graph using fattree topologies. Topology size is measured as (#hosts, #switches, #edges).*

NEAt's policies are graph-based, we express the reachability policy with a single policy graph. As NetGen's specification is path-based, we encode the policy as a separate data plane change (i.e., regular expression) for each pair of nodes. For *NEAt*, we report the repair time with our topology limitation optimization. We do not report results with our clustering algorithm, as it is not applicable when the configuration graph is empty. We can see *NEAt* repairs an empty configuration graph more than 1000X faster than NetGen.

9 Related Work

SDN programming languages: Many programming languages have been proposed to provide abstractions to program SDNs, e.g., Frenetic [11], Pyretic [24] and Maple [29]. These allow programmers to compose complex rules without manually resolving conflicts between rules. However, these languages face limitations of expressing general policies that deliver higher-level intent, such as middleware functionality or QoS constraints.

SDN synthesis platforms: Network state can also be synthesized from a set of pre-specified correctness criteria. NetGen [26], for example, takes as input a specification using regular expressions to define paths changes and a set of ECs to modify. It uses an SMT solver to find the minimal number of changes. However, similar to Merlin [28] and FatTire [25], this tool is designed to be used as compiler, with performance that is too slow for real-time applications (i.e., minute-scale synthesis). Instead, NEAT formulates repairs as an ILP and discovers possible repairs in under a second. While using NetGen in place of our ILP is possible, certain policies cannot be expressed in NetGen's language, such as multipath and load balancing. Similarly, Marham [13] proposes a framework for automated repair, but with performance on the order of several seconds for topologies with dozens of nodes and links. Margrave [21] analyzes changes to access control policy changes, highlighting to an operator the effect it has on the policy, without suggesting repairs to violations.

10 Limitation and Discussion

Stateful Network Applications Some stateful network applications keep internal state to provide finer grained control of network traffic. The internal state is normally constructed based on the current network state. Under pass-through mode, as *NEAt* verifies and corrects rule insertions without notifying the application, network state might become different from the application's internal state. If improperly written, the application might crash. We note that this is true for many platforms which virtualize the network [27]. This might sound unsatisfactory yet it is likely desirable, since the application may be developed by an untrusted third party, and *NEAt* can protect the network from unforeseen bugs or undesirable behavior of that application. If desired, applications could be implemented with *NEAt* in mind. We encourage developers to use interactive mode for stateful applications.

Evolving Policies In practice, the policy graph can change over time, on human time scales as network operators revise and evolve earlier policy decisions. To simplify processing, *NEAt* can pause updates while the policy graph is updated. Since loading a new policy graph is a nearly-instantaneous process, this procedure introduces minimal delay in updates reaching the network. In future work, we plan to examine how *NEAt* handles such scenarios, and make design improvements if needed.

Different Optimization Goals In the current design of *NEAt*, the repair effort uses a minimal number of edits as the optimization goal. In practice, there may be other goals, for example, ensuring critical traffic free of congestion, minimizing the amount of traffic shifts, etc. We plan to extend the design in the future to optimize user defined utility functions, and study how accurate *NEAt*'s solution is under different scenarios, and under what types of scenarios *NEAt* is applicable.

11 Conclusion

In this paper we presented *NEAt*, a system that provides network administrators with a network analogue of a smartphone's autocorrect. As a transparent layer, *NEAt* repairs, in real-time, updates from an SDN controller that violate generic policies such as reachability, service-chaining, and segmentation. *NEAt* casts the repair process as an optimization problem, and repairs each update by adding or removing a minimal number of rules to satisfy the policy. Experiments on large fattree topologies show our formulation can discover repairs in under one second for applications with non-overlapping rules, and two seconds for applications issuing rules with more complex dependencies. Applying *NEAt* to a large enterprise network uncovered and repaired 28 loops

We thank NSF for supporting this work with grant CNS 15-13906, and our shepherd Cole Schlesinger and the anonymous reviewers for their valuable comments.

A Multipath Policy Perseverance

Theorem 1. *(Multipath Equivalence): A multipath policy for a flow* (p,q) *holds in* G iff *the policy also holds for* (p,q) *in* G_c.

Proof. Consider a multipath policy that requires at least m paths for flow (p,q). Trivially, if a flow (p,q) satisfies the policy in G, the policy also holds for (p,q) in G_c, and flow conservation equations (Equation 17, 15 and 8) are satisfied.

Next, we need to prove when the policy holds in G_c, i.e., when Equation 17, 15 and 8 are satisfied, it also holds in G.

Let $path_1^c$, $path_2^c$, ... $path_n^c$ ($n \leq m$) be the set of paths from p to q in G_c that collectively satisfy Equation 17, 15 and 8. That is, the sum of weights of all paths' starting edges is m. If n equals m, then there are at least m link-disjoint paths in G_c between p and q, and thus there are at least m link-disjoint paths in G, i.e., the policy is satisfied.

If n is less than m, then there must be at least one path in G_c, whose starting edge's weight is more than one. Let such paths be $path_{m0}^c$, ..., $path_{mj}^c$, whose starting weights are k_0, ..., k_j respectively. Consider path $path_{m0}^c$ first. The starting weight being more than one means that its starting edge is pointing from p to a cluster C_{next} which contains at least k_0 nodes which are also p's successors. Because the predecessors of q are labeled differently, each of them is a separate cluster. By the definition of bisimulation relation, two nodes are bisimular (and thus can be clustered together) only if their children's label set are the same. Via back propagation, and the constraint of Equation 8, there must be at least k_0 disjoint paths in G from p's successors in C_{next} to q's predecessors. When $path_{m0}^c$ is expanded in G, it becomes k_0 link-disjoint paths. Similarly, suppose we iterate through all the paths from p to q in G_c and expand each of them in G. As the sum of the starting weights is equal to m, there are at least m paths from p to q in G.

□

References

[1] http://www.infosecurity-magazine.com/opinions/to-err-is-human-to-automate-divine/.

[2] http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/.

[3] Gurobi optimization. `http://www.gurobi.com/`.

[4] Mininet: Rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[5] Network virtualization for cloud data centers. http://tinyurl.com/c9jbkuu.

[6] The pox controller. https://github.com/noxrepo/pox.

[7] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologiesperiments and Technologiesnference on emerging Networking EXperiments and Technologies (CoNEXT)* (2011), ACM, p. 8.

[8] B.RAGHAVAN, M.CASADO, T.KOPONEN, S.RATNASAMY, AND A.GHODSI, A. S. S. Software-defined Internet architecture: Decoupling architecture from infrastructure. In *HotNets* (2012).

[9] DOVIER, A., PIAZZA, C., AND POLICRITI, A. A fast bisimulation algorithm. In *CAV* (2001), vol. 2102, Springer, pp. 79–90.

[10] FAN, W., LI, J., WANG, X., AND WU, Y. Query preserving graph compression. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 157–168.

[11] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A network programming language. In *ICFP* (2011).

[12] HELLER, B., SEETHARAMAN, S., MAHADEVAN, P., YIAKOUMIS, Y., SHARMA, P., BANERJEE, S., AND MCKEOWN, N. ElasticTree: Saving energy in data center networks. In *NSDI* (2010).

[13] HOJJAT, H., REUMMER, P., MCCLURGH, J., CERNY, P., AND FOSTER, N. Optimizing horn solvers for network repair. In *FMCAD* (2016).

[14] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving high utilization with software-driven wan. In *SIGCOMM* (2013).

[15] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HOLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM* (2013).

[16] JIN, X., MAHAJAN, R., LIU, H. H., GANDHI, R., KANDULA, S., ZHANG, M., REXFORD, J., AND WATTENHOFER, R. Dynamic scheduling of network updates. In *SIGCOMM* (2014).

[17] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[18] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[19] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[20] M.CASADO, T.KOPONEN, S.SHENKER, AND A.TOOTOONCHIAN. Fabric: A retrospective on evolving sdn. In *HotSDN* (2012).

[21] NELSON, T., BARRATT, C., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. The margrave tool for firewall analysis. In *LISA* (2010).

[22] PLOTKIN, G. D., BJRNER, N., LOPES, N. P., RYBALCHENKO, A., AND VARGHESE, G. Scaling network verification using symmetry and surgery. In *POPL* (2016).

[23] PRAKASH, C., LEE, J., TURNER, Y., KANG, J.-M., AKELLA, A., BANERJEE, S., CLARK, C., MA, Y., SHARMA, P., AND ZHANG, Y. PGA: Using graphs to express and automatically reconcile network policies. In *SIGCOMM* (2015).

[24] REICH, J., MONSANTO, C., FOSTER, N., REXFORD, J., AND WALKER, D. Modular sdn programming with pyretic. In *USENIX ;login, 38(5)* (October 2013), pp. 40–47.

[25] REITBLATT, M., CANINI, M., GUHA, A., AND FOSTER, N. Fattire: Declarative fault tolerance for software-defined networks. In *HotSDN* (2013).

[26] SAHA, S., PRABHU, S., AND MADHUSUDAN, P. NetGen: Synthesizing data-plane configurations for network policies. In *SOSR* (2015).

[27] SHERWOOD, R., GIBB, G., YAP, K.-K., APPENZELLER, G., CASADO, M., MCKEOWN, N., AND PARULKAR, G. Can the production network be the testbed? In *OSDI* (2010).

[28] SOULE, R., BASU, S., MARANDI, P. J., PEDONE, F., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Merlin: A language for provisioning network resources. In *CoNEXT* (2014).

[29] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: Simplifying sdn programming using algorithmic policies. In *SIGCOMM* (2013).

[30] ZHOU, W., JIN, D., CROFT, J., CAESAR, M., AND GODFREY, P. B. Enforcing customizable consistency properties in software-defined networks. In *NSDI* (2015).

Net2Text: Query-Guided Summarization of Network Forwarding Behaviors

Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, Martin Vechev
ETH Zürich

net2text.ethz.ch

Abstract

Today network operators spend a significant amount of time struggling to understand how their network forwards traffic. Even simple questions such as *"How is my network handling Google traffic?"* often require operators to manually bridge large semantic gaps between low-level forwarding rules distributed across many routers and the corresponding high-level insights.

