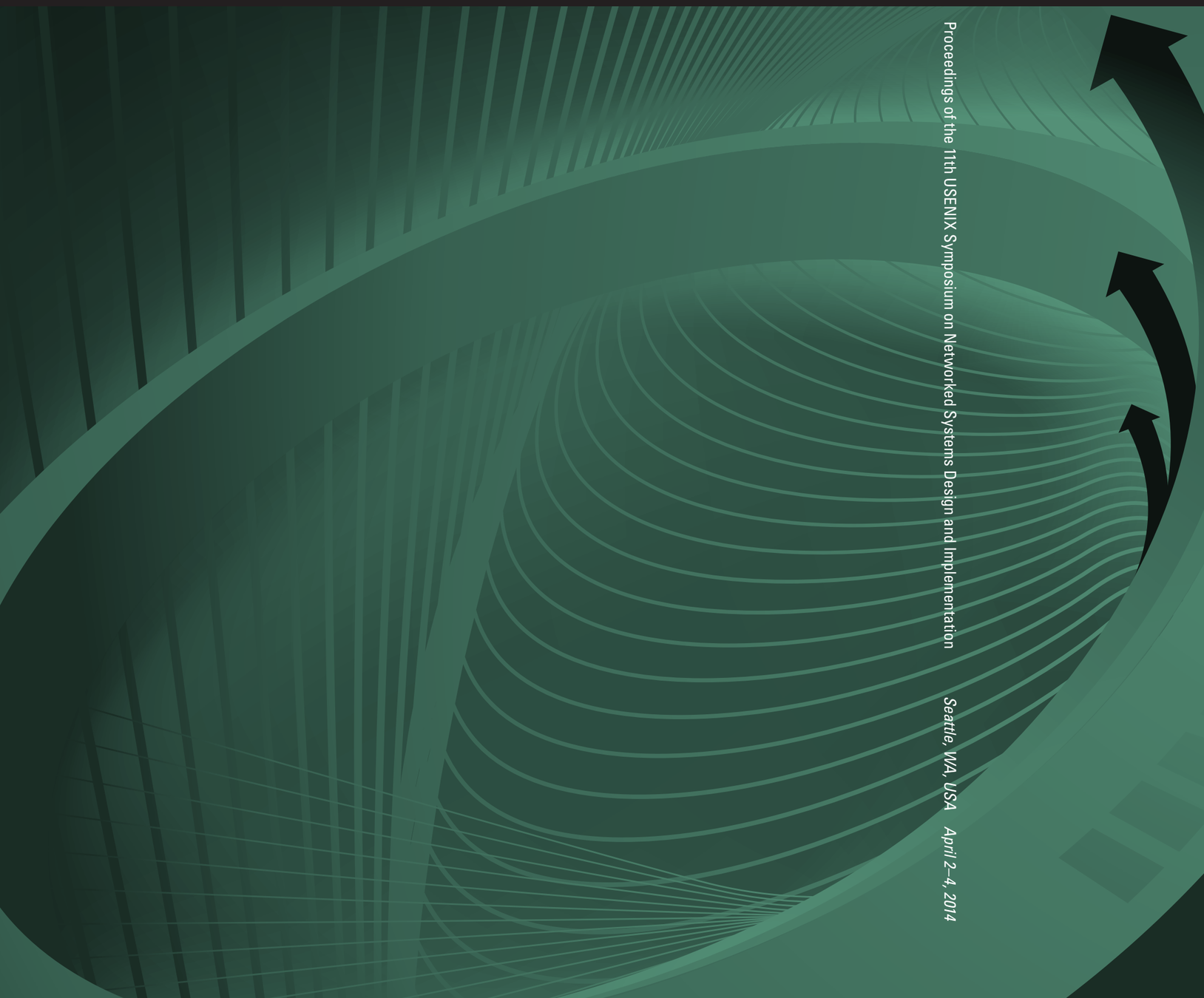
conference
proceedings



11th USENIX Symposium on Networked Systems Design and Implementation

Seattle, WA, USA
April 2–4, 2014

Sponsored by
usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION
In cooperation with ACM SIGCOMM and ACM SIGOPS

Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation
Seattle, WA, USA April 2–4, 2014

Thanks to Our NSDI '14 Sponsors

Gold Sponsors



Silver Sponsors


Bronze Sponsors



General Sponsor

Media Sponsors and Industry Partners

ACM *Queue*
ADMIN magazine
Distributed Management Task Force (DMTF)
HPCwire
InfoSec News
Linux Pro Magazine
LXer
No Starch Press
O'Reilly Media
Raspberry Pi Geek
UserFriendly.org

ISBN 978-1-931971-09-6

Thanks to Our USENIX and LISA SIG Supporters

USENIX Patrons

Google Microsoft Research NetApp VMware

USENIX Benefactors

Akamai *Linux Pro Magazine* Puppet Labs

USENIX and LISA SIG Partners

Cambridge Computer Google

USENIX Partners

EMC

USENIX Association

Proceedings of NSDI '14:
11th USENIX Symposium on Networked Systems Design and Implementation

April 2–4, 2014
Seattle, WA

Conference Organizers

Program Co-Chairs

Ratul Mahajan, *Microsoft Research Redmond*
Ion Stoica, *University of California, Berkeley*

Program Committee

Katerina Argyraki, *EPFL*
Aruna Balasubramanian, *University of Washington*
Miguel Castro, *Microsoft Research Cambridge*
Prabal Dutta, *University of Michigan*
Mike Freedman, *Princeton University*
Ali Ghodsi, *KTH Royal Institute of Technology and University of California, Berkeley*
Brighten Godfrey, *University of Illinois at Urbana-Champaign*
Sharon Goldberg, *Boston University*
Shyamnath Gollakota, *University of Washington*
Ramesh Govindan, *University of Southern California*
Saikat Guha, *Microsoft Research India*
Kyle Jamieson, *University College London*
Ethan Katz-Bassett, *University of Southern California*
S. Keshav, *University of Waterloo*
Changhoon Kim, *Microsoft*
Dejan Kostić, *IMDEA Networks*
T. V. Lakshman, *Bell Labs*
Nikolaos Laoutaris, *Telefónica*
Bruce Maggs, *Duke University*
KyoungSoo Park, *KAIST*
George Porter, *University of California, San Diego*
Lili Qiu, *University of Texas*
Sanjay Rao, *Purdue University*
Jennifer Rexford, *Princeton University*
Vyas Sekar, *Stony Brook University*
Srinivasan Seshan, *Carnegie Mellon University*
Jonathan Smith, *University of Pennsylvania*
Nina Taft, *Technicolor*
Amin Vahdat, *Google and University of California, San Diego*
Walter Willinger, *Niksun*
Yuan Yu, *Microsoft*
Ben Zhao, *University of California, Santa Barbara*

Poster Session Co-Chairs

T. S. Eugene Ng, *Rice University*
Amar Phanishayee, *Microsoft Research Redmond*

Steering Committee

Casey Henderson, *USENIX Association*
Arvind Krishnamurthy, *University of Washington*
Brian Noble, *University of Michigan*
Jennifer Rexford, *Princeton University*
Mike Schroeder, *Microsoft Research*
Alex C. Snoeren, *University of California, San Diego*
Chandu Thekkath, *Microsoft Research*

External Reviewers

Rachit Agarwal
Peter Bailis
Andrew Blake
Kirill Bogdanov
Mosharaf Chwdhury
Karthik Dantu
Tobias Flach
Mohammad Hajjat
Dongsu Han
George Katsikas
Maciej Kuzniar
Jie Liu
Mihai Moraru
Shankaranarayanan Narayanan
Miguel Peon
Peter Peresini
Ashiwan Sivakumar
Balajee Vamanan
Milan Vojnovic
Anduo Wang

NSDI '14: 11th USENIX Symposium on Networked Systems Design and Implementation
April 2–4, 2014
Seattle, WA

Message from the Program Co-Chairs vii

Wednesday, April 2, 2014

Datacenter Networks

Circuit Switching Under the Radar with REACToR 1
He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter, *University of California, San Diego*

Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Center 17
Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin, *Tsinghua University*

High Throughput Data Center Topology Design 29
Ankit Singla, P. Brighten Godfrey, and Alexandra Kolla, *University of Illinois at Urbana–Champaign*

Debugging Complex Systems

Adtributor: Revenue Debugging in Advertising Systems 43
Ranjita Bhagwan, Rahul Kumar, Ramachandran Ramjee, George Varghese, Surjyakanta Mohapatra, Hemanth Manoharan, and Piyush Shah, *Microsoft*

DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps 57
Bin Liu, *University of Southern California;* Suman Nath, *Microsoft Research;* Ramesh Govindan, *University of Southern California;* Jie Liu, *Microsoft Research*

I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks 71
Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown, *Stanford University*

Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks 87
Hongyi Zeng, *Stanford University;* Shidong Zhang and Fei Ye, *Google;* Vimalkumar Jeyakumar, *Stanford University;* Mickey Ju and Junda Liu, *Google;* Nick McKeown, *Stanford University;* Amin Vahdat, *Google and University of California, San Diego*

Software Verification and Testing

Software Dataplane Verification 101
Mihai Dobrescu and Katerina Argyraki, *École Polytechnique Fédérale de Lausanne*

NetCheck: Network Diagnoses from Blackbox Traces 115
Yanyan Zhuang, *Polytechnic Institute of New York University and University of British Columbia;* Eleni Gessiou, *Polytechnic Institute of New York University;* Steven Portzer, *University of Washington;* Fraida Fund and Monzur Muhammad, *Polytechnic Institute of New York University;* Ivan Beschastnikh, *University of British Columbia;* Justin Cappos, *Polytechnic Institute of New York University*

Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems 129
Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin, *The University of Texas at Austin*

(Wednesday, April 2, continues on p. iv)

Security and Privacy

ipShield: A Framework For Enforcing Context-Aware Privacy 143
Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, and Mani Srivastava, *University of California, Los Angeles*

Building Web Applications on Top of Encrypted Data Using Mylar 157
Raluca Ada Popa, *MIT/CSAIL;* Emily Stark, *Meteor, Inc.;* Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan, *MIT/CSAIL*

PHY Covert Channels: Can you see the Idles? 173
Ki Suh Lee, Han Wang, and Hakim Weatherspoon, *Cornell University*

cTPM: A Cloud TPM for Cross-Device Trusted Applications. 187
Chen Chen, *Carnegie Mellon University;* Himanshu Raj, Stefan Saroiu, and Alec Wolman, *Microsoft Research*

Thursday, April 3, 2014

Operational Systems Track

Network Virtualization in Multi-tenant Datacenters 203
Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, and Rajiv Ramanathan, *VMware;* Scott Shenker, *International Computer Science Institute and the University of California, Berkeley;* Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang, *VMware*

Operational Experiences with Disk Imaging in a Multi-Tenant Datacenter 217
Kevin Atkinson, Gary Wong, and Robert Ricci, *University of Utah*

VPN Gate: A Volunteer-Organized Public VPN Relay System with Blocking Resistance for Bypassing Government Censorship Firewalls 229
Daiyuu Nobori and Yasushi Shinjo, *University of Tsukuba*

Data Storage and Analytics

Bolt: Data Management for Connected Homes 243
Trinabh Gupta, *The University of Texas at Austin;* Rayman Preet Singh, *University of Waterloo;* Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan, *Microsoft Research*

Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. 257
James Mickens, Edmund B. Nightingale, and Jeremy Elson, *Microsoft Research;* Bin Fan, *Carnegie Mellon University;* Asim Kadav and Vijay Chidambaram, *University of Wisconsin—Madison;* Osama Khan, *Johns Hopkins University*

Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. 275
Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman, *Princeton University*

GRASS: Trimming Stragglers in Approximation Analytics 289
Ganesh Ananthanarayanan, *University of California, Berkeley;* Michael Chien-Chun Hung, *University of Southern California;* Xiaoqi Ren, *California Institute of Technology;* Ion Stoica, *University of California, Berkeley;* Adam Wierman, *California Institute of Technology;* Minlan Yu, *University of Southern California*

Interpreting Signals

Bringing Gesture Recognition to All Devices. 303
Bryce Kellogg, Vamsi Talla, and Shyamnath Gollakota, *University of Washington*

3D Tracking via Body Radio Reflections 317
Fadel Adib, Zach Kabelac, Dina Katabi, and Robert C. Miller, *Massachusetts Institute of Technology*

Epsilon: A Visible Light Based Positioning System 331
Liqun Li, *Microsoft Research,* Beijing; Pan Hu, *University of Massachusetts Amherst;* Chunyi Peng, *The Ohio State University;* Guobin Shen, *Microsoft Research, Beijing;* Feng Zhao, *Microsoft Research, Beijing*

Improving Throughput and Latency (at Different Layers)

Enabling Bit-by-Bit Backscatter Communication in Severe Energy Harvesting Environments 345
Pengyu Zhang and Deepak Ganesan, *University of Massachusetts Amherst*

Full Duplex MIMO Radios 359
Dinesh Bharadia and Sachin Katti, *Stanford University*

Recursively Cautious Congestion Control 373
Radhika Mittal, Justine Sherry, and Sylvia Ratnasamy, *University of California, Berkeley;* Scott Shenker, *University of California, Berkeley and International Computer Science Institute*

How Speedy is SPDY?. 387
Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall, *University of Washington*

Friday, April 4, 2014

In-Memory Computing and Caching

FaRM: Fast Remote Memory 401
Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro, *Microsoft Research*

Easy Freshness with Pequod Cache Joins 415
Bryan Kate, Eddie Kohler, and Michael S. Kester, *Harvard University;* Neha Narula, Yandong Mao, and Robert Morris, *MIT/CSAIL*

MICA: A Holistic Approach to Fast In-Memory Key-Value Storage 429
Hyeontaek Lim, *Carnegie Mellon University;* Dongsu Han, *Korea Advanced Institute of Science and Technology (KAIST);* David G. Andersen, *Carnegie Mellon University;* Michael Kaminsky, *Intel Labs*

Scalable Networking

NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms 445
Jinho Hwang, *The George Washington University;* K. K. Ramakrishnan, *Rutgers University;* Timothy Wood, *The George Washington University*

ClickOS and the Art of Network Function Virtualization 459
Joao Martins and Mohamed Ahmed, *NEC Europe Ltd.;* Costin Raiciu and Vladimir Olteanu, *University Politehnica of Bucharest;* Michio Honda, Roberto Bifulco, and Felipe Huici, *NEC Europe Ltd.*

SENIC: Scalable NIC for End-Host Rate Limiting 475
Sivasankar Radhakrishnan, *University of California, San Diego;* Yilong Geng and Vimalkumar Jeyakumar, *Stanford University;* Abdul Kabbani, *Google Inc.;* George Porter, *University of California, San Diego;* Amin Vahdat, *Google Inc. and University of California, San Diego*

mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems 489
EunYoung Jeong, Shinae Woo, Muhammad Jamshed, and Haewon Jeong, *Korea Advanced Institute of Science and Technology (KAIST);* Sunghwan Ihm, *Princeton University;* Dongsu Han and KyoungSoo Park, *Korea Advanced Institute of Science and Technology (KAIST)*

(Friday, April 4, continues on p. vi)

New Programming Abstractions

Warranties for Faster Strong Consistency 503
Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers, *Cornell University*

Tierless Programming and Reasoning for Software-Defined Networks. 519
Tim Nelson, Andrew D. Ferguson, Michael J.G. Scheer, and Shriram Krishnamurthi, *Brown University*

Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags 533
Seyed Kaveh Fayazbakhsh, *Carnegie Mellon University;* Luis Chiang, *Deutsche Telekom Labs;* Vyas Sekar, *Carnegie Mellon University;* Minlan Yu, *University of Southern California;* Jeffrey C. Mogul, *Google*

Message from the 11th USENIX Symposium on Networked Systems Design and Implementation Program Co-Chairs

Welcome to NSDI '14!

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 spans the gamut from novel wireless technologies to debugging complex systems to data analytics.

We are particularly excited about the Operation Systems Track. It is new this year and is intended for a different breed of papers. Rather than pure research results, these papers describe the design and experience with large-scale, operational systems and networks. They offer a behind-the-scenes look at real networked and distributed systems, which is otherwise hard to come by. This year's session includes papers that describe the design of VMWare's network virtualization system, experience with disk imaging in a multi-tenant data center, and design and experience with operating a VPN-based peer-to-peer system to bypass government censors.

We received a record number of 223 submissions (not counting those that violated submission guidelines). This number represents a sharp increase compared to last year (170) and the previous record (175 in 2010). We accepted 38 papers, a number that is identical to last year. Our program committee had 34 members, including the co-chairs, with diverse expertise and experiences.

The review process included two rounds of reviews, plenty of online discussion, and an in-person PC meeting. In the first round, each paper received three independent reviews. Erring on the side of inclusion, we advanced to the second round any paper with at least one positive review. 142 papers made the cut. In the second round, each paper received at least two additional reviews. Based on the reviews and online discussions, 76 papers were selected for discussion at the PC meeting. We sought external reviews sparingly, mostly in cases where the PC did not have sufficient expertise. Consequently, the PC members bore a significant reviewing load, with 27.4 papers on average.

A conference like NSDI cannot succeed without the collective effort and support of many individuals and organizations. This effort starts with the authors, and we thank them for submitting the product of their hard work. Our PC members were heroic in the face of the sharp jump in the number of submissions, and we are grateful for their reviews, online discussions, and meeting participation. Special thanks to Nikolaos Laoutaris for managing papers with which both chairs were conflicted. We are also grateful to our external reviewers for lending their expertise, often on short notice. We thank Eugene Ng and Amar Phanishayee for serving as Poster and Demo chairs. USENIX does a remarkable job of managing all non-technical aspects of the conference. USENIX does a remarkable job at managing all non-technical aspects of the conference. Working with their staff, including Garrett Johnson, Casey Henderson, and Michele Nelson, was a pleasure. They were always accommodating toward the many special-case requests we made. Kirstie Magness of Microsoft Research helped organize the PC meeting.

Finally, we thank you—the NSDI '14 attendees. It is your participation and interest that sustains and nourishes the conference and our community.

Ratul Mahajan, *Microsoft Research*
Ion Stoica, *University of California at Berkeley*
NSDI '14 Program Co-Chairs

Circuit Switching Under the Radar with REACToR

He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari
Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter

University of California, San Diego

Abstract

The potential advantages of optics at high link speeds have led to significant interest in deploying optical switching technology in data-center networks. Initial efforts have focused on hybrid approaches that rely on millisecond-scale circuit switching in the core of the network, while maintaining the flexibility of electrical packet switching at the edge. Recent demonstrations of microsecond-scale optical circuit switches motivate considering circuit switching for more dynamic traffic such as that generated from a top-of-rack (ToR) switch. Based on these technology trends, we propose a prototype hybrid ToR, called REACToR, which utilizes a combination of packet switching and circuit switching to appear to end-hosts as a packet-switched ToR.

In this paper, we describe a prototype REACToR control plane which synchronizes end host transmissions with end-to-end circuit assignments. This control plane can react to rapid, bursty changes in the traffic from end hosts on a time scale of 100s of microseconds, several orders of magnitude faster than previous hybrid approaches. Using the experimental data from a system of eight end hosts, we calibrate a hybrid network simulator and use this simulator to predict the performance of larger-scale hybrid networks.

1 Introduction

Designing scalable, cost-effective, packet-switched interconnects that can support the traffic demands found in modern data centers is already an extremely challenging problem that is only getting harder as per-server link rates move from 10 to 40 to 100 Gb/s and beyond. In this paper, we focus particularly on the challenge of upgrading an existing network fabric that supports 10-Gb/s end hosts to a network that can deliver 100 Gb/s to each end host. We argue that this transition is inevitable because the PCIe Gen3 bus found in many current servers can support 128 Gb/s, making emerging 100-Gb/s NICs a drop-in upgrade for existing hardware.

Unlike previous generational upgrades, moving from 10- to 100-Gb/s link rates requires a fundamental transition in the way a data center is wired. At 100 Gb/s, inexpensive copper cabling can no longer be used at distances greater than a few meters: virtually all cables other than those internal to an individual rack must be optical. If these cables interconnect electronic packet switches, they further require optoelectronic transceivers at both ends. Many popular packet-switched data-center topologies like multi-rooted trees [25] require large numbers of connections between racks. Hence, the cost of these designs begins to be dominated not by the constituent packet switches, but instead by the transceivers mandated by the optical interconnects necessary to support the increased link speed [10].

In contrast, if the switches internal to the network fabric are themselves optical, the need for transceivers can be significantly reduced. Researchers have previously proposed hybrid architectures consisting of a combination of packet switches and optical circuit switches managed by a common logical control plane [7, 10, 26]. Traditionally, however, their applicability has been limited by the delay incurred when reconfiguring the circuit switches, as traffic has to be buffered while waiting for a circuit assignment. Architectures based upon legacy optical circuit switches designed for wire-area applications are fundamentally dependent on stable, aggregated traffic to amortize their long reconfiguration delays. Therefore, their use has been restricted to either the core of the network [10] or to highly constrained workloads [26].

Researchers have recently demonstrated optical circuit switch prototypes with microsecond-scale reconfiguration delays [6, 17, 19]. In prior work, we showed that such a switch, when coupled with an appropriate control plane [23], has the potential to support more dynamic traffic patterns, potentially extending the applicability of circuit switches to cover the entire network fabric required to interconnect racks of servers within a data center. Circuit switching alone, however, incurs substantial delays in order to achieve efficiency (e.g., 61–300 μs to deliver 65–95% of the bandwidth of a comparable packet switch in the case of our switch [23]), rendering it inadequate to meet the demands of latency sensitive traffic within a data center. Moreover, the buffering required to tolerate such delays with large port counts at 100 Gb/s is substantial. By integrating a certain level of packet switching, hybrid fabrics have the potential to address

these shortcomings. Existing hybrid designs, however, are not capable of coping with the lack of traffic stability and aggregation present at the rack level of today's data centers [2, 4, 15, 16].

In this paper, we propose a hybrid network architecture in which optical circuit switching penetrates the data center network to the top-of-rack (ToR) switch. We leverage our recent work on the Mordia optical circuit switch to build and experimentally prototype the first hybrid network control plane that uses rapidly reconfigurable optical circuit switches to potentially provide packet-switch-like performance at substantially lower cost than an entirely packet-switched network. Our hybrid network design consists of a 100-Gb/s optical circuit-switched network deployed alongside a pre-existing 10-Gb/s electrical packet-switched network. In this model, ToR switches support 100-Gb/s downlinks to servers, and are "dual-homed" to a legacy 10-Gb/s electrical packet switched network (EPS) and a new 100-Gb/s optical circuit switched network (OCS).

REACToR's design is based on two key insights. The first is that it is impractical to buffer incoming traffic bursts from each end host within the ToR's switch memory. For a traditional in-switch time-division, multiple-access (TMDA) queueing discipline, this architecture would require a dedicated input buffer for each potential circuit destination. Given the unpredictable nature of the end-host network stack [16], these buffers would likely need to be quite large.

Instead, REACToR buffers bursts of packets in low-cost end-host DRAM memory until a circuit is provisioned, at which point the control plane explicitly requests the appropriate burst from each end host using a synchronous signaling protocol that ensures that the instantaneous offered load matches the current switch configuration. Because each REACToR is dual-homed to an EPS, the control plane can simultaneously schedule the latency-sensitive traffic over the packet switch. The packet switch can also service unexpected demand due to errors in demand estimation or circuit scheduling.

The second insight is that if circuit switching is sufficiently fast, then delays due using flow-level circuit-switched TDMA at the end-host network stack will not degrade the performance of higher-level packet-based protocols; in a sense the circuit switch will "fly under the radar" of these end-host transport protocols. As technology trends enable faster OCS reconfiguration times, this hybrid architecture blurs the distinction between packets and circuits. By combining the strengths of each switching technology, a hybrid network can deliver higher performance at lower cost than either technology alone, even at the level of a ToR switch.

We evaluate our design for a 100/10-Gb/s OCS/EPS hybrid network using a scaled-down 10/1-Gb/s hardware prototype that supports eight end hosts. The prototype consists of two FPGA-based REACToRs with four downward-facing 10-Gb/s ports each. Both REACToRs connect to the Mordia [23] microsecond OCS and a commodity electrical packet switch. The circuit switch supports a line rate of 10 Gb/s while the packet switch is rate limited to 1 Gb/s to enforce a 10:1 speed ratio. End hosts connect to our prototype using commodity Intel 10-Gb/s Ethernet NICs that we synchronize using standard 802.1Qbb PFC signaling.

Link rate	Full fat tree	Helios-like	REACToR
10 Gb/s	2 – 4	1 – 3	N/A
100 Gb/s	4	3	1†

Table 1: Number of transceivers required *per upward-facing ToR port* for different network architectures. (†Presuming a 10-Gb/s packet network is already in place.)

Our experiments show that our REACToR prototype can provide packet-switch-like performance by delivering efficient link utilization while reacting to changes in traffic demand, and that its control plane is sufficiently fast that changes in circuit assignment and schedule can be made without disrupting higher-level transport protocols like TCP. Using simulation of more hosts, we also illustrate the large benefits that a small underprovisioned packet switch provides to a hybrid ToR relative to a pure circuit ToR. We conclude that REACToR can service published data-center demands with available technology, and can easily scale up to make effective use of next-generation optical switches and 100-Gb/s hosts by reusing an existing 10-Gb/s electrical packet-switched network fabric.

2 Background

We start by motivating the benefits of a hybrid-ToR network design, describing the work that REACToR builds upon, and then discussing our design assumptions.

2.1 Motivation

Consider a data-center operator that wants to upgrade an existing 10-Gb/s data center network—i.e., the part of the network that connects the top-of-rack switches together—to 100 Gb/s.

Table 1 shows the number of optical transceivers required for each upward-looking port of the ToR for three different network architectures. The first architecture is a fully provisioned three-level fat-tree network [1]. If all of the links in the backbone network are optical, then this network requires four transceivers per upward port. In a Helios-like [10] architecture an optical circuit switch is placed at the uppermost layer of the network, saving one transceiver per port as compared to the number used in a fat-tree network.

At 10 Gb/s, if the links between aggregation switches are short enough to be electrical, then transceivers may only be required between aggregation and core switches, potentially reducing the number of transceivers by up to three per port. At 100 Gb/s, however, while electrical interconnects may still be viable from an end host to a ToR (i.e., distances less than 5 meters), all connections from the ToR to the rest of the network are likely to be optical. Hence, either architecture will require a full compliment of optical transceivers. Moreover, in order to upgrade the network the operator will have to replace the existing 10-Gb/s transceivers with new 100 Gb/s transceivers.

The REACToR architecture, in contrast, re-uses the existing 10-Gb/s packet-based network and deploys a parallel 100-Gb/s circuit-switched optical network under a common control plane. As compared to the other two architectures listed in Table 1, REACToR requires only one 100-Gb/s transceiver per upward-facing port of the ToR because the OCS does not use transceivers. This means that for a fully provisioned three-level fat-tree network, if the per-port cost of the OCS used in REACToR is less than three times the cost of a 100-Gb/s optical *transceiver*, then a REACToR hybrid network will cost less than an equivalent 100-Gb/s packet-based network—even if the 100 Gb/s *switches* themselves were free. Larger networks require even more transceivers per end host: a five-level network requires eight transceivers to support each upward facing port, making the economics of REACToR even more compelling. Over-subscribed networks will use fewer transceivers in the core network, but the scaling trends are still applicable.

While this example uses a 10-Gb/s EPS and a 100-Gb/s OCS, the actual link rates for which a REACToR architecture will be cost competitive with a fully provisioned or over-subscribed packet-switched network depends on market trends. Many OCS architectures are based on MEMs devices and can easily support link rates in excess of 100 Tb/s[1] per port. For this kind of device, the cost per optically switched bit is decreasing and is fundamentally inversely proportional to link rate. While the costs per switched bit of optical transceivers and packet switches are also decreasing, the rate of decrease is much slower. These trends imply the cost per switched bit will eventually become comparable at some link rate. What is less clear is the precise link rate when this crossover point will occur and the economic viability of a data-center network that supported such a link rate.

2.2 Related work

Hybrid data-center network architecture design is an active research area. Helios [10] and c-Through [26] both rely on slower 3D-MEMs based optical circuit switching, restricting their use to either highly aggregated traffic (i.e., in the core of the network), or highly stable traffic (e.g., long file transfers). OSA [7] combines an OCS-based reconfigurable topology with multi-hop overlay networking. The bandwidths of links in OSA can be varied through the use of wavelength-selective switching. In addition to optical switching, reconfigurable wireless links have also been proposed in data-center contexts [12, 14, 28]. In contrast to these previous approaches, which employ switching technologies with relatively long reconfiguration times, REACToR relies upon the Mordia [8, 23] OCS, which can be reconfigured in 10s of microseconds, in order to service a much larger portion of the offered demand through the circuit-switched portion of the hybrid network fabric.

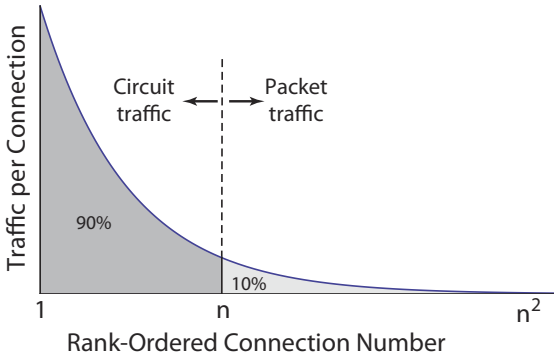


Figure 1: Rank-ordered traffic for each of the n^2 elements of a demand matrix, for which most of the traffic (e.g. 90%) is carried in a few ($O(n)$) flows.

The question of how much buffering should be deployed in a network has been considered under a wide variety of settings. In the Internet, a common rule of thumb has been that at least a delay-bandwidth product is necessary to support TCP effectively. Appenzeller *et al.* [3] challenged this assumption for core switches, and argue that for links carrying many TCP flows, less buffering is necessary. In the data center, Alizadeh *et al.* propose modifications to TCP that, along with appropriate switch support, can reduce the amount of buffering required down to a single packet per flow [2]. Other network technologies have also been created that reduce in-network buffering, including Myrinet [5] and ATM [20]. Numerous proposals for entirely bufferless "network-on-chip" (NoC) networks have been proposed [21], including hybrid NoC networks that also leverage packet switches [13].

2.3 Design assumptions

Studies of data-center traffic show that the traffic demand inside a data center is frequently concentrated, with a large fraction of the traffic at each switch port of a ToR

[1]The mirrors are typically reflective from approximately 1.3 μm to 1.6 μm which corresponds to a bandwidth of approximately 400 THz.

destined to a small number of output ports [15]. Such locality is not surprising, as application programmers and workload managers frequently use knowledge about the location of end hosts to coordinate workloads to minimize inter-rack traffic. Based on these empirical observations, a fundamental premise of all hybrid networks—including REACToR—is that a large fraction of the network traffic is carried by a small number of relatively long-lived flows. This observation can be expressed in terms of the n^2 rank-ordered elements of a demand matrix for a network that connects n ToRs. Figure 1 shows an example where 90% of the inter-ToR traffic is carried by only n flows.

In such settings, the demand matrix is frequently both sparse and stable [9, 12, 14]. This kind of traffic demand is generally suitable for a large port-count optical circuit switch, but these assumptions can be violated for specific workflows over any given time interval. Therefore, REACToR switches the few high-traffic flows using an OCS while forwarding the relatively small amount of traffic between most ToRs using an under-provisioned standard packet switch.

While rack-level coordination can lead to bursty traffic at the upward looking ports of a ToR, we carry this assumption one step further. Unlike previous hybrid designs that focus on the core of the network, REACToR critically depends upon *individual hosts* being able to fill circuits assigned to them with data, which in turn depends on hosts transmitting groups of packets to the same destination ToR at fine time scales.

To verify this assumption, in previous work we measured individual flows, at microsecond granularity, emanating from a single host under a variety of workloads [16]. We find that host mechanisms such as TCP segmentation offloading in the NIC and generalized segmentation offloading in the operating system, cause traffic to frequently leave the NIC in bursts of 10s to 100s of microseconds. In Section 5.1, we expand upon this analysis to show that circuit switching these flows can further enhance this behavior while not disturbing the transport protocol. For regimes in which circuit switching does not affect the transport performance of an end host, we say that its flows are "flying under the radar".