We introduce *Net2Text*, a system which assists network operators in reasoning about network-wide forwarding behaviors. Out of the raw forwarding state and a query expressed in natural language, *Net2Text* automatically produces succinct summaries, also in natural language, which efficiently capture network-wide semantics. Our key insight is to pose the problem of summarizing ("captioning") the network forwarding state as an optimization problem that aims to balance *coverage*, by describing as many paths as possible, and *explainability*, by maximizing the information provided. As this problem is NP-hard, we also propose an approximation algorithm which generates summaries based on a sample of the forwarding state, with marginal loss of quality.

We implemented *Net2Text* and demonstrated its practicality and scalability. We show that *Net2Text* generates high-quality interpretable summaries of the entire forwarding state of hundreds of routers with full routing tables, in few seconds only.

1 Introduction

Put yourself in the shoes of a network operator working for a large transit provider: you just received a call from one of the largest Content Delivery Networks (CDN) informing you that they observed bad performance for flows crossing your network. As a cautious operator, you run the latest control- and data-plane verification technologies and are confident that your network state is correct; you suspect a runtime problem. You start by collecting the CDN routing advertisements and identify a dozen of possible egress points used to reach them together with hundreds of ingresses. Analyzing some of the internal paths, you do not observe any signs of loss. Looking at traffic volumes, you realize that most of the CDN traffic leaves via one egress connected to an Internet Exchange Point (IXP). You suspect congestion inside the fabric (invisible to you). Indeed, lowering the preference for the CDN prefixes at the IXP solves the problem.

This example is loosely based on a real troubleshooting scenario observed at a Tier 1 and illustrates the challenges in understanding and reasoning about network-wide forwarding behavior. The main issue lies in the large semantic gaps that separate low-level forwarding rules distributed across the entire network and actionable high-level insights by network operators. Bridging this gap manually (the default nowadays) is slow. Reasoning about network behavior often takes hours (e.g., for the case above)—even for the most skilled network operator. As networks grow more complex (e.g. as the number of peering increases), so does the corresponding reasoning time. This tension is becoming even more palpable as networks carry more and more critical services.

The example also illustrates that human insights and domain-specific knowledge are *fundamental* for understanding non-trivial unwanted network behavior. Even if the network control- [1–4] and data-plane [5–10] are formally verified, subtle problems will arise at runtime. Here, no observable signs were available to the operator. The goal is therefore not to remove the human out of the loop, but instead to assist her.

Net2Text In this paper, we introduce *Net2Text*, an interactive system which assists the network operator in reasoning about network-wide forwarding state. Out of the low-level forwarding state and a query expressed in *natural language*, *Net2Text* automatically produces succinct, *natural language* descriptions, which efficiently capture network-wide semantics.

Relying on natural language ensures seamless human interactions. We think of *Net2Text* as a "chatbot" for networks. We confirmed the usefulness of such interfaces with the network operators themselves: all of them liked the idea of having natural language descriptions of their network. Of course, *Net2Text*'s reasoning capabilities could also be integrated with other high-level interfaces such as graphs [11].

Coming back to the example above, the operator could simply ask *Net2Text*: *"What happens to the traffic destined to CDN X?"* to which *Net2Text* would answer: *"Traffic enters via n ingresses and mostly (85%) leaves via IXP 1. Traffic is load-balanced between A and B."*. Using this high-level insight, the operator could then ask for more targeted information: *"Tell me more about the CDN traffic leaving via IXP 1"*.

The main challenge behind *Net2Text* is to generate concise summaries which "explain"' as much as possible out of the network forwarding state. We formulate this as an optimization problem that aims to balance *coverage* and *explainability* and show that it is *NP*-hard.

Fortunately, we show that the skewness of the network forwarding state (i.e. its inherent redundancy) makes it well-amenable to summarization in practice. This motivates us to focus on a subspace of solutions which we can prove contains good solutions. An important property of this subspace is that every search path is of polynomial size. This enables us to design an approximation algorithm that traverses the space efficiently.

We designed and implemented *Net2Text*. *Net2Text* takes a query in natural language, parses it to a database query, runs the query on a network database, summarizes the results, and translates the summarization back to natural language. The operator can then pose follow-up queries, and thereby interactively guide *Net2Text* towards producing summaries focusing on particular aspects of the network.

We evaluated *Net2Text* on a variety of realistic networks. Our results demonstrate that *Net2Text* is practical: it generates high-quality interpretable summaries of the entire forwarding state of hundreds of routers and full routing tables, in few seconds only.

Contributions Our main contributions are:

- A precise formulation of the network-wide summarization problem as an optimization problem (§4).
- An approximation algorithm for generating high-quality summaries (§5,§6), which scales to large data sets, and a translation of the abstract summaries to a description in natural language (§7).
- An implementation of *Net2Text*, along with a comprehensive evaluation. Experiments show that *Net2Text* can derive summaries for backbone networks with full routing tables within seconds (§9).

2 Overview

Consider an operator wondering how her network is forwarding traffic towards Google:

> *"How is Google traffic being handled?"*

Net2Text automatically parses the question expressed in natural language and produces a concise description (also in natural language) of the current forwarding behavior observed for Google:

> *"Google traffic experiences hot-potato routing. It exits in New York (60%) and Washington (40%). 66% of the traffic exiting in New York follows the shortest path and crosses Atlanta."*

Producing such a summary is challenging: the system has to understand what the operator is interested in, extract the relevant information, summarize it, and then translate it to natural language. Extracting this information goes beyond simply querying a database: it requires processing the data to identify common path features (e.g., the New York and Washington egresses) as well as high-level features pertaining to different paths (e.g., hot-potato routing, shared waypoints). In addition, the entire process should be quick (even if the network is large) to guarantee interactivity and deal with traffic dynamics.

In the following, we give a high-level overview of how *Net2Text* manages to solve these challenges and go from the above query to the final summary using a three-staged process (see Fig. 1).

Parsing operator queries in natural language (§8) *Net2Text* starts by parsing the operator query in natural language using a context-free grammar. This grammar defines a natural language fragment consisting of multiple network features (e.g., ingress, egress, organization, load-balancing) and possible feature values (e.g., New York, Google) allowing a network operator to express a wide range of queries. Our grammar consists of ~ 150 derivation rules which are extended with semantic inference rules to infer implicit information. In the above example, our grammar infers that the operator refers to traffic destined to the *organization* Google. *Net2Text* also understands other kinds of queries: *(i)* yes/no queries, "*Does* all traffic to New York go through Atlanta?"; *(ii)* counting queries, "*How many* egresses does traffic to Facebook have?"; and *(iii)* data retrieval queries, "*Where* does traffic to New York enter?". Our grammar is extendable with new features, keywords, and names.

Net2Text maps the parsed query to an internal query language, similar to SQL. Here, the query is mapped to: `SELECT * FROM paths WHERE org=GOOGLE`. This query is then run over a network database that stores the entire forwarding state of the network. Afterwards, the results are passed to the core part of *Net2Text*: the summarization module.

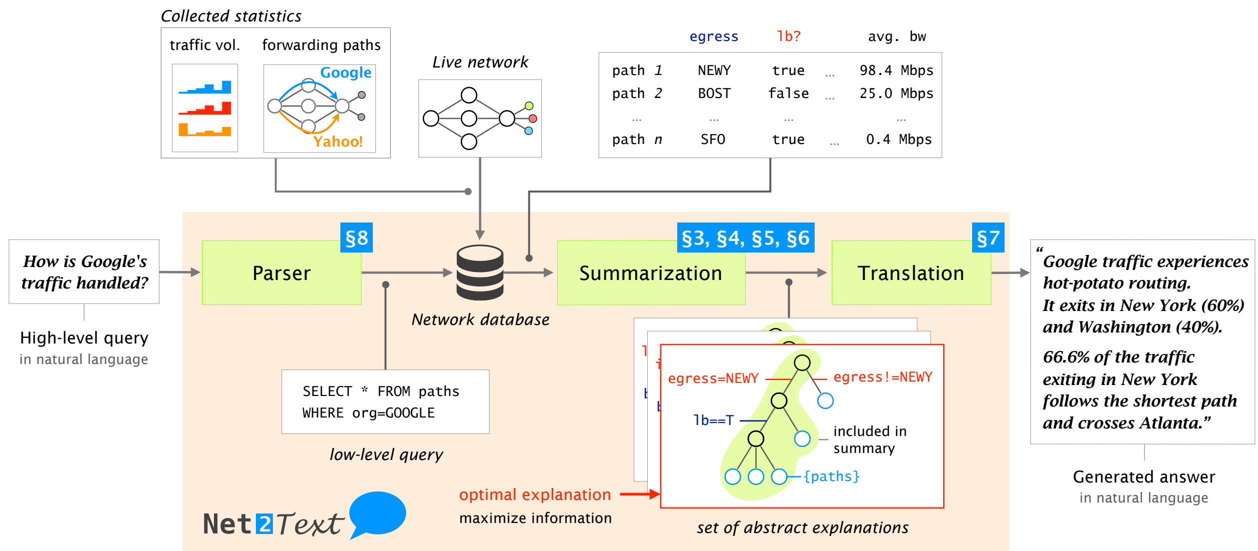


Figure 1: *Net2Text*: Workflow and key components.

Summarizing forwarding states (§4, §5, and §6) Most queries (including the one above) can and will return a plethora of low-level forwarding entries. *Net2Text* assists the operator in reasoning about forwarding state by automatically generating high-quality interpretable summaries out of low-level forwarding entries.

Summarizing network-wide forwarding states requires overcoming a fundamental tradeoff between *explainability* (how much detail a summary provides) and *coverage* (how many paths a summary describes). By defining a score function capturing both concepts analytically, we show that we can formally phrase this problem as an *NP*-hard optimization problem (§4). This renders both exhaustive techniques along with techniques based on Integer Linear Programming impractical.

To scale, we leverage the insight that traffic is skewed (heavy-tailed) across multiple levels: in the traffic distribution itself (few prefixes are typically responsible for most of the traffic [12]) or at the routing level (network topologies are usually built following guidelines, leading to repetitive forwarding patterns, e.g., edge/aggregation/core). This insight enables us to design an approximate summarization algorithm, called ComPass (§6), which explores a reduced search space that we can prove contains good summaries (§5). In addition, we show that ComPass can only summarize a sample of the forwarding entries instead of all of them with only a marginal loss in summarization quality.

Taken together, the reduced space and sampling optimization enable *Net2Text* to generate high-quality interpretable summaries for large networks (with hundreds of routers) running with full routing tables *in less than 2 seconds* (§9).

Converting path specifications to concrete text (§7) Given a set of path specifications, *Net2Text* finally produces a summary expressed in natural language in two steps. It first extends the set with additional properties inferred by examining the specifications as a whole. For example, if the specifications imply that there are multiple paths between the egress and ingress, *Net2Text* infers that the traffic is load balanced. *Net2Text* then maps the extended specifications to sentences in natural language.

3 Preliminaries

We now introduce the key terms we use in the paper.

Routing paths We model the network as a graph and define a network path P as a finite sequence of links. A routing path (d,P) is a pair of an IP prefix and a path, which describes that traffic to prefix d can be routed on P (a prefix can be routed on multiple paths). We denote by $\mathcal{R}$ the set of all routing paths in the network.

Feature functions Feature functions describe path features. Formally, a feature function $q\colon \mathcal{R} \to U_q$ maps routing paths $\mathcal{R}$ to *feature values* from U_q. We denote by v_q a value in U_q. We focus on the following feature functions. Organization $O\colon \mathcal{R} \to U_O$ maps every (d,P) to the organization owning d, e.g., Google. Egress $E\colon \mathcal{R} \to U_E$ maps every (d,P) to the egress of P, and ingress $I\colon \mathcal{R} \to U_I$ maps to P's ingress. Shortest path $SP\colon \mathcal{R} \to \{0,1\}$ maps to 1 if P is a shortest path between its ingress and egress, and 0 otherwise. We use the subscripts e, i, o, and sp to denote feature values of the egress, ingress, organization, and shortest path feature functions, e.g., New York$_e \in U_E$ and $1_{sp} \in U_{SP}$.

Path specifications To explain the behavior of the network and its routing paths, we define the concept of sets of feature values called *path specifications*. Given a set of l feature functions with disjoint ranges $U_1, ..., U_l$ [1] and a bound t (for $t \leq l$), a path specification is a (nonempty) set of feature values where the size of the set is at most t and each feature value describes a different feature function. Formally, a path specification is an element in:

$$\mathcal{S}^t_{U_1,...,U_l} = \bigcup_{1 \leq m \leq t} \bigcup_{1 \leq j_1 < ... < j_m \leq l} U_{j_1} \times ... \times U_{j_m}$$

Since the order of the feature values is not important for our needs, we treat path specifications as sets, e.g., $S_{G,NY} = \{\text{Google}_o, \text{New York}_e\}$.

We say a routing path (d,P) meets a path specification S, denoted $(d,P) \models S$, if for *every* feature value $v \in S$, if $v \in U_q$ for a feature function q, then $q(d,P) = v$. We define a specification set $\mathcal{S}$ as a set of path specifications, i.e., $\mathcal{S} \subseteq \mathcal{S}^t_{U_1,...,U_l}$. A routing path (d,P) meets a specification set $\mathcal{S}$, if there exists $S \in \mathcal{S}$ such that $(d,P) \models S$.

4 Problem Definition

Here, we formally phrase the problem of explaining network behaviors as an optimization problem.

Our goal is to find a summary of a (large) set of routing paths in the form of path specifications. The main challenge then is to infer a specification set that describes as many routing paths as possible while providing maximal amount of information about them. To evaluate the quality of a specification set, we define a score function. Intuitively, the score of a specification set is the sum of the "amount of explanation" of its routing paths. Given a score function, we formulate the problem of network summarization as constraint optimization.

We phrase our optimization problem as a MAP inference task [13], in which the goal is to find an assignment that maximizes a score while satisfying a set of constraints. In our context, an assignment consists of (up to) k path specifications each with at most t feature values and over feature functions $q_1, ..., q_l$. The score of an assignment is the weighted sum of the features the assignment describes for every routing path in $\mathcal{R}$. We define the score in two steps: *(i)* the score of a feature function $q \in \{q_1, ..., q_l\}$ and *(ii)* the score of all feature functions.

Feature score A score function of a feature function q maps sets of up to k specifications to a real number score:

$$\Phi_q \colon \left(\mathcal{S}^t_{U_1,...,U_l} \cup \{\emptyset\}\right)^k \to \mathbb{R}$$

The domain consists of k-ary tuples, whose elements are specification sets or the empty set. The empty set $\emptyset$ denotes "no specification", and it enables us to cleanly capture specification sets with less than k specifications. To simplify definitions, we assume: $(d,P) \not\models \emptyset$ for all (d,P). For a set $\mathcal{S}$, the score $\Phi_q(\mathcal{S})$ is the weighted sum of routing paths in $\mathcal{R}$ for which q is described by a specification in $\mathcal{S}$. A path (d,P) is part of the sum if there is a specification $S \in \mathcal{S}$ containing a feature value of q that (d,P) satisfies. The weight of a path $w_{d,P}$ is a positive number (e.g., the traffic size). Formally:

$$\Phi_q(\mathcal{S}) = \sum_{(d,P) \in \mathcal{R}} w_{d,P} \cdot [\bigvee_{S \in \mathcal{S} \colon q(d,P) \in S} (d,P) \models S] \quad (1)$$

In this definition, $[\cdot]$ denotes the Iverson bracket that returns 1 if the formula is satisfied or 0 otherwise.

Specification set	Φ_E	Φ_{SP}	$\Phi_{E,SP}$
$\{\{\text{NY}_e\}\}$	1	0	1
$\{\{\text{LA}_e\}\}$	2	0	2
$\{\{1_{sp}\}\}$	0	3	3
$\{\{\text{NY}_e\}, \{\text{LA}_e, 1_{sp}\}\}$	3	2	5

Table 1: Score functions for $\mathcal{R} = \{(d_1,P_1),(d_2,P_2)\}$, where $w_{d_1,P_1} = 1$ and $w_{d_2,P_2} = 2$, $E(d_1,P_1) = \text{NY}_e$ and $E(d_2,P_2) = \text{LA}_e$, and $SP(d_1,P_1) = SP(d_2,P_2) = 1_{sp}$.

Example 1 Table 1 shows an example for $\mathcal{R} = \{(d_1,P_1),(d_2,P_2)\}$ with $w_{d_1,P_1} = 1$ and $w_{d_2,P_2} = 2$. Assume $E(d_1,P_1) = \text{NY}_e$, $E(d_2,P_2) = \text{LA}_e$, and that P_1 and P_2 are shortest paths: $SP(d_1,P_1) = SP(d_2,P_2) = 1_{sp}$. Then, $\Phi_E(\{\{\text{NY}_e\}\}) = 1 \cdot 1 + 2 \cdot 0 = 1$ and similarly $\Phi_E(\{\{\text{LA}_e\}\}) = 1 \cdot 0 + 2 \cdot 1 = 2$. Since both P_1 and P_2 are shortest paths, $\Phi_{SP}(\{\{1_{sp}\}\}) = 1 \cdot 1 + 2 \cdot 1 = 3$. However, $\Phi_{SP}(\{\{\text{NY}_e\}, \{\text{LA}_e, 1_{sp}\}\}) = 1 \cdot 0 + 2 \cdot 1 = 2$

Feature set score A score function of feature functions $q_1, ..., q_l$ maps k specifications of size at most t to a score:

$$\Phi_{q_1,...,q_l} \colon \left(\mathcal{S}^t_{U_1,...,U_l} \cup \{\emptyset\}\right)^k \to \mathbb{R}$$

The score is the sum of all the features' scores:

$$\Phi_{q_1,...,q_l}(\mathcal{S}) = \sum_{j \colon [1,l]} \Phi_{q_j}(\mathcal{S})$$

The last column of Table 1 shows the feature set score of the previous example. We can now define the optimization problem.

Definition 1 (Optimization Problem) Given a set of routing paths $\mathcal{R}$, weights $w_{d,P}$ for each $(d,P) \in \mathcal{R}$, a set of feature functions $q_1, ..., q_l$, a constant k limiting the number of path specifications, and a constant t limiting the size of path specifications, we formulate the network summarization problem as:

$$\underset{\mathcal{S} \in (\mathcal{S}^t_{U_1,...,U_l} \cup \{\emptyset\})^k}{\arg\max} \Phi(\mathcal{S})$$

[1] This is not a limitation, because values can be uniquely annotated.

Example 2 Let $\mathcal{R} = \{(\text{Google}, P_i)\}_{i=1}^{100}$, each with weight 1, and $k = t = 3$. We assume that *(i)* if $i \leq 60$, $E(\text{Google}, P_i) = \text{NY}_e$, and $E(\text{Google}, P_i) = \text{LA}_e$ otherwise, *(ii)* for $i \leq 40$, $SP(\text{Google}, P_i) = 1_{sp}$, and *(iii)* all other feature values are unique for every path. An optimal solution is: $\{\{\text{NY}_e\}, \{\text{Washington}_e\}, \{\text{NY}_e, 1_{sp}\}\}$, and its score is $\Phi_E + \Phi_{SP} = 100 + 40 = 140$. Another optimal solution is $\{\{\text{NY}_e\}, \{\text{Washington}_e\}, \{1_{sp}\}\}$. Though scores are identical, the operator is likely to prefer the former specification set as it provides additional information (e.g., all traffic following the shortest path exits in New York). We leverage this insight in §5.

Definition 1 can be refined by extending the objective function or adding constraints, as demonstrated in the next section. Also, while this problem can be considered as a general summarization problem suitable for other contexts, the skewed nature of traffic makes our context a better instantiation to this problem: the heavy traffic is likely to share many feature values which can lead to solutions that are clearly better than others. At the same time, these properties are exactly the kind of information that an operator needs in order to understand the behavior of the main part of the traffic.

One approach to solving this optimization problem is to phrase it as an integer linear program and use an off-the-shelf solver. We show such a formulation and a performance evaluation in Appendix A. Computing an exact solution to this NP-hard optimization problem is (expectedly) too expensive for practical use when summarizing a large number of paths. Instead, we introduce an approximate and scalable optimization algorithm, which we describe in the next sections.