3 Design

A REACToR-enabled data center consists of N servers grouped into R racks, each consisting of n nodes. We assume that a preexisting 10-Gb/s packet-switched network is already deployed within the data center. Overlaid on top of this packet-switched network is an additional 100-Gb/s circuit-switched network. At each rack is a hybrid ToR called a REACToR, which is connected to the packet-switched network with $u_p \leq n$ uplinks and is connected to the circuit-switched network with a separate set of $u_c \leq n$ uplinks. This means that the packet-switched network supports $R \times u_p$ ports, and the circuit-switched network supports $R \times u_c$ ports. Each REACToR has n downward-facing 100-Gb/s ports to its n local servers. In this work, we consider the fully provisioned case where $u_p = u_c = n$; however, additional cost savings are possible when either or both of u_p and u_c are less than n. Our architecture is agnostic to the particular technology used to build the circuit-switched fabric, but, given technology trends, we presume it is optical.

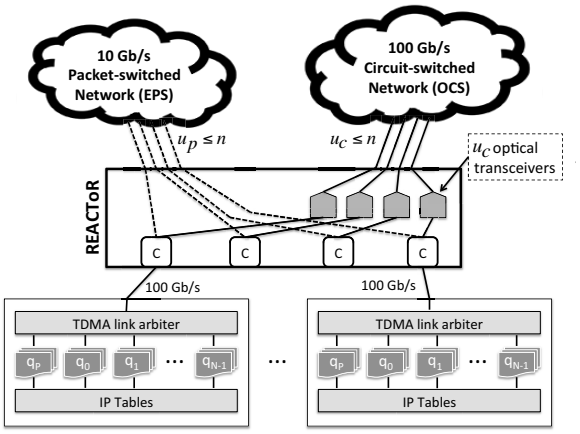


Figure 2: 100-Gb/s hosts connect to REACToRs, which are in turn dual-homed to a 10-Gb/s packet-switched network and a 100-Gb/s circuit-switched optical network.

Referring to Figure 2, an (n, u_p, u_c)-port REACToR consists of n downward-facing ports connected to servers at 100 Gb/s, $u_p = n$ uplinks connected to the packet-switched network at 10 Gb/s, and $u_c = n$ uplinks connected to the 100-Gb/s circuit-switched network. At each of the n server-facing input ports, there is a classifier (labeled 'C' in the figure) which directs incoming packets to one of three destinations: to packet uplinks, to circuit uplinks, or through an interconnect fabric to downward-facing ports to which the other rack-local servers are attached. There is no buffering on the path to the packet uplinks, as buffering is provided within the packet switches themselves. There is also no buffering on the path to the circuit uplinks; instead, packets are buffered in the end-host where they originate. When a circuit is established from the REACToR to a given destination, the REACToR explicitly pulls the appropriate packets from the attached end-host and forwards them to the destination.

REACToR relies upon a control protocol to interact with each of its n local end-hosts to: (1) direct the end host to start or stop draining traffic from its output queues (which we refer to as *unpausing* or *pausing* the queue, respectfully), (2) set per-queue rate limits, (3) provide circuit schedules to the end-host, and (4) retrieve demand

estimates for use in computing future circuit schedules. We first motivate the need for this functionality by describing the various other aspects of REACToR's design before detailing the host control protocol in Section 3.4.

3.1 End-host buffering

Each end-host buffers packets destined to the REACToR in its local memory, which is organized into traffic classes, one per destination ToR, with an additional class for packets specifically destined for the EPS (e.g., latency-sensitive requests). Each traffic class has its own dedicated output queue (i.e., $\{Q_0, Q_1, ..., Q_{N-1}\}$), with an additional queue for the EPS class, Q_P, as shown in Figure 2. At any moment in time, the REACToR can ask an end host to send packets from at most two classes: one forwarded at line rate to an OCS uplink (or local downlink port), and another forwarded to an EPS uplink. This latter class of traffic must be rate limited at the source NIC to conform to the link speed of the EPS to prevent overdriving the EPS. In the reverse direction, the EPS may emit packets into the REACToR at its full rate to a particular downward-facing port. Because that downward-facing port could potentially be shared by incoming line-rate circuit traffic heading to the same destination, REACToR must further ensure that the circuit traffic is sufficiently rate-limited so that there is enough excess capacity to multiplex both flows at the destination. Hence, end hosts will be directed by REACToR to similarly rate-limit traffic classes destined to the OCS at the source NIC, but at much higher rates. Further details on rate limiting are provided in Section 3.3.

Today, end-host NICs support modest amounts of buffering, on the order of a few megabytes. However, it is not organized in a way that can be directly used to support circuits. NICs partition their buffers into a small set of 8 to 64 transmit queues, which the OS uses to batch and store packets waiting to be sent. The scheduling policy for these queues is typically built into the NIC (e.g., round robin), so the actual transmit time of individual packets is outside the control of the OS.

To achieve high circuit utilization in REACToR, the NIC needs the ability to send data for a particular circuit destination to the ToR as soon as a circuit becomes established, and to fill that circuit continuously until it is torn down. At any one time, each circuit uplink within a REACToR is exclusive to a particular source port (attached end host), so efficiency degrades any time that source has no data to send. Thus, packets headed to the same circuit destination (i.e., remote host) should be grouped together within a host's memory, so that when a circuit to that destination becomes available, that group of packets can be sent from the NIC to the REACToR at line rate.

Within each host, we define a traffic class per destination host, and task the OS with classifying outgoing packets into the appropriate class based on, e.g., the destination IP address. REACToR then uses the host control protocol to pause and unpause end-host queues. In this model, the role of the OS and of the NIC changes somewhat: rather than the OS "pushing" packets to the NIC buffers based on queuing policies in the host, the NIC is responsible for "pulling" packets from the host memory into the NIC buffers according to the circuit schedule just in time to transmit them to the connected circuit. (We note that the NIC design advocated by Radhakrishnan et al. [24] would be especially well suited for this model.)

Demand estimation. Over a short time scale (i.e., 100s of μs, depending on the size of the NIC buffers), the occupancy of these traffic classes defines the imminent end-host demand because the packets in these queues have already been committed to the network by the OS. It is possible to query the OS, the application, or even a cluster-wide job scheduler to form longer-term demand estimates. For example, Wang et al. [26] use TCP send buffer sizes as estimates of future demand. Our prototype uses a demand oracle. In any case, the circuit scheduler uses these demand estimates to determine a set of future OCS circuit configurations.

3.2 Circuit scheduling

To make effective use of the capacity of the circuit switch, REACToR must determine an appropriate schedule of circuit switch configurations to service the estimated demand over an accumulation period W. This is the responsibility of a logically centralized, but potentially physically distributed, circuit scheduling service, which implements a hybrid circuit scheduling algorithm. This service collects estimates of network-wide demand, in the form of an $N \times N$ matrix D. The service computes a schedule, P_k, of m circuit switch configurations, which are permutation matrices[2], and corresponding durations, ϕ_k.

The number m of configurations that comprise the schedule is constrained because each circuit configuration requires a finite reconfiguration time δ, during which time no data can be forwarded over the circuit switch. When δ is large with respect to W, it is more efficient to use fewer configurations. When δ is small with respect to W, more configurations can be used. Including this reconfiguration delay, the duration of the schedule is constrained by the length of the accumulation period so that $\sum_{k=1}^{m} \phi_k + \delta m \leq W$. The goal of the scheduling algorithm is to maximize $\sum_{k=1}^{m} \phi_k$ subject to these constraints.

Obviously, if the switch introduces a reconfiguration delay, then it is impossible to service fully saturated demand at line rate. Existing research in constrained scheduling has focused on switches that run

[2] A permutation matrix is a matrix of 0s and 1s in which each row and column has and only has a single 1.

faster than the link rate, with the ratio of the switch rate to the link rate called the speedup factor. These algorithms [11, 18, 27] produce a variable-length schedule which is dependent on the actual reconfiguration delay.

Hybrid networks in general, and REACToR in particular, do not use a speedup factor. Instead, REACToR uses the lower-speed packet switch as a way to make up for the reconfiguration delay and any scheduling inefficiency. This "back channel" is a key distinction between REACToR and traditional blocking circuit scheduling because REACToR continues to service a subset of flows over the EPS when circuits are not available, thereby increasing support for latency sensitive workloads.

We leave the selection and evaluation of an circuit switch algorithm as future work; for now we compute the schedule offline using a variant of existing constrained switching algorithms based on a predetermined demand matrix D. Any schedule computed for use in REACToR, however, is subject to a number of constraints.

Class constraints. In order to ensure the offered load can be effectively serviced by the ToR, REACToR imposes a number of constraints on the set of queues that can be unpaused at any particular time. First, the dedicated EPS queue (Q_P) is always unpaused but rate-limited to at most 10 Gb/s, providing the host with the ability to send latency-sensitive traffic directly to the EPS at any point in time. Second, at most one additional queue can be unpaused at any one time for transmission at (near) link rate (i.e., 100 Gb/s). When such circuit-bound (or rack-local) traffic arrives at an input classifier in the REACToR, it is directly forwarded to the appropriate circuit uplink (or downward-facing port) without any intermediate buffering. The third constraint is that, if a queue is unpaused for link-rate transmission in the current scheduling period, then it should never be unpaused for transmission to the EPS. This constraint serves two purposes: it prevents the EPS from being burdened with high-bandwidth traffic better served by circuits, and it gives that traffic class additional time to accumulate demand so that the circuits are highly utilized.

Fourth, any traffic class which is not assigned to a circuit (or downward port) during a scheduling period is instead remapped to the EPS, meaning that any packets in that class's queue are routed to the EPS uplink. Finally, all of the queues corresponding to EPS-bound traffic (i.e., both the dedicated EPS queue and and any classes not scheduled for a circuit in this period) must be rate limited such that the sum of their limits is less than or equal to the EPS link rate (e.g., 10 Gb/s).

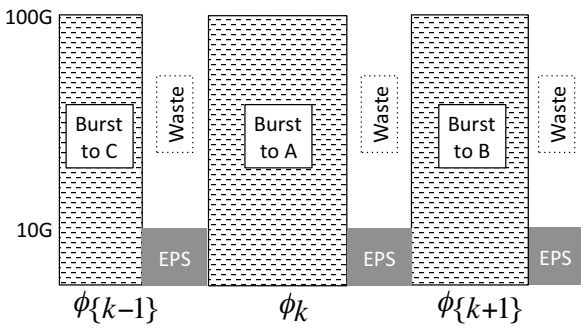
3.3 End-host rate limiting

At any given time, each of the REACToR's downward-facing server ports can transmit data from two sources: a circuit from a single source established through the OCS (or rack-local connection) fabric, and traffic from any number of sources forwarded through the EPS. At each downward-facing port there is a multiplexer which performs this mixing. When the sum of bandwidth from the EPS (B_{EPS}) and OCS (B_{OCS}) exceeds the rate of the REACToR port (B_{ToR}), then without intervention, $(B_{\text{EPS}} + B_{\text{OCS}}) - B_{\text{ToR}}$ traffic would be dropped. To prevent such drops, and to ensure high overall utilization, we rely on end-host rate limiting.

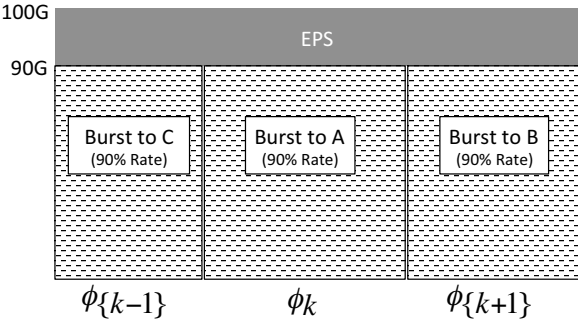
The first way that we use end-host rate limiting is to ensure that in steady state, $B_{\text{EPS}} + B_{\text{OCS}} \leq B_{\text{TOR}}$. Since the OCS is bufferless, the multiplexer gives priority to packets arriving from the OCS because otherwise they would have to be dropped. Assuming a REACToR with a 100-Gb/s OCS and 10-Gb/s EPS as an example, each end host would rate limit its circuit-bound traffic in the range of 90–100% of the link capacity to leave sufficient head room for the EPS traffic, based on the estimate of EPS demand in the current schedule. Each time that a set of configurations for a scheduling period is computed, a rate limit is also computed per configuration, reflecting the estimated load from the EPS. Note that this estimate need not be perfect, and in fact we expect the EPS to absorb inaccuracies in scheduling, demand estimation, and rate limiting. For each scheduling interval, the associated rate limits are computed and sent to each end-host via the host control protocol.

The circuit rate limit also serves a second purpose, which is providing statistical multiplexing at the downward-facing REACToR port. Underpinning the design of REACToR is the assumption that on short time scales, traffic emanating to a single destination is bursty. Each burst by definition consists of a number of packets sent back-to-back. From the point of view of the REACToR port multiplexer, this means that, absent other controls, during the first portion of a given circuit-switch configuration interval ϕ_k, the entire port's bandwidth would be dedicated to servicing a single burst of traffic from the OCS. Thus, any packets originating from the EPS would be delayed until the end of ϕ_k. Figure 3(a) shows a pictorial representation of this behavior. The challenge that arises is that the line rate of the EPS is presumed to be lower than the REACToR port speed and the OCS. Hence, the open region at the end of ϕ_k can only be filled with packets at the rate of the EPS (e.g., 10 Gb/s) instead of the OCS (100 Gb/s). Thus, for this example, the region at the end of ϕ_k only gets 10% utilized since the EPS can only drive 10% of the outgoing port bandwidth.

Instead, REACToR seeks to ensure that the circuit traffic is spread out across ϕ_k by limiting it to less than full line rate (e.g., 90 Gb/s of a 100-Gb/s link). Rate limiting over time allows the EPS-serviced traffic to be multiplexed on REACToR's downward-facing ports at a uni-



(a) When a circuit-switch configuration interval ϕ_k begins, queued traffic forms bursts which saturate the link during the first part of the configuration, leaving capacity for EPS traffic at the end of ϕ_k; since the EPS runs at a fraction of the line rate, it cannot efficiently use the remaining time.



(b) By rate limiting circuit traffic, the EPS can spread its traffic out over the entire configuration interval.

Figure 3: Rate limiting prevents bursts from the OCS from starving the EPS, which would otherwise be unable to make full use of each circuit-switch configuration interval ϕ_k. In both cases, the circuit-switched traffic achieves 90 Gb/s during each interval.

form rate across all configuration intervals ϕ_k, enabling the entire interval to be utilized by both circuit traffic and packet traffic. By setting circuit rate limits in the end host, as described above, the traffic headed to the circuit is paced to the appropriate rate. Figure 3(b) shows the resulting treatment of circuit and packet data within that same configuration interval ϕ_k.

3.4 REACToR host control protocol

An instance of the REACToR host control protocol runs between each end-host and its REACToR switch. REACToR uses the protocol to retrieve demand estimates collected by end-hosts, to set per-queue rate limits, as described above, and to convey impending schedules to the end host from the circuit scheduler. These functions are relatively straight forward. In this section, we examine the fourth use of the host control protocol: managing end-host traffic classes and buffering. The key to achieving efficient use of the hybrid network is being able to drain the appropriate classes with fine-grained precision at the right times. We now describe the host control protocol that achieves this precision.

Overview: To ensure reliable transmission, we cannot reconfigure the OCS until all incoming circuit traffic has ceased, since the OCS is unable to carry traffic during the time δ when it is being reconfigured. While classifiers on each REACToR input port can shunt all traffic to the EPS nearly instantaneously, in general we would like to ensure that almost all circuit-bound traffic has been paused before reconfiguring the OCS. Otherwise, a massive queue would build up at the EPS at the end of each schedule. To avoid this buildup, we leverage the 802.1Qbb Priority Flow Control (PFC) protocol to pause traffic at the end host. Each traffic class in the end host corresponds to a PFC class.[3] At the end of each schedule, for each attached host, the REACToR first sends a PFC frame to pause the traffic class destined for the current schedule's circuit (if any). Note that PFC frames are selective, so traffic destined to the EPS will continue to flow while the OCS is being reconfigured. Once inbound circuit traffic has ceased, the OCS can be reconfigured. After reconfiguration, the traffic class corresponding to the next schedule's circuit can be enabled by a PFC unpause frame.

Performance: The overall speed of the control plane is bounded by the speed at which REACToR can pause and unpause traffic classes buffered at the end hosts. Because the PFC frame must be both received and processed at the NIC before traffic stops, there will be some delay between when the controller wants to pause traffic and when the traffic finally stops arriving at the incoming ports at the REACToR. To quantify this delay, we extended the classifiers on our prototype to timestamp all incoming packets and mirror these timestamped packets to a collection host. We then measured the time from when the classifier sends a PFC frame to a host until it stops receiving packets from that host.

We measured the minimum (maximum) delay on an Intel 82599-based 10 Gb/s NIC as 1,014 (2,188) ns, with the actual delay varying as a function of PFC offset, meaning that if the PFC frame arrives more than 185.6 ns after the start of the current frame, the NIC will generate an additional frame before pausing, likely due to pipelining within the NIC implementation.

Once the OCS has established a circuit and is ready to receive traffic, the REACToR needs to restart traffic for the newly connected destination by sending another PFC frame. The measured 'on' delay (i.e., from when the configuration is started by the transmission of a PFC frame

[3] Although the current PFC specification is limited to eight frame priority levels, it is possible to reuse classes across schedule periods by recoloring.

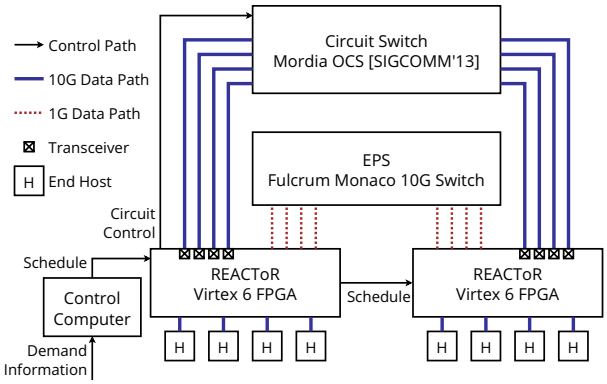


Figure 4: Our prototype REACToR network.

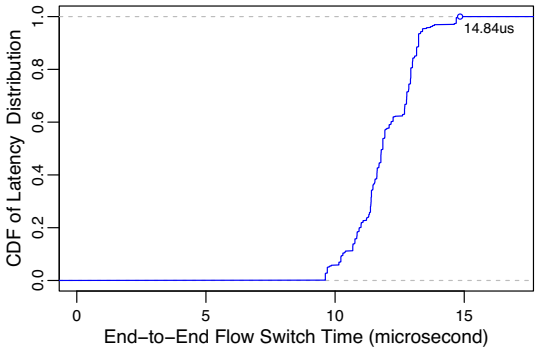


Figure 5: Observed end-to-end circuit switch reconfiguration delay δ.

unpausing the traffic) ranges between 1.2 μs and 1.3 μs. From the 'off' delay measurement, it is clear that we can hide the first microsecond of delay by sending the PFC frame before we actually want the traffic to stop, but it may take an additional 1.3 μs for all ports to cease sending. There is one additional source of delay: a port may be busy sending an outgoing packet at the moment the classifier wishes to send the PFC frame. This delay is bounded by the 1500-byte MTU in our prototype, leading to a worst-case combined delay of approximately 2.5 μs, which is the lower bound of the speed of the control plane achievable in REACToR with 10-Gb/s end hosts, a 1500-byte MTU size, and the 802.1Qbb implementation on our NIC.

4 Implementation

To evaluate our design, we have implemented two prototype four-port 10-Gb/s REACToRs (shown in Figure 4) using two FGPAs, a Fulcrum Monaco 10-Gb/s electrical packet switch, and the Mordia microsecond OCS [23]. Mordia is 24-port reconfigurable OCS built from six 4-port "binary MEMs" wavelength-selective switches, with a reconfiguration delay of $\delta = 12\mu$s, which includes the physical switching time of the MEMs devices and the time to reinitialize the attached 10-Gb/s transceivers. Thus, our REACToR prototype supports the same 10:1 bandwidth ratio described earlier, but at 10 Gb/s (OCS) and 1 Gb/s (EPS) rather than 100/10 Gb/s.

Each REACToR is implemented with a HiTech Global HTG-V6HXT-100GIG-565 FPGA development board, which supports 24 ports of 10-Gb/s I/O. The circuit scheduling service runs as a user-level process on a dedicated Linux-based control server, and transmits schedules to the FPGA via a dedicated 10-Gb/s Ethernet connection. In our implementation, the end hosts are servers equipped with Intel 82599-based NICs. The end hosts classify traffic according to the destination using the Linux `tc` facility. The classifier on the FPGA selectively enables or disables packets to a given destination using the IEEE 802.1Qbb priority-based flow control standard, which supports eight flow classes. We use seven of these classes to correspond to the n circuit destinations reachable from a REACToR, and the eighth is reserved for the EPS-dedicated class.

At each switch reconfiguration, the controller on the FPGA updates the OCS and enables the corresponding end-host traffic classes using 802.1Qbb PFC frames. The controller also configures the classifiers so that they forward the appropriate line-rate flow to the circuit uplink, and forward the remaining traffic to the EPS.

Circuit switch characterization: The average reconfiguration delay for the Mordia switch is approximately 12 μs, with a maximum observed delay of 14.84 μs (as shown in Figure 5). The transceivers we use vary in their "lock" time, necessitating setting a more conservative reconfiguration delay. This variance is an engineering artifact of our hardware and is not fundamental; the IEEE 802.3av (10G-EPON) specification, for instance, calls for a 400-ns lock time. Except as noted, in the experiments that follow, we configure REACToR to assume a 30-μs reconfiguration time which, contained within at least a 160-μs configuration period, delivers at least 81% link efficiency.

REACToR host control protocol: To tightly time synchronize the attached hosts, REACToR sends the schedule to each attached host using two UDP packets. The first packet contains the impending schedule for the upcoming 3-millisecond period, whereas the second packet indicates the start of the three-millisecond time period, serving as a precise periodic heartbeat. End hosts receive these packets in a kernel module via the `netpoll` kernel APIs, which reduces the delay in acting on them to less than 15 μs.

5 Evaluation

In this section, we evaluate the performance of our REACToR prototype implementation. We first show that, with buffering and scheduling packets at end-host NICs, circuit-switching does not negatively impact TCP throughput. Second, we show that the REACToR can dynamically update and switch schedules of many flows without impacting throughput. Third, we show that REACToR can serve a time-varying workload that consists of multiple high- and low-bandwidth flows, promoting flows as appropriate from the packet-switched fabric to the circuit-switched fabric. Finally, we use simulation to illustrate the large benefits that a small underprovisioned packet switch provides to a hybrid ToR.

To generate arbitrary traffic patterns, we implemented a Linux kernel module based on `pktget` [22] that can send MTU-sized UDP packets at arbitrary rates up to line rate. When the module is sending, it runs on a dedicated core and each packet it sends has a sequence number. At the same time, the module also serves as a traffic sink that receives UDP traffic via the `netpoll` kernel interface, and records the sequence number and source address of packets. For packet timing measurements, we configured the FPGA to generate a record for each packet that captures the source, destination, and a timestamp with 6.4-nanosecond precision. The prototype sends these records out-of-band to a collection host using one of the 10 Gb/s ports of the FPGA, which we then process offline.

5.1 TCP under TDMA scheduling

In Section 2.3, we described how application flows exhibit intrinsic short-term correlated bursts as a consequence of the NIC trying to efficiently use the link. We therefore consider how flows behave in a hybrid fabric where a circuit scheduler pauses flows at the host while they wait for an assigned circuit and unpauses them when the circuit is established. While its flow is paused, an application may generate additional packets, increasing the size of its burst when its flow is eventually unpaused and thereby more efficiently use its circuit. However, the increased latency and latency variation induced by pausing and unpausing flows may detrimentally impact the transport protocol (e.g., TCP) or the application itself.

To study the impact of circuit scheduling on TCP throughput, we generate stride workloads where a single host sends to another host, and at the same time sinks a TCP flow from a third host. First we consider the case where we pause and unpause a bi-directional circuit, i.e., pause both data and TCP ACKs at the same time. Next we consider the case where we pause the data in the flow, but allow ACKs to return unimpeded (e.g., via the EPS). Finally, we consider the case where we pause the ACKs, but enable data packets to transmit unimpeded.

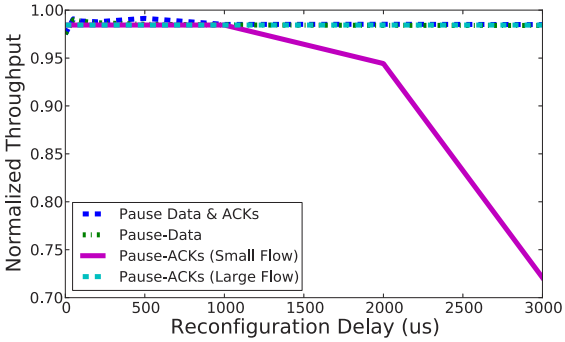


Figure 6: Effect of pausing/unpausing data/ACK packets on TCP throughput.

Figure 6 shows the resulting normalized throughput when varying the reconfiguration delay δ for a stride workload with eight hosts. In the first case, the normalized throughput of uni-directional and bi-directional circuits is close to ideal, showing that pausing data packets on the end hosts does not affect throughput for pause lengths considered by REACToR. When pausing only the ACKs, we find that there are two regimes to consider. During slow start ('Small Flow'), pausing ACKs decreases the overall throughput of the flow—up to 30% for 3-ms delays. For shorter delays (e.g., $\leq$1 ms) there is no detectable effect for pausing ACKs. Once the flow leaves slow start ('Large Flow'), there is no effect on throughput regardless of the reconfiguration delay.

These experiments consider the effect of circuit scheduling on TCP traffic in the absence of packet loss. In practice, packets may be lost for a variety of reasons. We repeated the experiments above where each end host drops packets uniformly at random with a configurable drop probability. While TCP throughput suffers as expected with increasing drop rates, the difference in performance with and without circuit scheduling (e.g., with and without issuing PFC pause frames) is insignificant for steady state loss rates up to 1%.

5.2 Switching "under the radar"

Next we evaluate the speed and flexibility with which REACToR can be reconfigured. We first run an all-to-all workload on eight hosts, where every host streams a TCP flow to each of the other seven hosts using all available bandwidth. To serve this workload, we load REACToR with a schedule of seven TDMA periods that fairly shares the links among all the flows. Each schedule period is 1.5 ms, within which each host sends and receives from each other host for 214.3 μs (including a 30-μs circuit reconfiguration delay) in each circuit configuration. We schedule all data packets via the circuit switch, and all TCP ACKs via the packet switch. We could use the same schedule for every period, but to further exercise

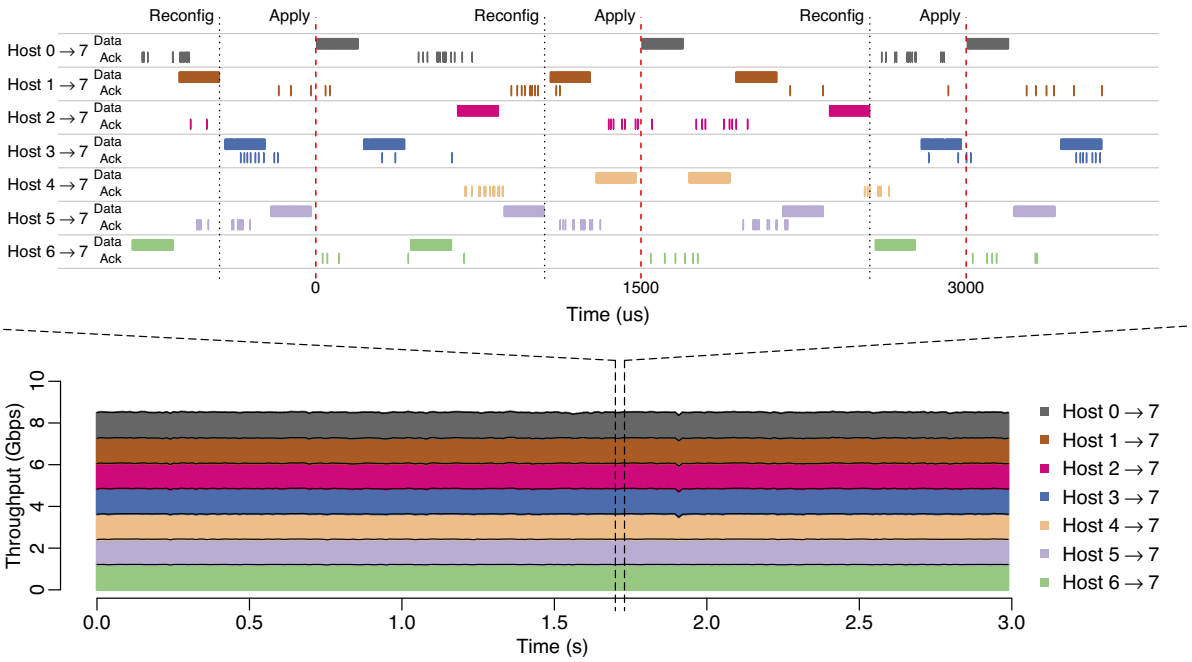


Figure 7: All-to-all workload with circuit configurations changing every scheduling period.

our prototype we change the schedule so that hosts receive circuits in different permutations in each period.

Figure 7 shows three seconds of an all-to-all workload where flows start at the same time on the hosts. The bottom part shows the achieved throughput as reported by one of the hosts: the flows from the other seven hosts evenly split the available bandwidth. Total TCP goodput received is 8.1 Gb/s, the maximum given the 86% duty cycle resulting from the 30-μs reconfiguration delay in a 214.3-μs circuit.

At the application level, the achieved TCP goodput maximizes network capacity and is stable over time. However, if we zoom in and look at the packet traces, as shown in the top part of the figure, we can see the fine-grained behavior of scheduling the flows on circuits. A control packet triggers a new schedule each period, which the controller sends to the REACToR during the previous period (at the time marked 'Reconfig') and the switch loads just before the new period starts ('Apply'). The schedule partitions each period into seven circuit configurations, one for each of the seven hosts sending to the host we are observing.

At time offset zero, for instance, host 0 has the first configuration in the schedule. Its data packets arrive over the circuit it receives, and no other host can send data packets through the circuit switch to host 7. The second configuration schedules host 3, and so on. ACKs received at host 7 use the packet switch, and hence can overlap circuits scheduled for other hosts. (The flow assignments are asymmetric; when host 0 is sending to host 7 at time zero, host 7 is sending to host 6 and receiving ACKs from it.)

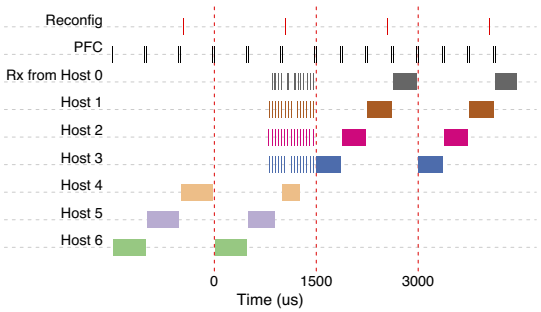


Figure 8: Changing the number and duration of configurations in scheduling periods.

This all-to-all workload does not vary demand over time. Given the frequency with which we can reconfigure the circuit switch, we can also serve time-varying workloads by serving different workload demands under different scheduling times with different numbers of configurations and circuit assignments.

We use another experiment to demonstrate this flexibility. We divide the eight hosts into two groups: G_A consists of hosts 0–3, and G_B hosts 4–7. We then generate traffic among the hosts using two workloads. The first is a group-internal all-to-all, where each host streams TCP packets to the other three hosts in its group at the maximum possible rate. To serve this workload, REACToR uses a schedule that has three configurations in a schedul-

ing period. The period lasts 1,500 μs, and each configuration lasts 500 μs (including a 30-μs reconfiguration delay). The second is a cross-talking all-to-all workload where each host in G_A streams to all the other four hosts in G_B, and vice versa. For this workload, REACToR uses schedules with four configurations. These scheduling periods also last 1,500 μs, but each configuration lasts 375 μs (again including a 30-μs reconfiguration delay).