5 Approximate Optimization

A key challenge when designing an inference algorithm for an NP-hard problem is dealing with the size of the search space that is at least exponential. In our setting, we show that the search space is exponential in both t and k, making the search very challenging (§5.1). Intuitively, this stems from the fact that we need to explore two dimensions: path coverage and path explainability. To address the issue with the large search space, we leverage the fact that traffic is skewed and focus on parts of it, enabling us to trade-off expressivity of the specification set with the size of the search space. We show that the optimal solution for this part of the search space: *(i)* has at least $min\{1/k, 1/t\}$ of the score of the optimal solution for the full search space, *(ii)* the length of every search path is polynomial in t, and *(iii)* the number of children of every node is polynomial in the number of feature values (§5.2). We further identify an equivalence relation over the path specifications and leverage it to define a search space with solutions of higher quality (§5.3).

5.1 An Exponential Search Space

In this section, we analyze the size of the search space, organize the solutions in a graph, and discuss the challenges of traversing it.

Size of search space We begin with showing that the size of the search space is exponential in t and k. The search space is the set of all specifications, that is $(\mathcal{S}^t_{U_1,\dots,U_l} \cup \{\emptyset\})^k$. Thus, it immediately follows that its size is exponential in k. To conclude that the size is exponential in k and t, we show that the size of $\mathcal{S}^t_{U_1,\dots,U_l}$ is exponential in t. To prove this, we reduce this computation to the combinatorial problem of choosing without replacement up to t feature functions from l feature functions (we assume $l \geq t$) and then for each, picking a feature value (we assume $|U_i| \geq 2$ for all i). Then, using a combinatorial identity [14, Vol. 2, (1.37)] we get:

$$\sum_{m=0}^{t} \binom{l}{m} \cdot 2^m \geq \sum_{m=0}^{t} \binom{t}{m} \cdot \frac{2^m}{m+1} = \frac{3^{t+1}-1}{2(t+1)}$$

Search space as a graph We organize the solutions in a directed graph $\mathcal{G}$. The nodes of $\mathcal{G}$ are the solutions: $(\mathcal{S}^t_{U_1,\dots,U_l} \cup \{\emptyset\})^k$. There is an edge (u, v) if v extends one of u's specifications with one feature value (Fig. 2). We distinguish between two kinds of edges: edges that extend an empty specification (colored blue) and those that extend an existing specification (colored red). Intuitively, the blue edges try to increase *coverage* by including more path specifications. This increases the number of routing paths for which the overall specification set holds. The red edges aim to increase the amount of detail captured in a path specification, resulting in better *explainability*. However, they can reduce the number of routing paths that satisfy the specification set (and thus, have the opposite effect of blue edges). Two extreme points in this coverage versus explainability exploration are: *(i)* specification sets that maximize explainability (specifications are of size t) and *(ii)* specification sets that maximize coverage (all specifications are of size 1). Depending on the weights and number of routing paths, the optimal solution sits in-between these two extremes.

Example 3 $\{\{\text{New York}_e\}\}$ maximizes coverage, while $\{\{\text{New York}_e, \text{Dallas}_i, \text{Google}_o, 1_{sp}\}\}$ explainability.

Search challenge An important ingredient in any search strategy is the solution scoring function, which guides the search towards the optimal result, while effectively pruning subspaces. In our setting, such a score function is even more critical as the size of the search space is exponential in k and t. An immediate candidate for a score function is Φ, as in Definition 1. However, Φ can guide us towards a good solution only if we restrict our traversal to nodes reachable through the blue edges. This is due

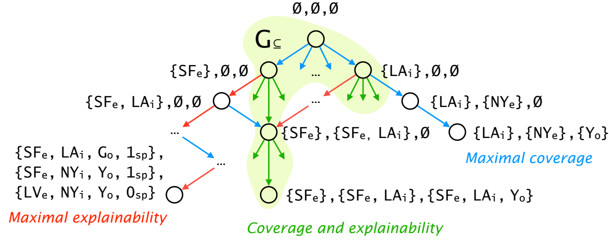


Figure 2: Part of the search space for k=3 specifications, t=4 feature values per specification and features egress *(e)*, ingress *(i)*, organization *(o)* and shortest path *(sp)*.

to a monotonicity property guaranteeing that if v is reachable from u only through blue edges, then $\Phi(v) > \Phi(u)$ (since v includes all feature values described by u). However, the red edges do not have this property for Φ (as it trades off path coverage with explainability). Even if we consider a different scoring function, pruning is unlikely to be effective and the traversal may end up exploring an exponential number of nodes. Instead, we consider a reduced subspace that has shorter paths and satisfies the monotonicity property for *every type of edge*.

5.2 A Reduced Search Space

In this section, we define a reduced space $\mathcal{G}_{\subseteq}$, which is a subspace of $\mathcal{G}$. Our reduced space leverages the fact that traffic is skewed, and thus the heavy part of the traffic shares many feature values. This means that specifications consisting of these common feature values have higher score than other specifications and that these higher-scored specifications intersect. This motivates us to focus only on solutions whose specifications are contained in one another. Such an approach guarantees that the solutions balance path coverage (provided by the shorter specifications) and explainability (provided by the larger specifications). We show that $\mathcal{G}_{\subseteq}$ contains solutions which are not significantly worse than an optimal solution in $\mathcal{G}$. Specifically, we show that $\mathcal{G}_{\subseteq}$ contains a solution whose score is at least $min\{\frac{1}{k}, \frac{1}{t}\}$ of an optimal solution in $\mathcal{G}$, in the worst case. In $\mathcal{G}_{\subseteq}$, the size of the search paths is t (instead of $t \cdot k$ as in $\mathcal{G}$), and every node has at most $\sum_{i=1}^{l} |U_i|$ children (instead of $k \cdot \sum_{i=1}^{l} |U_i|$).

The nodes of $\mathcal{G}_{\subseteq}$ are all specification sets whose path specifications are *extensions of one another*. More formally, a node has the property that its (nonempty) specifications can be ordered to $S_1, ..., S_m$ such that *(i)* $S_1 \subset ... \subset S_m$ and *(ii)* for all $1 \leq i \leq m$, $|S_i| = i$. For example, $\{\{\text{New York}_e\}, \{\text{New York}_e, 1_{sp}\}\}$ is a node in $\mathcal{G}_{\subseteq}$, while $\{\{\text{New York}_e\}, \{\text{Washington}_e\}\}$ is not.

The edges of $\mathcal{G}_{\subseteq}$ combine both kinds of edges of $\mathcal{G}$. Concretely, there is an edge (u, v) if v contains *all* specifications of u and *also* contains a specification that extends the largest specification of u with an additional feature value. More formally, if the (nonempty) specifications of u are ordered as defined before to $S_1, ..., S_m$, then v has the specifications $S_1, ..., S_m, S_{m+1}$ such that $S_m \subset S_{m+1}$ and $|S_{m+1}| = m + 1$. Fig. 2 highlights the nodes of $\mathcal{G}_{\subseteq}$ with a green background and shows the edges of $\mathcal{G}_{\subseteq}$ (which are different from the edges of $\mathcal{G}$) in green.

Optimality We now discuss how solution optimality in $\mathcal{G}_{\subseteq}$ relates to that in $\mathcal{G}$. Intuitively, there are two "worst case scenarios". First, if specifications are of size t, a solution of $\mathcal{G}_{\subseteq}$ that contains any such specification contains subsets of this specification as well, which "take the spot" of the other specifications, *without necessarily contributing to the score*. To illustrate this, consider the scenario where $k = 3$, $t = 4$ and there are 3 paths, p_1, p_2, p_3 with weight 1 whose feature values are $\{e_1, i_1, o_1, sp_1\}$, $\{e_2, i_2, o_2, sp_2\}$,$\{e_3, i_3, o_3, sp_3\}$, respectively (where e_n is an egress, i_n is an ingress, o_n is an organization, and sp_n is an indicator for shortest path). An optimal solution is to pick exactly these three feature values resulting in a score of 12. However, in $\mathcal{G}_{\subseteq}$, a solution that includes one of these specifications contains also its subsets, making the score of the optimal solution only 3. The other extreme is if all optimal solutions are of size 1. In this case, sets of size greater than 1 may add little gain to the score. To illustrate this, consider the scenario where $k = 3, t = 4$ and there are 12 paths, $p_1, \dots, p_{12}$ with weight 1 such that $p_1, \dots, p_4$ have property e_1, $p_5, \dots, p_8$ have property e_2 and $p_9, \dots, p_{12}$ have property e_3 (besides this, there are no common feature values). An optimal solution is $\{e_1\}$, $\{e_2\}$,$\{e_3\}$ whose score is 12. However, because of the structure of our space, the optimal solution has score 6.

The next lemma states that the maximum gap between the scores of the optimal solution in $\mathcal{G}_{\subseteq}$ and $\mathcal{G}$ is at most a factor of $min\{\frac{1}{t}, \frac{1}{k}\}$. Proof is provided in Appendix B.

Lemma 1 Let $OPT_{\mathcal{G}}, OPT_{\mathcal{G}_{\subseteq}}$ be the optimal solutions in $\mathcal{G}$ and $\mathcal{G}_{\subseteq}$. Then, $\min\{\frac{1}{t}, \frac{1}{k}\} \cdot \Phi(OPT_{\mathcal{G}}) \leq \Phi(OPT_{\mathcal{G}_{\subseteq}})$.

5.3 A Path Equivalent Space

In this section, we define a search space which is similar to $\mathcal{G}_{\subseteq}$ but may contain solutions with higher score. Intuitively, this is obtained by "merging" nodes in $\mathcal{G}_{\subseteq}$ that are equivalent with respect to the satisfying paths. In other words, for every two nodes in this space, there is at least one path satisfying one but not the other. Path equivalence does not imply the same score. For example, if $\{e_1\}, \{i_1\}$ are path equivalent, then $\{e_1, i_1\}$ is also path equivalent to them, but with a score twice as high from each (because each path contributes its weight twice, once per feature). By considering only nodes that are not path equivalent, we can potentially obtain better solutions, without sacrificing the lower bound of Lemma 1.