In the experiment, we change from the group-internal to the cross-talk workloads midway through, loading the REACToR with correspondingly different schedules. Figure 8 shows the incoming packets to host 7 around the workload transition time. We controlled the experiment so that the workload changes at an inconvenient, but more realistic, time for REACToR: *during* a scheduling period, at time 750 μs on the graph. REACToR's schedules commit the switch based on predicted demand, and workloads are apt to change their demand independent of when REACToR can conveniently accommodate them. At this workload transition, REACToR is halfway through its scheduling period and packets already queued at the first three hosts continue to arrive via circuits. Overlapping these flows, the other four hosts start sending packets to host 7. These hosts do not have circuits, so the packets arrive via the EPS at a much lower rate.

At the end of its committed scheduling period (time 1,500 μs), REACToR can then react to the workload transition and schedule circuit configurations that match the workload. At this time, host 7 changes from receiving packets in 500-μs configurations, scheduled round-robin from hosts 4–6, to receiving packets in 375-μs configurations from hosts 0–3.

In summary, these experiments demonstrate the speed and flexibility with which REACToR can reconfigure its circuit schedules given a known demand. Applications achieve their expected goodput at a high level, while individual flows are paused and released at fine time scales when their circuits are scheduled. Further, REACToR can adjust the circuit schedule to adapt to changes in application behavior and demand.

5.3 Time-varying workloads

Next we show that REACToR can dynamically serve rapidly changing traffic demands and efficiently move flows from the EPS to the OCS.

In this experiment, we vary the number of high bandwidth and low bandwidth flows among hosts at small timescales. The workload pattern is again all-to-all among eight hosts, which we observe from the perspective of one of the hosts and its seven incoming flows. Initially one of the flows is a high-bandwidth flow sending at full demand and served on the circuit switch, and the other flows are lower bandwidth flows (each paced at 96 Mb/s) served on the packet switch. At time t_1, one of the low bandwidth flows changes to a second high bandwidth flow — representing a dynamic shift in application behavior — and needs to be served on the circuit switch. At each subsequent time step, another lower bandwidth flow becomes high bandwidth and transitions from the EPS to the OCS.

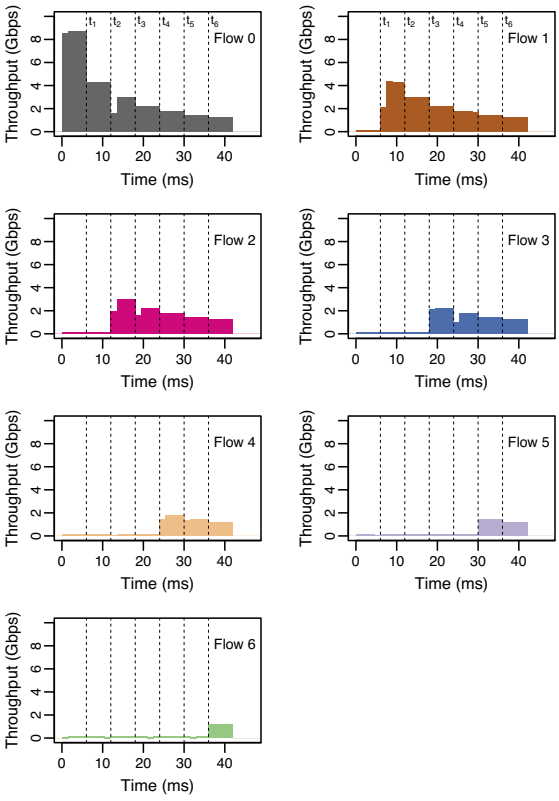


Figure 9: Goodput achieved for a time-varying workload of three flows to a single end host.

Figure 9 shows the throughput achieved by each of the flows. Initially, the high bandwidth flow has exclusive use of the circuit switch. At each time step t_i, another flow transitions from low to high bandwidth and REACToR promotes it from the EPS to the OCS. Each time, all high bandwidth flows then adjust to fairly share the link bandwidth to the receiving host. In each case, REACToR seamlessly handles the shift in traffic demands.

Note that a flow might send at a lower rate during the first 1.5 ms scheduling period. This happens when the schedule changes and a flow is served earlier in this period than the previous one. As a result, the queue buffer does not yet have enough enqueued packets to fully utilize the link. The queue buffer will be built up starting from the second scheduling period, and the flow will fully utilize the link again.

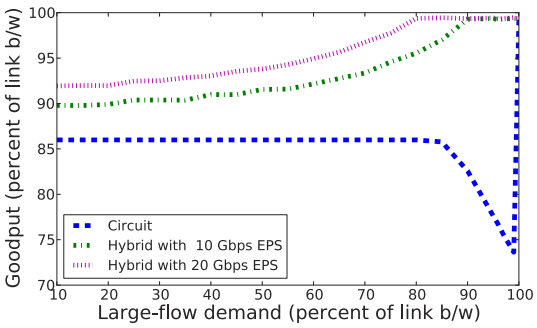


Figure 10: Performance of a circuit switch ToR and REACToR in different workload regimes.

5.4 Large benefits from a small EPS

As a final step, we illustrate how an underprovisioned packet switch in REACToR substantially relaxes the constraints of a pure circuit switch. In particular, we show how the ability to offload small flows to the packet switch enables REACToR (1) to maintain high circuit utilization and high workload goodput under our workload assumptions, and (2) to support full simultaneous endpoint connectivity for small flows.

We use simulation for these experiments to evaluate behavior beyond the constraints of our testbed. The simulator models a single REACToR switch, including the behavior of end hosts with NIC buffers, a circuit switch with switching overhead, a packet switch with buffers, and the circuit scheduler from Section 3.2. We validated the simulator output using our prototype: for workloads involving eight or fewer hosts, flow goodput calculated by the simulator always had errors less than 1% of flow goodput measured on the prototype implementation.

Maximizing circuit utilization Using the simulator, we explore the performance regimes of a single hybrid ToR-like REACToR at rack scale. For comparison, we simulate 64 end hosts connected first to a pure circuit switch and then to a REACToR switch via 100-Gb/s links. We compare with a circuit switch, not because we expect it to perform ideally well, but because it helps illustrate how a REACToR switch performs. In this experiment, each host j sends traffic to its neighboring twenty-one hosts $j+1$ through $j+21$, one flow per host. The total offered demand across all twenty-one flows is 100 Gb/s. The flow from j to $j+1$ is a "large" flow whose demand D we vary up to the full 100 Gb/s. The other twenty "small" flows have equal demands dividing the remaining (100-D)-Gb/s bandwidth equally. For scheduling circuits, each configuration has a duration of at least 40 μs (including reconfiguration delay), the scheduling period is 3000 μs (at most 75 configurations), and the reconfiguration delay is 20 μs (hence each configuration has at least 50% utilization). For REACToR, we simulate a 100-Gb/s circuit switch and a packet switch internally, where the packet switch is 10 Gb/s or 20 Gb/s.

Figure 10 shows the results for this experiment. The x-axis shows the demand of the large flow from each host as a percentage of link rate (100 Gb/s), and the y-axis shows the goodput of the ToR given the offered workload. We show three curves, one for a pure circuit switch and two for REACToR, with the curves overlapping at points. We note, of course, that a fully-provisioned packet switch as the ToR could switch this workload at full rate.

The lowest curve shows the results of using a pure circuit switch for the ToR, with the right-most point of the curve as the ideal case for a circuit switch. Hosts send all of their traffic in the large flow at 100 Gb/s (small flows have zero demand). In this case all of the flows can take full advantage of a circuit when the switch schedules one for them: each flow has data to transmit during their entire allocation in the circuit schedule. Once the small flows start to have a non-zero demand, though, there is a cliff in circuit switch performance. The demands to the other hosts, although small, are all non-zero; as a result, the switch schedules each small flow a circuit to carry its traffic. But the small flows do not have the traffic demand to fully utilize their circuit allocations, leaving them under-utilized. As the larger flow decreases in demand moving to the left, and the smaller flows correspondingly increase, the circuit switch performance improves as the small flows are better able to utilize their allocations. Once the small flows are able to fully use their circuits (when the large flow demand is at 87%), the pure circuit switch performance levels off. At this point, the lower goodput of the circuit switch is entirely due to reconfiguration delay overhead.

In comparison, the middle curve shows the performance of a hybrid ToR-like REACToR with a 100:10 capacity ratio. Between 90–100 Gb/s for the large flow ($<$ 10 Gb/s combined for the small flows), REACToR performs just like a packet switch because the combined demands of the small flows go through REACToR's packet switch while the large flows go through REACToR's circuit switch. This regime represents REACToR's ability to efficiently switch traffic that does not have good burst behavior. As long as the combination of those flows fits within the EPS "budget", REACToR has the performance of a 100-Gb/s packet switch using a combination of a 100-Gb/s circuit switch and a 10-Gb/s packet switch.

Below 90 Gb/s, REACToR performance gradually and gracefully degrades as the combined demands of the small flows exceed the 10-Gb/s per-host rate of REACToR's packet switch; notably, it avoids any discontinuities in performance. REACToR then needs to schedule

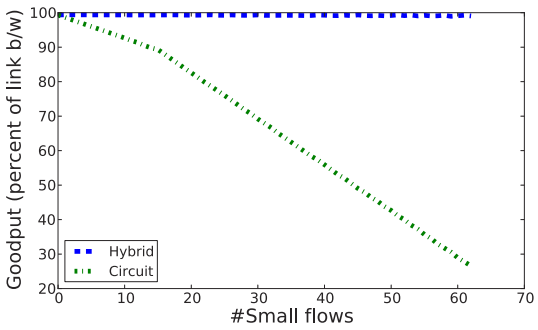


Figure 11: Performance of a circuit switch ToR and REACToR as a function of the number of small flows.

an increasingly larger portion of small flow demand on the circuit switch. REACToR goodput will decrease as a combination of imperfect utilization of circuits when assigned to small-flow demand, and additional reconfiguration delays for those circuit assignments.

Note that for this curve the circuit and packet switches had a 100:10 capacity ratio. There is nothing fundamental about this choice. A network using REACToR switches could tailor this ratio to balance cost and workloads: networks with more shorter-burst flows can deploy more EPS resources at higher cost, or vice versa. In terms of Figure 10, more EPS resources shift the point of 100% goodput for REACToR to the left, as shown by the top-most curve corresponding to an internal 20-Gb/s packet switch in REACToR.

Endpoint connectivity. In addition to maintaining high circuit utilization, the underprivisioned packet switch also enables REACToR to support many simultaneous flows between endpoints in the tail of the workload distribution. To illustrate this point, we perform one last experiment focusing on the number of simultaneous small flows between distinct endpoints in the network. In a network of 64 hosts, we represent the aspect of the workload well matched to circuits using one single large flow consuming 90% of the capacity: an ideal case for a pure circuit switched network. We then evenly split the remaining 10% among n small flows, where n varies between 1 and 64.

Figure 11 shows the goodput of the ToR (percentage of offered demand serviced by the ToR) as a function of the number of small flows for this experiment. At $n = 1$, both the hybrid and pure circuit ToRs perform the same on the trivial single large flow. The bottom curve shows the pure circuit ToR goodput in the presence of small flows. Goodput steadily decreases because the circuit switch has to assign circuits to every flow. As the number of small flows increases, the demand in each flow decreases; circuit durations decrease, but the rate of reconfigurations correspondingly increases. Hence the pure circuit ToR becomes increasingly less efficient.

The top curve shows the hybrid ToR performance. By construction, its internal packet switch can satisfy the bandwidth demands of the small flows and therefore efficiently handle the full endpoint connectivity of the workload. If the total demand of the small flows comprising the tail of the workload exceeds the capacity of the underprovisioned packet switch, then the performance of the hybrid ToR will trend towards the left-hand regime in Figure 10 (e.g., where the large flow demand drops below 90% with a 10G EPS).

6 Conclusion

Hybrid ToRs, such as REACToR, have the potential to enable scalable, high-speed networks by pairing the numerous advantages of optical circuit switching with comparatively underprovisioned packet switching. The key insight driving our work is that by moving the vast majority—but not all—of the buffering out of the switch and into end hosts, more scalable interconnect fabrics can be supported.

Practically speaking, this only works if 1) end hosts emit bursts of traffic to a given destination that are both predictable and of sufficient duration to fill OCS circuits, and 2) the hybrid scheduler operates at timescales that are invisible to the transport and applications running on the end hosts. In the first case, in-NIC buffering that historically has been used to drive line rate transmissions can be repurposed to stage impending data bursts, therefore fully using OCS circuits. In the second case, for a two-REACToR prototype, we have shown that we can schedule end hosts to make use of an OCS without negatively impacting TCP performance. A design challenge posed by interconnecting a large number of REACToRs is co-scheduling and synchronizing directly connected REACToRs to avoid the need for buffering on uplink ports. We leave this global scheduling problem for future work.

Acknowledgments

This research was supported in part by the NSF through grants EEC-0812072, MRI-0923523, and CNS-1314921. Additional funding was provided by a Google Focused Research Award and a gift from Cisco Systems. The authors thank Mindspeed for technical support and providing the custom switch board. The manuscript benefited from feedback and discussions with D. Andersen, J. Ford, D. Maltz, A. Vahdat, our shepherd, Changhoon Kim, and the anonymous NSDI reviewers.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity, data center network architecture. In *Proc. ACM SIGCOMM*, Aug. 2008.

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

[3] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proc. ACM SIGCOMM*, Oct. 2004.

[4] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. ACM IMC*, 2010.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[6] N. Calabretta, R. Centelles, S. Di Lucente, and H. Dorren. On the Performance of a Large-Scale Optical Packet Switch Under Realistic Data Center Traffic. *Journal of Optical Communications and Networking*, 5(6):565–573, June 2013.

[7] K. Chen, A. Singla, A. Singh, K. Ramachandran, L. Xu, Y. Zhang, and X. Wen. OSA: An Optical Switching Architecture for Data Center Networks and Unprecedented Flexibility. In *Proc. USENIX NSDI*, Apr. 2012.

[8] N. Farrington, A. Forencich, G. Porter, P.-C. Sun, J. Ford, Y. Fainman, G. Papen, and A. Vahdat. A Multiport Microsecond Optical Circuit Switch for Data Center Networking. *IEEE Photonics Technology Letters*, 25(16):1589–1592, June 2013.

[9] N. Farrington, G. Porter, Y. Fainman, G. Papen, and A. Vahdat. Hunting mice with microsecond circuit switches. In *Proc. ACM HotNets*, Oct. 2012.

[10] N. Farrington, G. Porter, S. Radhakrishnan, H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *Proc. ACM SIGCOMM*, Aug. 2010.

[11] S. Fu, B. Wu, X. Jiang, A. Pattavina, L. Zhang, and S. Xu. Cost and delay tradeoff in three-stage switch architecture for data center networks. In *Proc. IEEE High Performance Switching and Routing*, July 2013.

[12] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting Data Center Networks with Multi-gigabit Wireless Links. In *Proc. ACM SIGCOMM*, Aug. 2011.

[13] N. Jerger, M. Lipasti, and L. Peh. Circuit-Switched Coherence. *Computer Architecture Letters*, 6(1):5–8, July 2007.

[14] S. Kandula, J. Padhye, and P. Bahl. Flyways To De-Congest Data Center Networks. In *Proc. ACM HotNets*, Oct. 2009.

[15] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *Proc. ACM IMC*, Nov. 2009.

[16] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter. Bullet Trains: A Study of NIC Burst Behavior at Microsecond Timescales. In *Proc. ACM CoNEXT*, Dec. 2013.

[17] B. Lee, A. Rylyakov, W. Green, S. Assefa, C. Baks, R. Rimolo-Donadio, D. Kuchta, M. Khater, T. Barwicz, C. Reinholm, E. Kiewra, S. Shank, C. Schow, and Y. Vlasov. Four- and Eight-Port Photonic Switches Monolithically Integrated with Digital CMOS Logic and Driver Circuits. In *Proc. OFC/NFOEC*, Mar. 2013.

[18] X. Li and M. Hamdi. On scheduling optical packet switches with reconfiguration delay. *IEEE Journal on Selected Areas in Communications*, 21(7), Sept. 2003.

[19] D. Marom. Switching Capacity of MEMS Tilting Micromirrors. In *Proc. IEEE Optical MEMS and Nanophotonics*, Aug. 2012.

[20] J. Martin, K. K. Chapman, and J. Leben. *Asynchronous Transfer Mode: ATM Architecture and Implementation*. Prentice-Hall, Inc., 1997.

[21] T. Moscibroda and O. Mutlu. A Case for Bufferless Routing in On-Chip Networks. In *Proc. ISCA*, June 2009.