We use this observation to modify $\mathcal{G}$ to a space $\mathcal{G}_=$ whose solutions consist of specifications that are *(i)* contained in one another (like $\mathcal{G}_\subseteq$) and *(ii)* maximal with respect to path equivalence. In our example, this means that $\{e_1\},\{i_1\}$ are not part of any solution in $\mathcal{G}_=$, but $\{e_1,i_1\}$ might be if its extensions are not equivalent to it. In $\mathcal{G}_=$, there is an edge (u,v) if, for u whose specifications are $S_1 \subseteq ... \subseteq S_m$, we have *(i)* the specifications of v are $S_1,...,S_m,S_{m+1}$, *(ii)* $S_m \subset S_{m+1}$, and *(iii)* for any subset S such that $S_m \subset S \subset S_{m+1}$, S_{m+1} and S are path equivalent. By construction, $\mathcal{G}_=$ has solutions which are at least as good as those in $\mathcal{G}_\subseteq$, which gives us:

Lemma 2 Let $O_{\mathcal{G}_\subseteq}$ and $O_{\mathcal{G}_=}$ be optimal solutions in $\mathcal{G}_\subseteq$ and $\mathcal{G}_=$, respectively. Then, $\Phi(O_{\mathcal{G}_=}) \geq \Phi(O_{\mathcal{G}_\subseteq})$.

By traversing $\mathcal{G}_=$, algorithms can return solutions with larger path specifications than if they traversed $\mathcal{G}_\subseteq$. This follows since the maximal size of a specification in $\mathcal{G}_\subseteq$ is k, while the size of specifications in $\mathcal{G}_=$ is up to t.

6 The ComPass Algorithm

We now introduce ComPass, our algorithm for **com**puting **pa**th **s**pecification**s** by traversing the search space $\mathcal{G}_=$. ComPass (Algorithm 1) lazily computes nodes in $\mathcal{G}_=$ and continues to the node with the highest increase in score. It takes as input a set of routing paths $\mathcal{R}$, a set of feature functions $q_1,\dots,q_l$, and constants k and t denoting the maximal number of specifications and the maximal size of each path specification. ComPass starts by initializing the set of solutions $\mathcal{S}$ and the current specification L to the empty set and Q to the set of all candidate feature functions (Lines 1–3). In up to k iterations, the best feature value is selected to extend L according to the score function – namely, the feature value that will maximize the score of $\mathcal{S}$ as defined by the score function (Eq. (1)) when adding it to L . This can be formalized as maximizing the function on Line 5.

Let v be this feature value and q its feature. Then, L is extended with v and q is dropped as L cannot contain another feature value from U_q. The paths in $\mathcal{R}$ not meeting v are dropped as well, as these will not be described by the next specifications (Lines 6–8). Then, if the size of L reaches the bound t, the loop breaks as it is impossible to extend L further (Line 9). Otherwise, ComPass computes the maximal specification that is equivalent to L by checking whether it can be extended with other feature values (Lines 10–13). Finally, L is added to $\mathcal{S}$ (Line 14), and the next iteration begins. To ensure the limit of t is not exceeded, once L has reached this bound, ComPass completes and returns the current specification sets. This means that ComPass may return fewer than k specifications. It can be shown that this solution has a higher score than a solution with k specifications that are not representative of their class. Intuitively, this follows since the paths described by the descendants are subsumed by the paths described by their ancestors.

Algorithm 1: ComPass ($\mathcal{R}$, $q_1,\dots,q_l$, k, t)

```
Input  : R: a set of routing paths.
         q_1,...,q_l: a set of feature functions.
         k: limit on the number of specifications.
         t: limit on the size of specifications.
Output: A set of specifications S.
1  S = ∅                                   // The specification set
2  L = ∅                        // The last computed specification
3  Q = {q_1,...,q_l}                          // Candidate features
4  while |S| < k do
5      q, v = argmax_{q∈Q, v_q∈U_q} Σ_{(d,P)∈R} w_{d,P} · [q(d,P) = v_q]
6      L = L ∪ {v}
7      Q = Q \ {q}
8      R = R \ {(d,P) | q(d,P) ≠ v}
9      if |L| = t then S = S ∪ {L}; break
10     while ∃v ∈ U_Q.(L ∪ {v} ≡ L) do
11         L = L ∪ {v}
12         if |L| = t then S = S ∪ {L}; break
13         Q = Q \ {q}
14     S = S ∪ {L}
15 return S
```

Example 4 Consider $\mathcal{R} = \{(\text{Google},P_i)\}_{i=1}^{100}$, each with weight 1, and $k = t = 2$. As before, we assume that *(i)* if $i \leq 60$, $E(\text{Google},P_i) = \text{NY}_e$, and $E(\text{Google},P_i) = \text{LA}_e$ otherwise, *(ii)* for $i \leq 40$, $SP(\text{Google},P_i) = 1_{sp}$, and *(iii)* all other feature values are unique for every path. We now show how ComPass computes the optimal solution $\{\text{NY}_e\},\{\text{NY}_e,1_{sp}\}$. In its first iteration, ComPass discovers that the feature value NY_e maximizes the score. It thus extends L to $\{\text{NY}_e\}$, prunes the egress feature E from Q, and removes from $\mathcal{R}$ all paths whose egress is not NY. Since $\{\text{NY}_e\}$ is the representative of its equivalence class, it is added to $\mathcal{S}$. In the second iteration, the feature value 1_{sp} maximizes the score. Hence, ComPass extends L with 1_{sp}. Since the limit $t = 2$ has been reached, the loop breaks (Line 7), and $\{\text{NY}_e\},\{\text{NY}_e,1_{sp}\}$ is returned.

Finding the best feature value To avoid iterating every feature value separately in Line 5 (which can incur high overhead), we find the best feature value by iterating over the feature functions in Q and the routing paths in $\mathcal{R}$ and storing the score of each feature value in a hash table. Then, with a single pass over the hash table, we find the feature value with the highest score.

Guarantees Our next theorem states that ComPass computes a solution whose score is at least $\frac{1-f}{1-f^{\min\{t,k\}}}$ of the optimal solution in $\mathcal{G}_=$, where $f \in (0,1)$ is the maximal portion of paths that a child of a node can have. Note that

since ComPass explores $\mathcal{G}_{=}$, whose nodes are not path equivalent, f cannot be 1. Proof is in Appendix B.

Theorem 1 Given that there is $f \in (0,1)$ such that for every pair of path specifications A, A' if $A \subset A'$, then $\Phi(\{A\}) \leq f \cdot \Phi(\{A'\})$. Then, if O is the solution returned by ComPass, we have $\frac{1-f}{1-f^{\min\{t,k\}}} \cdot OPT_{\mathcal{G}_{=}} \leq O$.

Example 5 The factor f is determined by the pair of nodes $A \subset A'$ whose scores are the closest. In the previous example, $A = \{\{\text{NY}_e\}, \emptyset\}$, $A' = \{\{\text{NY}_e\}, \{\text{NY}_e, 1_{sp}\}\}$. Since $\Phi(A) = 60$ and $\Phi(A') = 100$, we get that $f = 0.6$. By the theorem, ComPass returns a solution whose score is at least 62.5% compared to the optimal solution in $\mathcal{G}$.

Speeding up ComPass by sampling To compute the best feature value, ComPass iterates in Line 5 over all routing paths to determine the best feature value. This step is very expensive, especially if the number of routing paths and feature functions is large. To mitigate this problem, we leverage two observations that allow ComPass to uniformly sample the routing paths instead of considering all routing paths. First, Internet traffic is heavily skewed, which means that most traffic is directed towards a few organizations (e.g., CDNs), and egresses see different traffic volumes depending on the peering. This means that sampling is likely to pick representative routing paths. Second, by the score function definition, optimal solutions consist of specifications describing the main part of the traffic. This means that specifications representing little traffic have little effect on the decisions ComPass makes. This implies that sampling will perform well as it is more likely to ignore the specifications with few routing paths than the ones with many.

7 From Specifications to Summaries

In this section, we describe how *Net2Text* produces a natural language summary given a specification set $\mathcal{S}$ (generated by ComPass). It begins by augmenting $\mathcal{S}$ with additional information in three steps. It then transforms the path specifications in $\mathcal{S}$ to natural language sentences using templates. In the first two steps, *Net2Text* augments $\mathcal{S}$ with information computed as a byproduct by ComPass (i.e., additional specification sets and the amount of traffic). In the third step, *Net2Text* extends $\mathcal{S}$ with high-level features, which cannot be directly computed by ComPass. We next describe these steps and exemplify them on our running example, $\mathcal{S} = \{\{\text{NY}_e\}, \{\text{NY}_e, 1_{sp}\}\}$.

Step 1: Adding path specifications *Net2Text* extends every $S \in \mathcal{S}$ with the next m (a parameter) best path specifications that have the same parent in $G_{=}$ and are values of the same feature function. These path specifications can be extracted from the computation of ComPass (in Line 5). In our example, for $m = 1$, this step results in adding $\{\text{Washington}_e\}$ to $\mathcal{S}$ as NY_e and Washington_e have the same parent and same feature function (egress). This will eventually be translated to a single sentence: *Google traffic exits in New York and Washington*.

Step 2: Adding traffic size Then, *Net2Text* extends every $S \in \mathcal{S}$ with the total weight of the paths it describes to let the operator understand how much traffic the summary covers. In our example, this gives $\{(60\%, \{\text{NY}_e\}), (40\%, \{\text{Washington}_e\}), (39.6\%, \{\text{NY}_e, 1_{sp}\})\}$.

Step 3: Computing high-level features Next, we extend $\mathcal{S}$ with high-level features (e.g., load-balancing, waypointing, or hot-potato routing) that are not properties of single paths but rather of *sets of paths*, i.e., entire specifications. Thus, these features can only be identified after ComPass computed the best specification set. Each high-level feature defines the criteria that a specification has to meet for it to hold. For example, for load-balancing, the ingress and egress of a specification have to be fixed and the paths described by it need to be disjoint. In our example, *Net2Text* inferred that a common waypoint for $(39.6\%, \{\text{New York}_e, 1_{sp}\})$ is Atlanta as all the paths in this specification go through Atlanta, and thus this specification is extended to $(39.6\%, \{\text{New York}_e, 1_{sp}, \text{Atlanta}_w\})$. In addition, *Net2Text* inferred that the traffic to Google experiences hot-potato routing as it has multiple egresses and all the traffic is forwarded to the closest one.

Step 4: Translation to natural language Lastly, $\mathcal{S}$ is translated to natural language sentences using templates. The sentences are a composition of multiple basic templates. To create fluency in the summary, *Net2Text* connects related sentences by building upon the previous sentence. In addition, it does not repeat information. For example, the second sentence in our example summary in Section 2 does not repeat that it refers to Google traffic, and the percentage shown is relative (i.e., $39.6\%/60\% = 66\%$). Namely, $\{(39.6\%, \text{NY}_e, 1_{sp}, \text{Atlanta}_w)\}$ is mapped to: *66% of the traffic exiting in New York follows the shortest path and crosses Atlanta*.

8 Parsing Queries

To leverage *Net2Text*'s summarization capabilities, the operator needs to provide the feature functions Q, t, k, and the routing paths $\mathcal{R}$. Typically, once Q, t and k are specified, the operator queries the network database to obtain $\mathcal{R}$. To simplify this, *Net2Text* allows the operator to submit queries in natural language which it then translates to SQL-like queries for the network database. In the following, we describe how *Net2Text* parses these queries expressed in natural language.

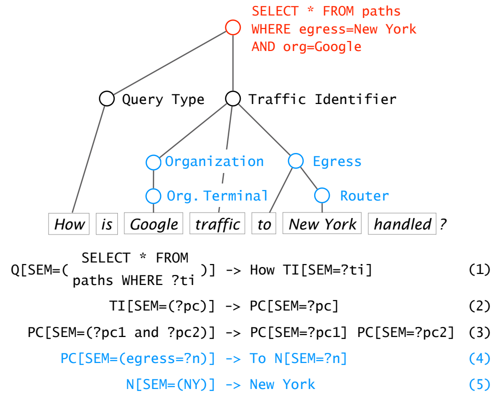


Figure 3: A parse tree and rules in the network grammar.

Network grammar The grammar consists of rules specifying how constituents of the queries (e.g., clauses, words) can be composed. The rules also specify the semantics of the constituents (e.g., the network terms such as "ingress"), which enables the parser to construct the SQL-like query. The grammar consists of two parts: *(i)* a structural part ($\sim$ 70 rules), which defines the allowed constituent compositions; and *(ii)* a domain-specific part consisting of mapping rules ($\sim$ 80 rules), which capture the network specific features (e.g., egress, organization) as well as keywords (e.g., router and location names). This split enables the operator to easily extend the grammar with new features and keywords without having to deal with the structure of the queries.

Structural grammar This grammar defines the query structure and its building blocks. We identify two main building blocks: query type and traffic identifier. Depending on the query type, there may be additional building blocks. There are four query types: yes/no ("Is/Does..."), counting ("How many..."), data retrieval ("What is/are..."), and explanation ("How is/does traffic..."). The query type determines whether the answer is yes/no, a count, a list, or a summary (obtained using ComPass). The traffic identifier defines the WHERE clause of the SQL-like query. The attributes selected by the query are either determined by the query (in data retrieval queries) or are simply a wildcard (i.e., $\star$).

Fig. 3 illustrates the structural parsing. *"How"* defines the desired behavior of *Net2Text* (summarize the data with ComPass), while "Google traffic to New York" is the traffic identifier. The black rules (1-3) are part of the structural grammar, while the blue rules (4 & 5) are part of the domain-specific grammar, which we discuss next.

Domain-specific grammar This grammar defines a mapping between keywords and names to features and their values. For example, the grammar defines the rules *(i)* "to $N \rightarrow$ egress=N" indicating that the natural language phrase "to N" means that N is a name of an egress, where N is a non-terminal and *(ii)* $N \rightarrow \text{NY}, \text{LA}, \ldots$, lists the possible egress names. Using these rules, "to NY" is parsed to `egress=NY` in the SQL-like query.

9 Evaluation

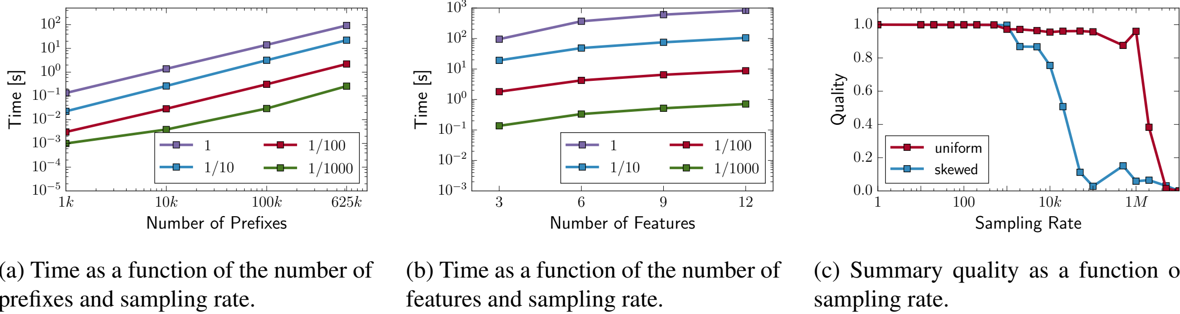
We evaluated *Net2Text*'s scalability and usability. For scalability, we show that *Net2Text* can summarize large forwarding state (§9.2) and generate summaries of high quality, even with sampling (§9.3). Worst-case queries complete within 2 seconds in large networks ($\sim$ 200 nodes). For usability, we show that *Net2Text* is useful for operators by conducting interviews (§9.4) and showcase its end-to-end implementation in a case study (§9.5).

9.1 Methodology

We run our Python-based prototype ($\sim$ 3*k* lines of code) on a machine with 24 cores at 2.3 GHz and 256 GB of RAM. For the experiments, we implemented an ISP-like forwarding state generator, which we use to produce realistic forwarding state for various Topology Zoo [15] topologies ranging from 25 to 197 nodes (Table 2). The generator enables us to control how "summarizable" a state is by varying how skewed it is.

Forwarding state generation Our generator synthesizes network-wide forwarding states (i.e., the set of routing paths $\mathcal{R}$) for a given number of IP prefixes and a given network topology in five consecutive steps. *First*, it randomly chooses a set of egress nodes (see Table 2). *Second*, it creates a prefix-to-organization mapping using the CAIDA AS-to-organization dataset [16]) and a full IPv4 RIB [17]. *Third*, for each organization, it chooses the number of egresses using an exponential distribution fitted according to real measurements [18, Fig.3.], after which the actual egresses are uniformly chosen from the set of egress nodes. *Fourth*, for each node, it computes its forwarding state by picking for each prefix the closest egress. *Fifth*, each routing path (d, P) is finally associated with an amount of traffic sampled from an exponential distribution. This leads to few organizations owning many prefixes, carrying relatively more traffic than others (as shown in [12]). The generator can also generate extra features whose values are arbitrarily picked.

Generality While we generate the input forwarding state, we stress that our results are representative because: *(i)* the scalability of ComPass does *not* depend on the actual feature values but only on the number of features (see §6); and *(ii)* the quality analysis does not depend on the actual score but rather on the ratio compared to other scores under the same setting.



(a) Time as a function of the number of prefixes and sampling rate.

(b) Time as a function of the number of features and sampling rate.

(c) Summary quality as a function of sampling rate.

Figure 4: *Net2Text* scales—both in the size of the forwarding state (sub-second, even for 625k prefixes) *(a)* and the number of features (sub-second, even for 12 features) *(b)*—while sampling affects the summary quality marginally *(c)*.

Topology	Nodes	Egresses	No sampling	1/1000
ATT NA	25	10	94.07 s	**0.26 s**
Switch	42	15	128.12 s	**0.24 s**
Sinet	74	30	223.91 s	**0.50 s**
GTS CE	149	40	611.81 s	**1.18 s**
Cogent	197	50	766.61 s	**1.84 s**

Table 2: With sampling (§6), *Net2Text* summarizes large network forwarding states (> 600k prefixes), *within 2 seconds*, for networks with close to 200 nodes.

9.2 Scalability Analysis

We evaluate *Net2Text* scalability by measuring the time it takes to summarize all routing paths (worst-case) while varying the number of key dimensions: prefixes, nodes, and feature functions. To evaluate the sampling optimization of ComPass, we run ComPass four times: without path sampling and with sampling rate of $1/10, 1/100, 1/1000$. We repeated each experiment 10 times and report median results (std dev is small).

Fig. 4a shows the results when varying the number of prefixes from 10^3 to 10^5 and the full RIB for the ATT NA topology using 3 feature functions. The results indicate that *Net2Text* scales linearly in the number of prefixes. The running time decreases proportionally to the sampling rate. Without sampling, summarizing forwarding states with 625k prefixes takes about 100 seconds and *less than one second* with a sampling rate of 1/1000. Fig. 4b shows a similar trend when varying the number of features from 3 to 12 and using a full RIB.

Table 2 shows the results when considering different topology sizes, with full routing tables (625k prefixes) and 3 feature functions. The table also reports results with (rate of $1/1000$) and without sampling. We see that the runtime is roughly linear in the number of nodes in the network. More importantly, our results indicate that *Net2Text* scales to large networks with hundreds of nodes thanks to sampling: it takes less than 2 seconds for *Net2Text* to summarize Cogent forwarding state.

9.3 Quality Analysis

We now evaluate the effect of sampling the input data (i.e. the forwarding paths) and show that doing so only marginally impacts the quality of the summary. In addition, we show that ComPass compares well against two baselines both in terms of quality and running time.

We measure the quality of a summary using the score function presented in §4. Intuitively, the score represents the traffic volume of the paths covered by the resulting summary, rewarding more detailed summaries by multiplying the volume of each path by the number of details (feature values) present in the summary. When computing the score, we always account for all entries that match the resulting summary and not just for the sampled entries. As in §9.2, we consider the problem of summarizing every single entry in the network database.

For the experiment, we generate forwarding state for the ATT NA topology with a full routing table and vary the sampling rate from 1 to 1/5,000,000. Note that we have more entries in the network database than the total number of prefixes as there is at least one path from every node to every prefix. Hence, even with sampling rates higher than the number of prefixes, we still have paths to summarize. For this setup, we have more than 15 million entries in the network database.