[22] R. Olsson. pktgen the linux packet generator. *Proc. Linux Symposium*, July 2005.

[23] G. Porter, R. Strong, N. Farrington, A. Forencich, P.-C. Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating microsecond circuit switching into the data center. In *Proc. ACM SIGCOMM*, Aug. 2013.

[24] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: A scalable NIC for end-host rate limiting. In *Proc. USENIX NSDI*, Apr. 2014.

[25] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan. Scale-out networking in the data center. *IEEE Micro*, 30(4):29–41, Aug. 2010.

[26] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time Optics in Data Centers. In *Proc. ACM SIGCOMM*, Aug. 2010.

[27] B. Wu, K. L. Yeung, and X. Wang. Improving scheduling efficiency for high-speed routers with optical switch fabrics. In *Proc. IEEE Globecom*, Dec. 2006.

[28] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *Proc. ACM SIGCOMM*, Aug. 2012.

Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Centers

Peng Cheng, Fengyuan Ren, Ran Shu, Chuang Lin
Dept. of Computer Science and Technology, Tsinghua University, Beijing, 100084, China
Email: {chengpeng5555, renfy, shuran, clin}@csnet1.cs.tsinghua.edu.cn

Abstract

An increasing number of TCP performance issues including TCP Incast, TCP Outcast, and long query completion times are common in large-scale data centers. We demonstrate that the root cause of these problems is that existing techniques are unable to maintain self-clocking or to achieve accurate and rapid packet loss notification. We present cutting payload (CP), a mechanism that simply drops a packet's payload at an overloaded switch, and a SACK-like precise ACK (PACK) mechanism to accurately inform senders about lost packets. Experiments demonstrate that CP successfully addresses the root cause of TCP performance issues. Furthermore, CP works well with other TCP variants used in data center networks.

1 Introduction

Modern large-scale data centers, which enable cloud computing and host online services with intensive server-side computing and storage, are significantly different from traditional data centers because of shorter round trip time (RTT), higher bandwidth, highly variable flow characteristics [4, 18] and very low latency requirements [4, 28]. Because of these differences, TCP has been found to have various performance issues in large-scale data center networks (DCNs).

In recent years, TCP Incast [22], TCP Outcast [24], the TCP out-of-order problem [32], and long-query completion times [4] have been found to significantly affect TCP performance in DCNs. Through experiments, we found that throughput cliff (described in § 2.1) and TCP unfairness in multiple-bottleneck scenarios [20] may also exist in DCNs. Collectively, these problems are referred to as "TCP problems" in this paper. These problems must be resolved to achieve high throughput and low latency, which are critical requirements of many data center applications [4, 14].

Our experimental observations and comprehensive analysis attribute TCP problems to three issues. First, self-clocking stall caused by insufficient ACKs results in Incast, throughput cliff, and unfairness. Second, inaccurate packet-loss notification caused by ambiguous loss indication results in the TCP out-of-order problem. Third, slow packet loss detection leads to long query completion time.

Packet loss notification is a powerful mechanism to address these problems. It maintains self-clocking, makes unambiguous differentiation between packet loss and out-of-order packets and shortens the detection time of packet loss. We propose a simple solution called CP[1] for packet loss notification. CP drops only the packet payload instead of the entire packet during buffer overload and uses a SACK-like precise ACK (PACK) technique to accurately inform senders of lost packets. In our experiments, CP successfully demonstrates its ability to solve TCP problems.

We implement CP in NetFPGA cards. In our implementation, CP only results in a 56-ns processing delay and less than a 2% increase in resource usage. Furthermore, the additional overhead is rarely introduced because the processing is only triggered by packet loss. Because of its low overhead and limited extra resource usage, CP can be easily added to existing commercial switches.

This paper makes two main contributions:

- Our comprehensive analysis identifies three key issues with TCP in large-scale DCNs: self-clocking stall, inaccurate packet loss notification, and slow packet loss detection.
- We propose CP, which leaves the packet header intact during packet drop processing, merely cutting out the payload to rapidly inform the sender of packet loss. Experiments demonstrate that this can solve many TCP problems and has good compatibility with other variations of TCP used in DCNs.

[1] CP is the abbreviation of "cutting payload".

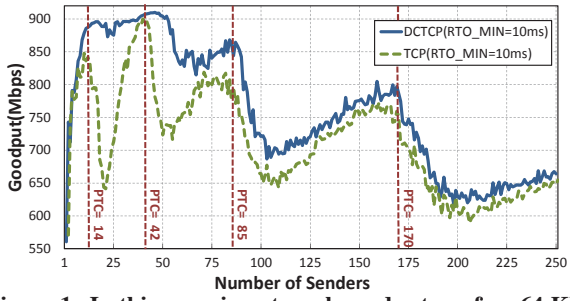


Figure 1: In this experiment, each sender transfers 64 KB data to one receiver through 128 KB bottleneck link simultaneously. The network topology and other experimental configurations are the same as MapReduce-like application scenario described in § 6.1. We observe that DCTCP avoids the throughput cliff where PTC=14 and postpones it from PTC=42 to PTC=50, but still suffers the throughput cliff with an increasing number of senders.

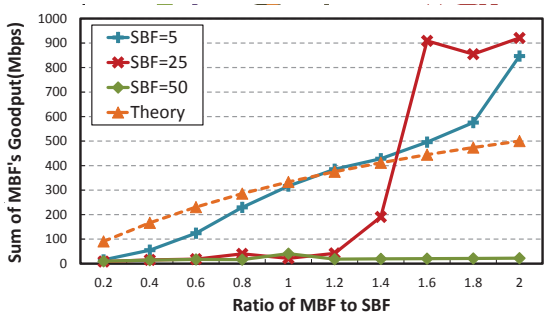


Figure 2: In this experiment, all flows use TCP with SACK, and the topology is shown in Fig. 9 (b). Other experimental configuration are the same as those described in § 6.2.

The remainder of this paper is organized as follows. In Section 2, we summarize the existing TCP problems. In Section 3, we analyze inherent reasons for these problems. In Section 4 and Section 5, we propose CP and describe its design and implementation in detail. In Section 6, we test and illustrate the performance of CP when handling TCP performance issues in DCNs. Finally, we conclude the paper in Section 7.

2 TCP in Contemporary DCNs

In this section, we summarize and discuss TCP problems in DCNs based on experimental results from a small testbed. Our testbed is made up of one aggregation switch, four top-of-rack (ToR) switches, and twelve end-hosts. More details are discussed in § 6.

2.1 Low and Volatile Throughput

In current data centers, the many-to-one communication pattern is common in applications such as MapReduce [10] and Web search applications [18]. In this pattern, data from many synchronized senders is transferred to the same receiver in parallel; thus, TCP Incast collapse naturally occurs and throughput decreases sharply. Many approaches have been proposed to address TCP Incast, such as DCTCP [4], ICTCP [29], and decreasing RTO$_{min}$ [27]. However, through experimental observation, we found that these approaches are not able to satisfy the particular demands of two major types of applications: MapReduce-like applications and Web search applications.

In MapReduce-like applications, the number of senders varies dramatically (the average is 154 with a standard deviation of 558) [19]. When the number of senders is large, the buffer associated with the bottleneck link overflows even though each sender transmits only one packet [4]. Then, because of packet loss and the one-packet transmission window, timeouts are inevitable. In order to keep high throughput and avoid TCP Incast collapse, even with DCTCP and ICTCP, the number of senders needs to stay within an upper limit. In practice, this limit fails to meet demands of large-scale applications. In addition, we observed a particular phenomenon when using a small value for RTO$_{min}$, as shown in Fig. 1. We call this a throughput cliff, reflecting that the throughput will sharply decline at the point of throughput cliff (PTC) and then slowly climb. We found that the throughput cliff is due to synchronous adjustment of the congestion window at senders and synchronous packet loss[2]. The synchronous packet loss leads to increasing timeouts and throughput decline. Thus, these well-known approaches to alleviate TCP Incast collapse cannot maintain sufficiently high throughput with a large number of senders.

In Web search applications, because of the fixed size of search result, the number of senders is not significantly large. However, the query completion time is closely related to Web search performance [28]. From our experiments, we were surprised to find that, because a high and stable throughput cannot be guaranteed by TCP or its variants, query completion time is far greater than expected and has frequent fluctuations[3].

2.2 TCP Unfairness

TCP unfairness in wide area networks and wireless networks is well known. In DCNs, TCP unfairness occurs in both single-bottleneck and multi-bottleneck scenarios.

In a single-bottleneck scenario, a port blackout [24], in which each input port accidently loses a series of packets, causes the consecutive packets loss or even entire window loss leading to TCP timeout. When there are

[2]For example, with 14 senders, if the bottleneck buffer is 128 KB and each sender increases its congestion window to 7 packets ($1.5KB \times 14 \times 6 = 126KB$), all senders will see a drop synchronously. Consequently, synchronous adjustment of congestion window and synchronous packet loss happen at PTC.

[3]The concrete results are presented in Fig. 8(b) in § 6.1. Using TCP with 10ms-RTO$_{min}$, there is a delay of twice the expected value when the number of senders is greater than 26. Using DCTCP, there is a severe fluctuation in query completion time when the number of senders is between 22 and 42.

both a large and small number of flows from two different input ports to one output port, timeouts preferentially happen on the small flows, which results in TCP Outcast. A straightforward solution called equal-length routing [24] changes the routing paths through core switches to address this problem. However, this imposes significant pressure on core switches and increases end-to-end delay. Therefore, alleviating or eliminating the damage caused by a port blackout without imposing additional costs is desirable.

In the multiple-bottleneck scenario, multiple-bottleneck flows (MBF), i.e., cross-rack flows, share two bottlenecks with two different single-bottleneck flows (SBF), i.e., in-rack flows, which have the same number of flows. Fig. 2 shows that when the packet loss ratio is very high (i.e., SBFs = 50), MBFs have a significantly lower throughput. However, when the packet loss ratio is low (i.e., SBFs = 5 or 25), the throughput of MBFs is higher than the theoretical value. Further analysis indicates that MBFs always have a higher packet loss ratio than SBFs; thus, they may lose all transmitted packets and trigger timeouts in a high packet loss ratio scenario. However, in a low packet loss ratio scenario, SBFs suffer the TCP Outcast problem, which leads to the higher throughput of MBFs. Thus, in both situations, either SBFs or MBFs suffer from unfairness.

2.3 TCP Out-of-order Problem

Several data center topologies [1, 14, 15] have been proposed to deal with bottlenecks in core switches; however, they employ many redundant links that are utilized to only 40% -50% [8, 25]. Therefore, many multipath routing techniques have been proposed, such as ECMP, Hedera [2], and Valiant Load Balancing (VLB) [14]. All these techniques are flow-based traffic splitting schemes, which are dramatically worse than an ideal packet-level one. However, packet-level schemes lead to serious throughput degradation because of out-of-order packets [11]. Therefore, tackling the out-of-order problem can significantly improve the performance of multipath routing techniques.

2.4 Long Query Completion Time

Data centers host many soft real-time online services such as retail, advertisement, and Web search[28]. Query completion time is very critical because it is directly related to the quality of these online services.

Query completion time is influenced by the amount of queuing delay and retransmission time. Long queuing delay has been addressed by some proposed schemes [4, 5, 28]. We conducted an experiment to determine if query completion time is affected by a long retransmission time. In our experiment, we ran a query with 20 flows. As shown in Fig. 3, in flow No.18, the delay

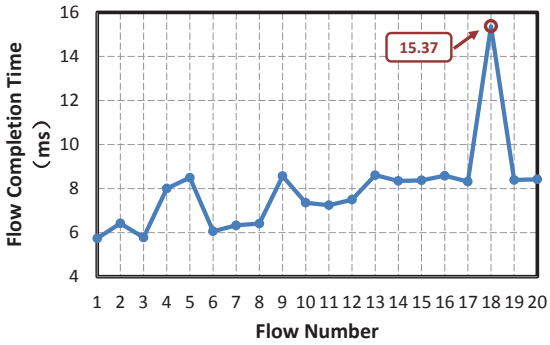


Figure 3: Each of 20 DCTCP flows transfers 50 KB to the same receiver through the 128 KB bottleneck link. We conducted experiments for 10,000 random instances, and found that in 1,231 experiments, some flow completion times are larger than 15ms. The figure shows detailed information of the completion time of each flow in one of these experiments.

is approximately two times that of other flows, thereby resulting in delay of the completion time of the entire query. After more in-depth observation, we found that flow No.18 suffers from many fast retransmissions and wastes significant time retransmitting. More than 12% of the queries suffer the same problem in our 10,000 random experimental instances. Therefore, we conclude that retransmission delay imposes a significant influence on query completion time.

3 Why TCP Does Not Work Well in DCNs

Here, we discuss three fundamental weaknesses of TCP and its variants used in DCNs.

3.1 Self-clocking Stall

The self-clocking mechanism was proposed in 1988 [17]. Essentially, the arrival of an ACK tells the sender that the network can accept another packet. Through this mechanism, TCP can maintain continuous and stable transmission and quickly fill up the pipeline. Without sufficient ACKs, however, self-clocking stops, which causes a timeout. In DCNs, the congestion window of each flow is relatively small because of the low delay-bandwidth product and large number of concurrent connections. In addition, the many-to-one traffic pattern often causes severe congestion and a high packet loss ratio. Therefore, self-clocking stall in a many-to-one traffic pattern often triggers timeouts that result in low and volatile TCP throughput. The port blackout phenomenon and high packet loss ratio of MBFs lead to the loss of the vast majority of packets in a window or even the entire window. This makes it increasingly less likely that a flow can receive a sufficient number of ACKs to maintain self-clocking. Therefore, self-clocking stall is an essential reason for low and volatile TCP throughput and TCP unfairness.

To maintain self-clocking, three enhancements have been proposed in prior work. The first mechanism is defined in RFC 3042 [7]: when a sender receives two duplicate ACKs, it sends one more packet immediately. The second mechanism is the TCP implementation in Linux kernel 2.6.3, in which a sender sends one more packet every time it receives a duplicate ACK. Unfortunately, the two mechanisms mentioned above fail when all segments in whole congestion window are lost, such as with the port blackout phenomenon. The third mechanism is tail loss probe (TLP) [12]. It transmits one packet every two round-trips when no ACK is received at the end of the transaction. This mechanism is an incomplete solution to avoid self-clocking stall because it still fails except in the tail loss case. Therefore, whether we use standard TCP, RFC 3042, TCP in Linux, or TLP, self-clocking stall remains inevitable.

This leads to the realization that we need a complementary method, apart from ACK, to maintain self-clocking for addressing TCP Incast and unfairness.

3.2 Inaccurate Packet-loss Notification

In § 2.3, we see that packet-level multipath routing techniques cause performance problems due to TCP's reaction to out-of-order packets. Here, by considering the features of DCNs, we provide further explanation. Since queuing delay causes most of the end-to-end delay in DCNs, a high-latency path is always accompanied by link congestion. Therefore, the phenomena of packet loss and out-of-order packets often co-exist in multipath DCNs. TCP deals poorly with these mixed paths due to the one-size-fits-all solution of using a fixed threshold value (three duplicate ACKs) to determine whether packet is out-of-order or lost. This often leads to spurious retransmission or sluggish congestion control [32] because it is impossible to determine the exact number of allowable out-of-order packets to set an effective threshold. TCP SACK is an effective way to avoid retransmitting received packets, however, it also cannot address this problem because it is not able to distinguish whether a packet is out-of-order or lost. Therefore, only by accurately distinguishing between lost or out-of-order packets, can TCP make correct decisions regarding packet retransmission and congestion window size.

3.3 Slow Packet loss Detection

In § 2.4, we claimed that reducing retransmission delay is important to shorten query completion time. Retransmission delay is composed of the detection time of packet loss and the retransmission time of lost packets. However, it is difficult to improve the retransmission time of lost packets because it depends on a relatively mature congestion control algorithm. Thus, we focus on reducing detection time.

There are two indicators of packet loss in TCP: timeout and three duplicate ACKs. The detection time of packet loss caused by timeout is directly related to RTT estimations and RTO_{min}. Imprecise RTT estimations [31] and improper RTO_{min} [27] lead to slow packet loss detection. Timeouts lead to other problems as well [31]; thus, avoiding timeouts is a good strategy, as discussed in § 3.1.