Fig. 4c shows the score of the summary for different sampling rates normalized to the score without sampling. We ran the experiment for two different scenarios: *(i)* highly skewed traffic distributions among the feature values, where the size difference between the feature values is high; and *(ii)* uniform distributions, where the difference between them is low. Our results show that the sampling rate at which the score of the summary drops significantly is very high. Even with sampling rates of 1/1000, ComPass still creates summaries whose qualities are within 5% of the unsampled summary.

To further illustrate the quality of ComPass summaries, we compare it against two baselines. Both iterate once over the relevant routing paths

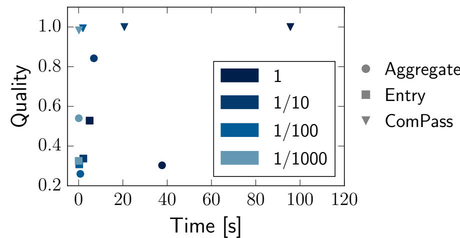


Figure 5: *Net2Text* produces summaries of higher quality than two simple baselines.

and pick the most detailed specification (e.g., $\{$New York$_e$, Philadelphia$_i$, Google$_o\}$). From this specification, we build the full specification set by randomly removing one feature value after the other to obtain for example $\{\{$New York$_e\}$, $\{$New York$_e$, Google$_o\}$, $\{$New York$_e$, Philadelphia$_i$, Google$_o\}\}$. The baselines differ in how they choose the most detailed specification: *Entry*, takes the routing path with the highest weight and uses it as the most detailed specification; and *Aggregate*, aggregates all routing paths with the same feature values and uses the largest aggregate. When computing the quality of the summaries, we consider all routing paths matching the resulting summary. Thanks to sampling, ComPass produces *higher* quality summaries in the same amount of time as the two baselines (see Fig. 5). If we also consider the information added to the summary by extending it as described in 7, we see that ComPass outperforms the baselines by almost 5 times.

9.4 Usefulness

To better assess the usability of *Net2Text*, we conducted five interviews with network operators of ISP networks (research, Tier 1 and Tier 2) and one enterprise network. We questioned them about four aspects.

Aspect 1: Need for virtual assistants All operators see opportunities for virtual assistants in tasks requiring to process a lot of data to identify and extract the relevant information. An assistant allows them to focus on remedying, rather than identifying and analyzing the event.

Aspect 2: Relevance of the NL input The possibility to write queries in natural language was well-perceived. Some operators, however, do not mind a fixed query language or writing their own scripts.

Aspect 3: Relevance of the NL output Most operators see value in natural language summaries as they are concise and simple to understand, especially for less technical persons. Depending on the query, some operators mentioned that they would like to see visualizations of the summary (e.g., a graph) in addition to text.

Aspect 4: Usefulness of *Net2Text* queries All operators confirmed that the queries currently supported by *Net2Text* are relevant. In particular, they appreciated the ability to query about incoming traffic. In addition, most operators testified interest in service-oriented queries, instead of purely destination-oriented ones (e.g., traffic to the Gmail-service instead of Google traffic in general).

In the discussions, we saw a clear difference between the queries of ISP and enterprise network operators. While the ISP operators were mostly concerned about where traffic was entering and leaving the network, the enterprise operator was more interested in the status of the different applications running in the network and their policies (e.g., is there always a firewall on the path).

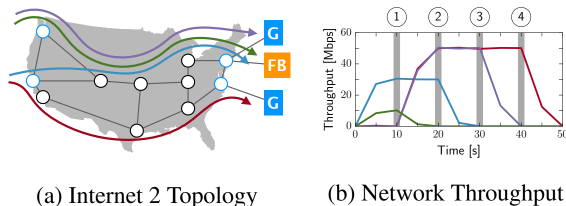
9.5 Case Study

We showcase our end-to-end implementation of *Net2Text* by running it in a Quagga-based network emulating Internet2 (Fig. 6a). Routers in Seattle, Sunnyvale, New York and Washington are connected to external peers. The router in New York receives routes to both Google and Facebook, the router in Washington only to Google. All external routes have the same local-preference. We generate transit traffic entering via Seattle and Sunnyvale towards both destinations. The flows are highlighted in Fig. 6a and the measured throughput is depicted in Fig. 6b. Every ten seconds, *Net2Text* summarizes the entire forwarding state as indicated by the grey bars.

Table 3 shows the 4 summaries produced by *Net2Text*. We see that *Net2Text* is able to explain the current forwarding behavior at different levels of detail and automatically zoom in on the largest part of the traffic. At the time of the second summary, for example, traffic for Google has spiked (purple and red) and is now three times larger than Facebook. We see that *Net2Text* automatically focuses on the traffic to Google and provides more details about it, yet it still mentions traffic to Facebook. In the third summary, we see how *Net2Text* captures higher-level constructs that are not directly present in the database such as "hot-potato routing" (§7).

10 Discussion

Why natural language? We believe that a chat-like interface provides a familiar and intuitive way for operators to interact with their network. That said, our summarization contribution is useful in its own right, independently of the NLP interface. As an illustration, we could easily translate *Net2Text* summaries to a graph-based representation (e.g. using PGA [11]) rather than natural language.



(a) Internet 2 Topology (b) Network Throughput

Figure 6: We ran our *Net2Text* implementation in a *live* network emulating Internet 2 (a) and vary the network throughput according to (b).

①	*"Traffic has a single egress (New York), and goes to a single destination (Facebook). It enters at the following ingresses: Sunnyvale (76%) and Seattle (24%)."*
②	*"Traffic goes to the following destinations: Google and Facebook. Traffic for Google exits through Washington (50%) and New York (50%)."*
③	*"Traffic is destined to Google. It experiences hot-potato routing. It exits through the following egresses: New York (50%) and Washington (50%)."*
④	*"Traffic leaves through Washington, has a single ingress (Sunnyvale), and goes to Google."*

Table 3: Actual summaries produced by our *Net2Text* implementation when run on the network depicted in Fig. 6.

What about new feature functions? While we only deal with a limited set of features in this paper, we stress that ComPass is flexible and can deal with *any* features defined over paths. Additional features (e.g. such as the TCP port number) can be easily added by adding a new field to the database. For the translation, the singular and plural of the feature name also have to be added to the rules. The operator can also add a mapping of feature values to some string, e.g., TCP port 80 to HTTP.

What about the network database? We assume that the network database is fed with high-quality *and* consistent data and focus on the problem of summarizing it. This is a strong assumption. Gathering high-quality state consistently is challenging and the quality of our summaries will inevitably suffer should the data be incomplete, outdated or inconsistent. Fortunately, multiple works have looked at the problem of extracting network data in a fast and consistent manner, which *Net2Text* can directly leverage. In particular, Libra [19] tackles the problem of capturing consistent snapshots of the network forwarding state. Similarly, FlowRadar [20] and Stroboscope [21,22] tackle the problem of quickly gathering fine-grained traffic statistics. Yet, summarizing network-wide behavior in the presence of incorrect or inconsistent data is an interesting problem we plan to address in future work.

11 Related work

Network verification & testing *Net2Text* directly complements previous initiatives on data-plane [5–10] and control-plane verification [1–4] as it does not aim at verifying, but *explaining* network-wide behavior. The reason is that even perfectly correct networks might exhibit unwanted or suboptimal behaviors at runtime, for instance, due to unforeseen traffic shifts or partial failures.

Network provenance *Net2Text*'s high-level objectives of *explaining how networks behave* bear similarities with many works on Network Provenance (e.g., [23–29]). The main difference between these works and *Net2Text* is that *Net2Text* does not aim at explaining *why* a particular state is observed (by following the derivation history), but rather summarizing *what* is the current state being observed to make it understandable to human operators. *Net2Text* can therefore be seen as complementary to these frameworks. Once the network operator understands what is the network behavior, he or she can then ask questions about why. We also believe that *Net2Text*'s summarizing capabilities can be applied to summarize provenance explanations which often tend to be large.

Connecting natural languages and networks A prior work [30] introduced NLP techniques to network management. It proposes to use natural language as interface between operators and an SDN network. Unlike *Net2Text*, it does not provide any abstraction capability and is limited to simple yes/no questions/answers along with simple control tasks such as rate limiting a flow.

12 Conclusions

We presented *Net2Text*, a novel approach to assist network operators in reasoning about network forwarding behaviors. *Net2Text* is based on efficient summarization techniques which generate interpretable summaries (in natural language) out of low-level forwarding rules. We propose an efficient approximation algorithm (with provable bounds) to solve the summarization problem. We fully implemented *Net2Text* and showed that it is highly effective—it only takes 2 seconds to summarize the state of hundreds of routers carrying full routing tables.

Acknowledgements

We are grateful to our shepherd Ranjita Bhagwan, the anonymous reviewers, Roland Meier and Dimitar Dimitrov for the constructive feedback and comments. We are also grateful to all the network operators who provided feedback and insights about *Net2Text*.

References

[1] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. A General Approach to Network Configuration Analysis. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[2] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis Using an Abstract Representation. In *ACM SIGCOMM*, Florianópolis, Brasil, 2016.

[3] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *ACM OOPSLA*, Amsterdam, Netherlands, 2016.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *ACM SIGCOMM*, Los Angeles, CA, USA, 2017.

[5] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*, Toronto, Canada, 2011.

[6] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*, San Jose, CA, USA, 2012.

[7] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking using Header Space Analysis. In *USENIX NSDI*, Lombard, IL, USA, 2013.

[8] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *USENIX NSDI*, Lombard, IL, USA, 2013.

[9] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking Beliefs in Dynamic Networks. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[10] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. SymNet: Scalable Symbolic Execution for Modern Networks. In *ACM SIGCOMM*, Florianópolis, Brasil, 2016.

[11] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *ACM SIGCOMM*, London, United Kingdom, 2015.

[12] Jennifer Rexford, Jia Wang, Zhen Xiao, and Yin Zhang. BGP Routing Stability of Popular Destinations. In *ACM IMC*, Marseille, France, 2002.

[13] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[14] Henry Wadsworth Gould. *Combinatorial Identities: A Standardized Set of Tables Listing 500 Binomial Coefficient Summations*. Morgantown, 1972.

[15] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *IEEE JSAC*, October 2011.

[16] CAIDA. AS Organizations Dataset, 2017-04-01. `http://www.caida.org/data/as-organizations`.

[17] CAIDA. BGPStream. `https://bgpstream.caida.org/`.

[18] Jaeyoung Choi, Jong Han Park, Pei-chun Cheng, Dorian Kim, and Lixia Zhang. Understanding BGP Next-Hop Diversity. In *IEEE Global Internet Symposium*, Shanghai, China, 2011.

[19] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *USENIX NSDI*, Seattle, WA, USA, 2014.

[20] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*, Santa Clara, CA, USA, 2016.

[21] Olivier Tilmans, Tobias Bühler, Ingmar Poese, Stefano Vissicchio, and Laurent Vanbever. Stroboscope: Declarative Network Monitoring on a Budget. In *USENIX NSDI*, Renton, WA, USA, 2018.

[22] Olivier Tilmans, Tobias Bühler, Stefano Vissicchio, and Laurent Vanbever. Mille-Feuille: Putting ISP Traffic under the scalpel. In *ACM Hotnets*, Atlanta, GA, USA, 2016.

[23] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *ACM SIGCOMM*, Florianópolis, Brasil, 2016.

[24] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Diagnosing Missing Events in Distributed Systems with Negative Provenance. In *ACM SIGCOMMM*, Chicago, IL, USA, 2014.

[25] Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. Distributed Time-aware Provenance. In *VLDB*, Riva del Garda, Trento, Italy, 2013.

[26] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient Querying and Maintenance of Network Provenance at Internet-Scale. In *ACM SIGMOD*, Indianapolis, IN, USA, 2010.

[27] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *USENIX ATC*, Portland, OR, USA, 2011.

[28] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *ACM SIGCOMMM*, Chicago, IL, USA, 2014.

[29] Colin Scott, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing Faulty Executions of Distributed Systems. In *USENIX NSDI*, Santa Clara, CA, USA, 2016.

[30] Azzam Alsudais and Eric Keller. Hey Network, Can You Understand Me? In *IEEE INFOCOM Workshop on Software-Driven Flexible and Agile Networking*, Atlanta, GA, USA, 2017.

[31] Gurobi Optimization, Inc. Gurobi Optimizer Reference Manual, 2016.

A ILP Formulation

$$\max \sum_{(d,P)\in\mathcal{R}} \sum_{1\leq i\leq k} \sum_{v\in U_1\cup\ldots\cup U_l} w_{d,P}\cdot y_{d,P,i,v}$$

$$\sum_{v\in U_j} x_{i,v} \leq 1 \quad (1)$$

$$\sum_{v\in U_1\cup\ldots\cup U_l} x_{i,v} \leq t \quad (2)$$

$$y_{d,P,i} - y_{d,P,v} + x_{i,v} \leq 1 \quad (3)$$

$$y_{d,P,i} + x_{i,v} - y_{d,P,i,v} \leq 1 \quad (4.1)$$

$$y_{d,P,i,v} - y_{d,P,i} \leq 0 \quad (4.2)$$

$$y_{d,P,i,v} - x_{i,v} \leq 0 \quad (4.3)$$

$$\sum_{1\leq i\leq k} y_{d,P,i} \leq 1 \quad (5)$$

$$x_{i+1,v} - x_{i,v} \geq 0 \quad (6)$$

$$y_{d,P,i}, x_{i,v}, y_{d,P,i,v} \in \{0,1\}$$

Figure 7: An integer program for computing a specification set to explain the routing paths. $i \in \{1,...,k\}, j \in \{1,...,l\}, (d,P) \in \mathcal{R}, v \in \{U_1 \cup \ldots \cup U_l\}$

In the following, we show how to formulate the inference problem from Section 4 as an integer linear program (ILP) where the objective encodes the score function Φ and the constraints encode the path specification search space $\mathcal{S}^t_{U_1,\ldots,U_l}$.

Variables We have two kinds of variables: the x-variables which encode the path specification set, and the y-variables which encode the features and specifications that paths meet. For each path specification, we introduce a set of variables, one for every feature value that may be in the path specification. Formally, we have a variable $x_{i,v}$ for every $1 \leq i \leq k$ and $v \in U_1 \cup \ldots \cup U_l$. These variables are indicator functions and range over $x_{i,v} \in \{0,1\}$. That is, if $x_{i,v} = 1$, it means that v is part of the ith path specification ($v \in S_i$), otherwise v is excluded. Thus, an assignment to the x's uniquely defines a path specification set.

The y variables encode whether paths meet the path specifications and which of their features are described by the specs. Concretely, for every routing path $(d,P) \in \mathcal{R}$, we maintain multiple binary variables:

- $y_{d,P,i}$: encodes whether (d,P) meets the ith specification.
- $y_{d,P,v}$: indicates whether (d,P) contains a feature value of v. Note that the values $y_{d,P,v}$ are known a-priori and need not be computed during optimization.
- $y_{d,P,i,v}$: encodes whether the feature v of P is described by the ith specification.

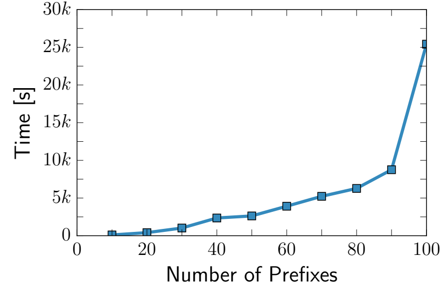


Figure 8: Running time using the ILP (optimal, but slow).

These variables allow us to capture precisely in what detail a path is being described by a specification that it meets. Note that $y_{d,P,i,v}$ can be 1 only if (d,P) meets the ith specification and the ith specification has feature v (i.e., $y_{d,P,i} = x_{i,v} = 1$). This requirement will be encoded as part of the general constraints.

Objective function We encode the objective function of Definition 1 as the weighted sum of $y_{d,P,i,v}$ variables.

Constraints The path specification space is expressed as a set of constraints which states that each path specification can have at most one feature value for the same feature (constraint set (1) in Fig. 7) and at most t features across all features (constraint set (2) in Fig. 7). The next constraint sets encode the score function. Constraint set (3) encodes whether the routing path (d,P) meets the ith specification. Intuitively, the constraints can be presented as $y_{d,P,i} \leq 1 + (y_{d,P,v} - x_{i,v})$, which means that $y_{d,P,i}$ can be 1 (to indicate that (d,P) meets the ith specification) only if $y_{d,P,v} \geq x_{i,v}$ for all v, which indicate that the routing path meets all features in the ith specification. Constraint set (4) in Fig. 7 encodes whether the feature value v of a routing path (d,P) is described, which may only be true if (d,P) meets the specification and that the specification contains v. Lastly, the constraint set (5) guarantees that each feature value v met by (d,P) is counted only once. The total number of variables and constraints is $O(k \cdot |\mathcal{R}| \cdot |U_1 \cup \ldots \cup U_l|)$.

As we explain in Section 5, it is useful to impose a certain shape or relation between the path specifications. In particular, we will see why it is useful to require path specifications to be extensions of one another. Constraint set (6) in Fig. 7 encodes this optional requirement.

Scalability To show the need for an efficient algorithm like ComPass, we evaluate the scalability of solving the corresponding ILP (Fig. 7, with all constraints, including (6)) using the `gurobi` solver [31]. Fig. 8 shows the running times for the ATT North America network with

a forwarding state encompassing between 10 to 100 prefixes (up to three orders of magnitude smaller than the experiments in §9.2). Unsurprisingly, the running time quickly explodes due to the large number of variables and constraints. With only *100 prefixes*, the ILP already requires more than 25k seconds to complete.

B Proofs

In this section, we provide the proofs for the lemmas and theorems presented in the paper.

Lemma 1 proof sketch Denote the optimal solution as the specification set: $\{x_1^1,,...,x_{t_1}^1\},\ldots,\{x_1^k,...,x_{t_k}^k\}$. By the score definition and since $t_i \leq t$, $\forall i,j. \frac{j}{t}\Phi(\{x_1^i,,...,x_{t_i}^i\}) \leq \Phi(\{x_1^i,...,x_j^i\})$. W.l.o.g., assume that $\{x_1^1,,...,x_{t_1}^1\}$ has the highest score. Then, $\Phi(\{x_1^1,,...,x_{t_1}^1\}) > OPT/k$. We split to cases. If $k \leq t_1$, then the score of $\{x_1^1\},\{x_1^1,x_2^1\},...,\{x_1^1,...,x_k^1\}$ is at least $k/t \cdot (OPT/k)$. Since $\{\{x_1^1\},...,\{x_1^1,...,x_1^k\}\}$ is a node in $\mathcal{G}_{\subseteq}$, the claim follows. Otherwise, if $t_1 < k$, then $\{\{x_1^1\},...,\{x_1^1,,...,x_{t_1}^1\}\}$ is a node in $\mathcal{G}_{\subseteq}$ and since $\Phi(\{x_1^1,,...,x_{t_1}^1\}) > OPT/k$, the claim follows.

Theorem 1 proof sketch Let the optimal solution be $OPT = \{\{a\},\{a,b\},...,\{a,b,...,m\}\}$ and the specification set that ComPass returned be the specification set $S_{ComPass} = \{\{a'\},\{a',b'\},...,\{a',b',...,m'\}\}$. By the assumption, $\Phi(\{a,b\}) \leq f \cdot \Phi(\{a\}) \leq f \cdot \Phi(\{a'\})$. By induction, $\Phi(\{a,b,...,j\}) \leq f^{|\{a,...,j\}|} \cdot \Phi(\{a'\})$. Since the length of the largest specification in OPT is $\min\{k,t\}$, the length of the optimal solution is at most $\Sigma_{1\leq j\leq \min\{k,t\}} f^j \cdot \Phi(\{a\}) = \frac{1-f^{\min\{t,k\}}}{1-f} \cdot \Phi(\{a\})$. By the greedy operation, we have $\Phi(\{a\}) \leq \Phi(\{a'\})$. Since $\Phi(\{a'\}) \leq \Phi(S_{ComPass})$, we get $\Phi(\{a\}) \leq \Phi(S_{ComPass}) \leq \frac{1-f^{\min\{t,k\}}}{1-f} \cdot \Phi(\{a\})$, which means that ComPass is a $\frac{1-f}{1-f^{\min\{t,k\}}}$-approximation algorithm.