In the traditional Internet, packet loss is detected by the reception of three duplicate ACKs. In DCNs, due to the small congestion window, it is very unlikely that the window size is large enough to cause enough duplicate ACKs to be received by the sender in one RTT. Moreover, packet loss implies congestion where RTT becomes very long as a result of a crowded queue. Therefore, timeouts and the need for three duplicate ACKs further delays packet loss detection.

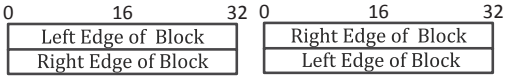
3.4 Related Work

There are three typical types of schemes to address these problems.

One is to use a TCP-like protocol, such as DCTCP. DCTCP uses ECN to adjust the congestion window. Unfortunately, when packets are dropped by the switch, the ECN information is also lost and DCTCP degenerates to standard TCP or SACK, which do not work well in DCNs. DCTCP also does not work normally in such an environment, especially with the loss of the entire congestion window. Furthermore, DCTCP does not work well in a multipath environment because of the potential for out-of-order packet. Finally, DCTCP must wait to receive three packets before retransmission when packet loss occurs, and this period can be relatively long because of potential queuing delays on switches. Both HULL [5] and D^2TCP [26], which extend DCTCP, have these same problems.

The second class of schemes is based on rate control. QCN [3], D^3 [28] and PDQ [16] use implicit or explicit rate control to avoid self-clocking stall. However, they cannot achieve accurate and rapid packet loss detection. pFabric [6] is a clean-slate design that totally solves all the above problems. Using rapid retransmission by reducing RTO_{min} to one RTT, pFabric avoids inaccurate and slow packet loss detection. However, it is hard to implement it in practice because it needs changes to both endhosts and switch hardware.

The third class of schemes uses special packets to inform the sender of a packet loss. For example, source quench [23] sends ICMP packets for every loss. The problem with such schemes is that it is very difficult for a switch to create a packet to be sent in the reverse direction when traversing the forward path. It also destroys TCP self-clocking and self-pacing.



(a) SACK (b) PACK

Figure 4: Unambiguous identities of SACK and PACK.

4 CP Design

We now discuss the design of CP.

4.1 Packet Types

CP uses four types of packets.

(1) Normal packets are sent by a sender that does not support CP. These packets are dropped by a switch when the buffer exceeds the defined threshold.

(2) CP-enabled packets are data packets with payloads that are sent by CP-capable senders. Only the payload portion of a CP-enabled packet is dropped when the buffer exceeds the defined threshold.

(3) Payload-cut packets are header-only packets without payloads. A CP-enabled packet becomes a payload-cut packet after payloads are cut off by the switch. Switches should ignore the IP length field in payload-cut packets.

(4) PACKs contain accurate packet loss information and are sent by CP-capable receivers when payload-cut packets are received.

4.2 CP Drop Processing

A switch with the payload-cutting function is called a CP switch. A CP switch has a buffer for each port to store packets awaiting forwarding. If the total size of packets in any port buffer exceeds the threshold, and a new CP-enabled packet arrives, the switch will cut off the payload of the CP-enabled packet. During this operation, first, the packet is marked as payload-cut packet. Then, the IP length is preserved for the receiver to calculate the packet size (further explanations in § 4.3), and the TCP checksum is recalculated and revised. After that, the regenerated packet without the payload queues in the buffer as a normal packet and waits for forwarding. It should be noted that a little extra buffer space is necessary to store new payload-cut packets. This will be further discussed in § 4.6.

Because of IP length field conservation, payload-cut packets may be dropped at modern switch ASICs or many middle boxes. As an alternative to achieve CP drop processing, we can preserve the first two bytes of the data payload to carry the IP length. In consideration of the similarity between these two methods, we will just discuss the first method.

4.3 Packet-loss Feedback: PACK

For backwards compatibility, we define a PACK option similar to the SACK option. As shown in Fig. 4(a), the SACK option uses the left and right edges of the block to indicate a SACKed block. The left edge represents the first sequence number of the block, and the right edge corresponds to the sequence number immediately following the last seen sequence number [21].

There are two key considerations to accomplish designing PACK. The first is determining the left edge and right edge of the lost packet from the payload-cut packet. The second is how to represent the lost packet information such that it is compatible with SACK. For the first issue, we can easily obtain the left edge from the sequence number in the TCP header of the payload-cut packet. The original packet length in the IP header is preserved; hence, we can use it to calculate the right edge[4]. For the second issue, as shown in Fig. 4(b), we swap the position of the left and right edge to indicate lost packet information. This arrangement will not produce ambiguity because the left edge must be smaller than the right edge in the SACK option. In this manner, we can parsimoniously indicate lost packet information in the PACK option. When a payload-cut packet arrives, the receiver simply parses it and adds the lost packet information in the first block of the PACK option, which is similar to the processing in SACK. For convenience, the receiver marks the packet to tell the sender to parse the PACK option. Marking will be discussed in § 5.1.

For simplicity, the two-byte PACK-Permitted option is similar to SACK-Permitted option. The option is sent in the SYN by the sender to inform the receiver that it can support the PACK option. In our implementation, we just use the SACK-Permitted option: this does not affect the performance of CP.

4.4 Sender Reaction to PACK

TCP SACK maintains a "scoreboard" to store the status of packets [9]. Every packet is in one of four states, i.e., "received" (or "SACKed"), "out-of-order," "lost," and "retransmitted." Like SACK, CP stores packet information in a "scoreboard." When the new packet is transmitted, its status changes to "out-of-order." We divide the sender action on receiving PACK into three steps. In the first step, the sender parses the PACK option as described in § 4.3 and checks state of the packet. If the status of the packet is "received" or "lost," the sender takes no action. Otherwise, it enters the second step, converting the packet status in the scoreboard to "lost" and adding the packet to the retransmission queue. In the third step, if TCP is not in a state of fast retransmission, the sender triggers the fast retransmission mechanism and performs congestion control. When the packet has been retrans-

[4] right_edge = left_edge + SYN + FIN + original_length - IP_header_length - TCP_header_length. In Linux kernel, SYN and FIN have only 1 bytes payload by default, although the payload length is 0 in reality. Therefore, if the packet is SYN or FIN packet, SYN or FIN in the above formula equals 1. Otherwise, both are 0.

mitted, the packet status changes to "retransmitted". It should be noted that using three duplicate ACKs to indicate packet loss is not necessary unless payload-cut packets or PACKs themselves are lost. Therefore, we recommend that the threshold for duplicate ACKs is set to a relatively large number.

4.5 Benefits

CP overcomes the three TCP defects as follows:

Self-clocking Stall: Payload-cut packets and PACKs provide the sender with packet loss notification. Using this notification, the sender can maintain a self-clocking mechanism and avoid timeouts.

Inaccurate Packet loss Notification: Payload-cut packets provide the receiver with accurate packet-loss information, which is brought back to the sender by PACKs. Thus, the sender can easily distinguish between lost and out-of-order packets.

Slow Packet loss Detection: Because PACKs quickly carry the information of packet loss back to the sender, the sender could start the retransmission immediately without waiting for three duplicate ACKs. Therefore, CP shortens the packet loss detection time.

Note that CP is compatible with existing congestion control protocols proposed for data centers because it is an additional mechanism for loss detection. In addition, it is TCP-friendly and backward compatible.

4.6 Discussion

The extra buffer can be small: An extra buffer or a lower drop threshold is needed at a CP switch to hold payload-cut packets during an output buffer overload. It only needs to be one maximum packet size because the input bandwidth becomes smaller than output bandwidth after CP processing. For example, the average packet size (without MAC header) in DCNs is 850 bytes [8]. CP reduces the packet length to only 66 bytes, a reduction by about 92.37%[5]. Thus, with CP, a 1 Gbps output link can tolerate an input burst at about 13.1 Gbps (1 Gbps / 7.63%). Therefore, the extra buffer can be quite small without loss of payload-cut packets.

Loss of payload-cut packets or PACKs: The loss of payload-cut packets or PACKs is a rare occurrence because of their small size. Nevertheless, even if one PACK is dropped, the sender can recover the information of the lost PACK from a subsequent PACK. Similarly, if a payload-cut packet is dropped, the sender can use duplicate ACKs to confirm the lost packet, which causes CP to degenerate to standard TCP SACK.

[5]The average packet length is 864 B (14 MAC header + 850 average length) and payload-cut packet length is 66 B (14 MAC header + 20 IP header + 20 TCP header + 12 timestamp option). The ratio of the header length to packet length is about 7.63% for this packet length.

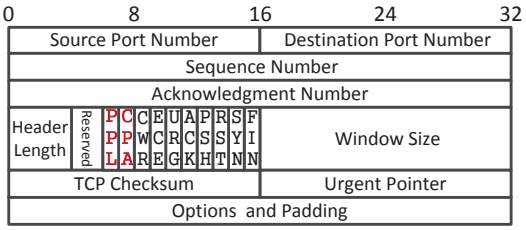


Figure 5: Extended TCP header for CP implementation

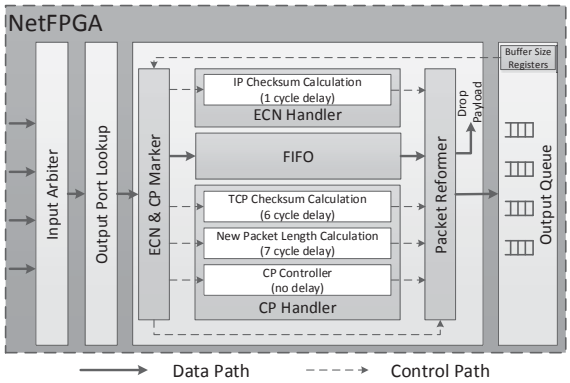


Figure 6: Structure of NetFPGA for implementing the CP switch. The packets are forwarded through the data path, and the decision-making modules in the data path generate signals that tell the processing modules whether or how to deal with the packet via the control path.

5 Implementation

This section discusses CP implementation details.

5.1 Protocol Details

As shown in Fig. 5, two reserved bits in the TCP header are defined as CP-available (CPA) and precise packet loss (PPL) to identify the four types of packets mentioned in § 4.1. Packets with CPA = 1 and PPL = 0 are CP-enabled packets and those with CPA = 0 and PPL = 1 are PACK packets. When the payload of CP-enabled packets are cut off, PPL becomes 1. Therefore, packets with CPA = 1 and PPL = 1 are payload-cut packets. To maintain compatibility, packets with CPA = 0 and PPL = 0 are defined as normal TCP packets. CP switches handle them in the same way as commodity switches.

5.2 CP switch

CP switches cut off the payload of CP-enabled packets and change the TCP checksum when a buffer exceeds the threshold. In our prototype, we also add ECN capability to allow us to compare DCTCP with CP as discussed in § 6, .

Fig. 6 shows the basic structure of the CP switch implemented in NetFPGA cards. When a packet enters the switch, it first looks for an output port and then waits for the handling of the ECN & CP marker. Depending on the packet type and buffer occupancy level, the ECN & CP marker determines how to deal with packets

Table 1: Resource Usage of NetFPGA

	Reference Switch	ECN Switch	CP Switch
Slices	12807	13579	13777
Slice Flip Flops	14158	14365	14511
4 Input LUTs	17589	17907	18239

and directs the operation of related modules. For example, on seeing payload-cut packets or PACK packets, the ECN & CP marker allows them to pass directly through all modules in the data path and places them in the forwarding buffer. All modules work in parallel, and the packet waits in the FIFO queue until all modules have finished their processing. After a short processing delay (seven-cycle delay for CP handler; one-cycle delay for ECN handler; no delay for others[6]), packet reformer regenerates the packet, which introduces no delay and little additional overhead. Finally, the packet is placed in the output buffer and waits for forwarding. As discussed in § 4.6, an extra buffer space may be necessary to absorb all payload-cut packets corresponding to dropped packets; however, the extra buffer size can generally be quite small, because some payload-cut packet loss is acceptable. Therefore, the extra buffer is set to 4 KB in our implementation.

The CP handler comes out three operations to implement CP: **(1) TCP checksum calculation.** Recomputing a checksum is only needed for the TCP header whose maximum size is 48 B (8 IP address + 20 TCP header + 20 padding); thus, the overhead is very limited. For example, NetFPGA can process 8 B in a cycle, and no more than 6 cycles (i.e., 48 ns) are consumed to re-compute the checksum. With increased processing capacity, this overhead can be further reduced and will become negligible. **(2) New packet length calculation.** This module changes the packet length parameter when sending a packet header because of NetFPGA requirement. **(3) CP controller.** This module tells packet reformer to cut out the payload.

Clearly, CP implementation is quite simple, and the processing delay at the switch is very small. Furthermore, the cutting off payload mechanism is not a normal operation for each packet, because it is only triggered to avoid packet dropping. Therefore, the additional processing delay at the switch is not introduced frequently. In addition, CP introduces only a little resource consumption on switches. Table 1 shows that, compared with ECN switches, the resource usage in CP switches only increases by less than 2%.

[6]The clock rate of NetFPGA is 125 MHz. Each cycle is equivalent to 8 ns, i.e., CP drop processing introduces a delay of 56 ns, and an additional delay of ECN is 8 ns. ECN marking must change one bit in each packet; thus, the new IP checksum is similar to the former and is easily calculated. Otherwise, the CP mechanism will cut off all payloads and require more time to recalculate the TCP checksum.

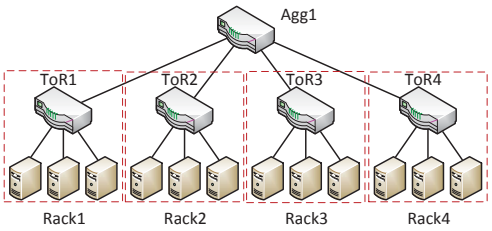


Figure 7: Basic topology of our testbed

We conclude that the implementation complexity, processing delay, and resource consumption of CP switches are acceptable; thus, CP drop processing can be built into commercial switches.

6 Evaluation

This section is divided into four parts. In each subsection, we test the effectiveness of CP in addressing each TCP problem mentioned in § 2. We also compare its performance with that of other state-of-the-art protocols used in DCNs. Our experimental results showed that CP addresses the TCP problems and achieves the expected performance goals.

Testbed. We use a real testbed to evaluate CP performance. Our testbed is shown in Fig. 7. All switches in our experiments are NetFPGA cards with four 1-Gbps Ethernet ports. The implementation of the CP switch as described in § 5.2 is downloaded to the NetFPGA cards, and the buffer sizes of each port are arbitrarily set from 1 KB to 512 KB. Each ToR switch communicates with others through the aggregation switch and connects three hosts (Dell OptiPlex 360 desktops with an Intel Celeron Dual-Core 2930 MHz CPU, 4 GB RAM and 1 Gbps NIC). All hosts in our testbed are running CentOS 5.5 with Linux kernel 2.6.38 with protocol patches applied. The RTT without queuing delay is approximately 100 μs between two endhosts in the same rack.

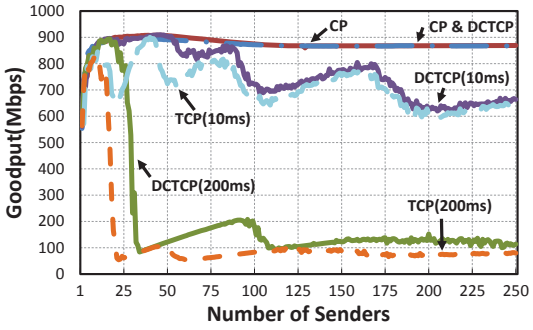
Protocols. We study four congestion control schemes.

(i) TCP: The TCP variant we study is TCP SACK. We also allow DSACK [13]. The TCP receive window size is set to 256 KB so that TCP can meet a 1 Gbps line rate. We disable delayed ACK to avoid performance problem [30]. The number of tolerable out-of-order packets is three by default. Furthermore, RTO_{min} is set to either 10 ms or 200 ms and this is denoted TCP (10ms) and TCP (200ms), respectively.

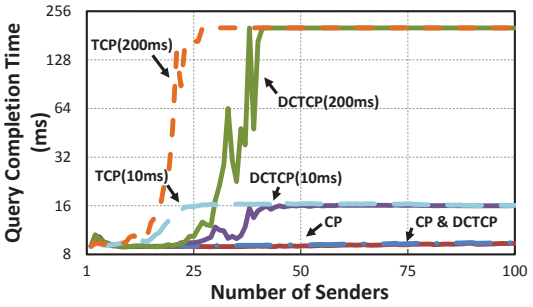
(ii) CP: Switches carry out CP; receivers generate PACKs. Other settings are the same as TCP.

(iii) DCTCP: The parameters are set to $K = 32$ KB and $g = 1/16$ [4]. The other settings are the same as for TCP, including two values for RTO_{min}.

(iv) DCTCP with CP (CP&DCTCP): CP is used along with DCTCP. Other settings are the same as for DCTCP.



(a) MapReduce-like Apps



(b) Web search Apps

Figure 8: Experimental results of scenarios for many-to-one transmission. Notice the log scale in Fig. 8(b).

6.1 Low and Volatile Throughput Impairment

Here, we explore the results of using CP in two typical scenarios (MapReduce-like and Web search application scenarios), which have different performance criteria (see § 2.1).

Topology and parameter settings. In these tests, nine hosts send data to a host on another rack. Each sending host is used to emulate multiple senders [28]. Flows are bottlenecked at the link from the aggregation switch to the receiving host. The buffer on the bottleneck link is 128 KB, and the other buffers are 512 KB. Because the settings of RTO_{min} can have a significant impact on performance in these scenarios [27], two different RTO_{min} settings (10ms and 200ms) were studied.

MapReduce-like application scenario. In this test, a receiver generates a query to each sender, and each of them immediately responds with 64 KB of data. Fig. 8(a) shows that only CP successfully avoids both TCP Incast and the throughput cliff when the number of senders is large. both DCTCP and TCP with 10-ms-RTO_{min} maintain a throughput of about 600 Mbps; however, the throughput cliff is not avoided. We see that CP and DCTCP with CP achieve a high throughput[7] and avoid throughput cliff because the payload-cut packets and PACK packets maintain the self-clocking.

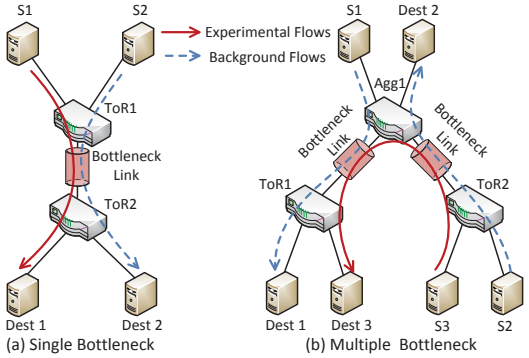


Figure 9: Experimental scenario of unfairness impairment assessment.

Web search application scenario. In this test, each of n different senders transfers $1024/n$ KB to the receiver through the bottleneck link. Fig. 8(b) shows that CP and DCTCP with CP have better performance than TCP and DCTCP in reducing query completion time. The query completion times of CP and DCTCP with CP are approximately 9 ms when the number of senders is less than 50 and slightly increases by 0.4 ms as the number of senders increases from 1 to 100. In comparison, depending on 10-ms- and 200-ms-RTO_{min}, the delays of TCP or DCTCP converge at 16.1 ms and 201.6 ms, respectively. Through closer observation and analysis, we found that CP totally eliminates timeouts and wastes only 3.49% of the 1 Gbps bandwidth to send 554 payload-cut packets (554×66 B = 35.7 KB) for each query when the number of senders is 100. However, increasing the number of senders[8] for TCP and DCTCP results in timeouts and prolongs the query completion time. In addition, using DCTCP with CP reduces the query completion time by 91.3 μs compared with CP because DCTCP with CP maintains short queue length and reduces the probability of packet loss.

6.2 TCP Unfairness

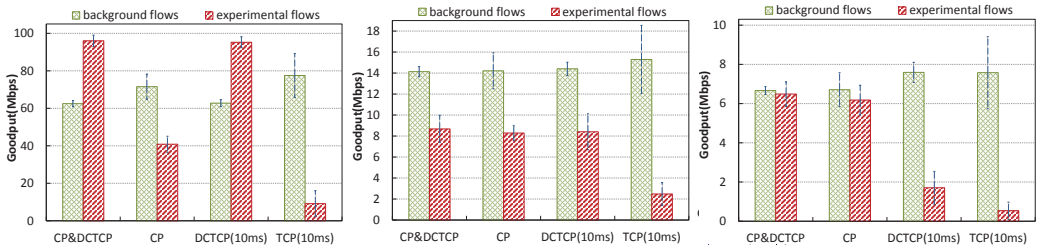
In this section, we focus on whether CP can improve TCP unfairness.

Topology. We use topologies studied in prior works for both the single-bottleneck case [24] and the multiple-bottleneck case [20], as shown in Fig. 9. It should be noted that the number of background flows shown in Fig. 9 (b) are equal. We use a subset of our testbed to achieve these scenarios, and each buffer on the bottleneck links is 128 KB[9].

[7] In CP implementation, the receiver obtains approximately 43 payload-cut packets for each sender and only wastes 4.32% of the 1 Gbps bandwidth when the number of senders is 250.

[8] In our experiment, DCTCP and TCP suffer from TCP Incast when the numbers of senders are 38 and 23, respectively. Below these number of senders, there are large fluctuations in the query completion time even if TCP Incast disappears.

[9] only the critical components are illustrated. An aggregation switch (Fig. 9(a)) and a ToR switch (Fig. 9(b)) are not shown because they do not affect the results.



(a) 12-2 flow pair (b) 60-10 flow pair (c) 120-20 flow pair

Figure 10: Average goodput of different flow-pairs in the single-bottleneck experiment. We use the notation $n-m$ to refer to n background and m experimental flows.

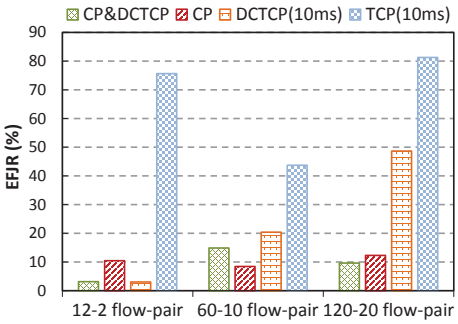
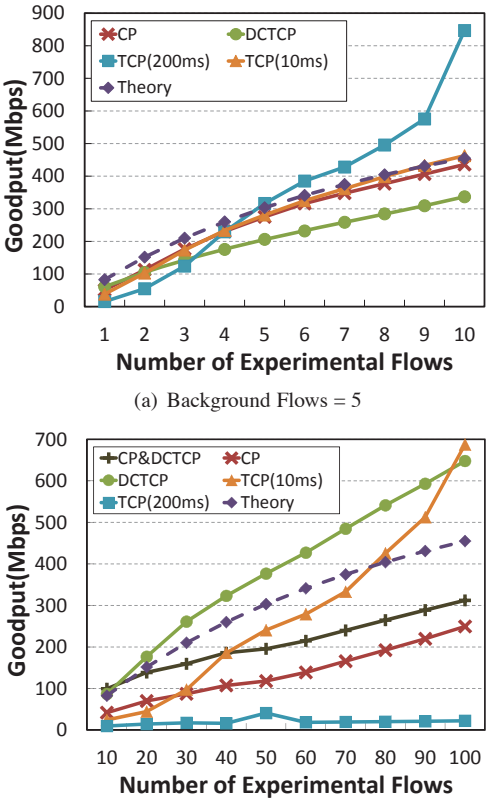


Figure 11: Experimental flows jitter ratio (EFJR) in single-bottleneck scenario

Single-bottleneck scenario. We refer to the combination of n background flows and m experimental flows as an n-m flow pair. From the experiments, we found that TCP and DCTCP with different RTO_{min} have similar features; therefore, we only show the results for 10-ms-RTO_{min} TCP and DCTCP. Three flow pairs from our experiment are shown in Fig. 10. The error bars indicate the average absolute deviation of goodput. For convenience, we refer to the experimental flow ratio of the average absolute deviation to average throughput as the experimental flow jitter ratio (EFJR). EFJR reflects the fairness among experimental flows.

Three points are evident from Fig. 10 and Fig. 11. First, the experimental flows using TCP suffer the TCP Outcast problem in all three cases. The goodput of flows with larger number is much lower than that with smaller number. Second, DCTCP can alleviate the TCP Outcast problem for the 60-10 flow pair and the 12-2 flow pair; however, when the number of flows is large (e.g., the 120-20 flow pair), DCTCP suffers the TCP Outcast problem and its EFJR is high. In additional, the EFJR of DCTCP significantly fluctuates from 3% to 48%. It is evident that DCTCP can maintain flow fairness only when number of flows is small. Third, CP prevents TCP Outcast from occurring and maintains the EFJR below 15%. DCTCP with CP has a smaller EFJR than CP; therefore, DCTCP with CP provides better fairness among flows. In conclusion, compared with TCP and DCTCP, DCTCP with CP and CP both avoid TCP Outcast problem and achieve reasonable fairness among flows.



(a) Background Flows = 5

(b) Background Flows = 50

Figure 12: Total goodput of all experimental flows in the multiple-bottleneck scenario.

Multiple-bottleneck scenario. In this scenario, from Fig. 9(b), we see that experimental flows are MBFs and background flows are SBFs. Two sets of experiments were conducted to explore the fairness of experimental flows under different packet loss conditions. We chose 5 flows and 50 flows for the number of background flows to represent low and high packet loss scenarios, respectively. Each experiment lasted 22 minutes. The ratio of the number of experimental flows to that of the background flows was increased by 20% every two minutes from 0 to 200%. From the experiments, we found that DCTCP with different RTO_{min} values has a similar per-

formance, so Fig. 12 only illustrates DCTCP with 200-ms-RTO$_{min}$, denoted DCTCP.

In Fig. 12, the line denoted "Theory" is determined by calculating the throughput according to RTT fairness [10]. Deviation above this line indicates unfairness to SBFs, and below this line to MBFs. Three conclusions can be drawn from the data presented in the low loss rate experiment shown in Fig. 12(a). First, we discover that 200-ms-RTO$_{min}$ TCP suffers the TCP Outcast problem when the number of flows is greater than 8 because background flows suffers timeout and share extremely low bandwidth. Under these conditions, the other protocols show normal performance. Second, compared with 10-ms-RTO$_{min}$ TCP and CP, DCTCP demonstrates poor fairness because it only achieves an average of 70% of the theoretical fair value, while 10-ms-RTO$_{min}$ TCP and CP achieve an average of 87.1% and 86.8%, respectively. Third, compared to 10-ms-RTO$_{min}$ TCP, CP has slight unfairness because rapid packet loss notification favors short RTT flows. Fig. 12(b) shows that, in a high packet loss environment, 200-ms-RTO$_{min}$ TCP has very low goodput. In addition, 10-ms-RTO$_{min}$ TCP and DCTCP experience the TCP Outcast problem when the number of experimental flows increases. In conclusion, CP and DCTCP with CP perform better.

From these results above, we conclude that, as the packet loss ratio increases, the TCP Outcast problem occurs initially and then low throughput of MBF occurs. Both phenomena affect TCP fairness; 10-ms-RTO$_{min}$ TCP and DCTCP can only alleviate the problem but not solve it. Furthermore, DCTCP with CP significantly improves TCP fairness and increases throughput up to 75.1% of the theoretical fair value. Unfortunately, CP alone, which achieves only an average of only 45.6% of the theoretical value, only eases unfairness rather than solving it because TCP congestion control is significantly affected by the violent queue oscillations in a high packet loss environment. Thus, CP with DCTCP is the best approach to maintain fairness in low or high packet loss environments with multiple bottlenecks.

6.3 TCP Out-of-order Problem

Topology. Fig. 13 shows the basic topology used in these experiments. An additional aggregation switch is added to the testbed. It should be noted that the ToR1 switch is a random forwarding switch that sends packets from S1 to Path A or Path B with the probability of selecting each path chosen by a configuration parameter. Other switches forward packets according to the look-up table.

[10] We define G as the theoretical goodput and N as the number of flows. Through experimental measurements, we found that the RTT of background flows is half that of experimental flows. Therefore, $G_{experimental} = N_{experimental}/(N_{background} * 2 + N_{experimental}) \times G_{total}$ where $G_{total} = 910Mbps$.

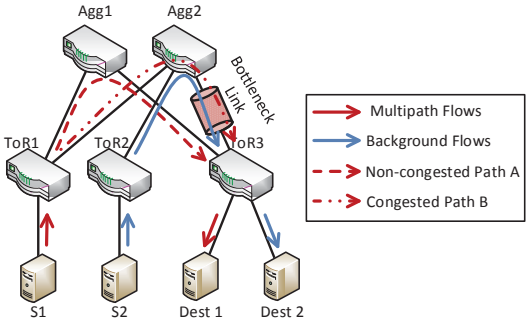


Figure 13: Basic topology of TCP out-of-order experiment

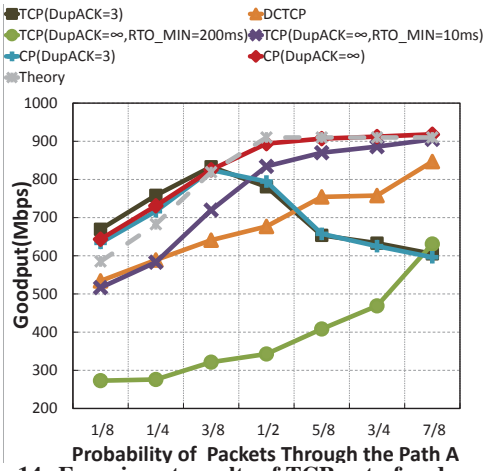


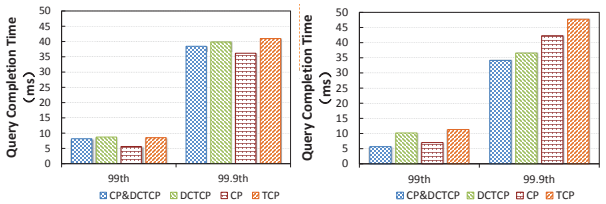
Figure 14: Experiment results of TCP out-of-order experiment

ACKs from Dest1 only pass through Path B back to S1. Furthermore, the bottleneck buffer is set to 256 KB. This creates an environment in which packets passing through Path A to reach Dest1 take less time than the packets through Path B.

Experimental parameter settings. We allow the number of tolerable out-of-order packets to be set to three or infinity. In our experiment, DCTCP achieved similar throughput in both cases; thus, we only show the results for DCTCP with three duplicate ACKs (denoted DCTCP in Fig. 14). The theoretical value is calculated by the probability of packets passing through Path A [11].

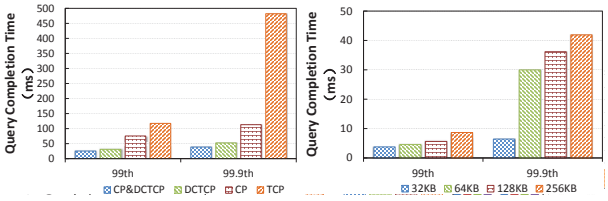
Experimental results. We can draw four conclusions from the results shown in Fig. 14. First, the small number of tolerable out-of-order packets (three) causes throughput decline even though most packets pass through the non-congested Path A. With the increasing number of packets passing through Path A, the throughput of TCP or CP with three duplicate ACKs decreases from 825 Mbps to around 600 Mbps. This occurs because a small number of packets moving through the congested Path B leads to spurious retransmission and

[11] We define G as the theoretical goodput, P as the probability of packets through the Path A. Theoretically the sub-flows of multi-path flows through Path B gets half of the bandwidth. Therefore, $G_{Multipath} = 1/2 \times G_{bandwith}/(1-P)$ where $G_{bandwith} = 910Mbps$.



(a) Agg Buffer=128KB (b) Agg Buffer=384KB

Figure 15: Query completion time under realistic Data Center traffic



(a) 10X Expanded Traffic (b) Different Buffer Size

Figure 16: Query completion time under different conditions.

unnecessary congestion control. Second, the large number of tolerable out-of-order packets (infinity) causes TCP to have a slow response to congestion and triggers a significant number of timeouts. From the comparison of 200-ms-RTO$_{min}$ TCP, 10-ms-RTO$_{min}$ TCP and the theoretical value, we find that 200-ms-RTO$_{min}$ TCP achieves less throughput than 10-ms-RTO$_{min}$ TCP because RTO$_{min}$ determines how quickly the protocol reacts to congestion as the timeout occurs. Rapid reaction to congestion increases total bandwidth utilization. In addition, the extent of congestion declines with increasing probability of packets passing through Path A. Therefore, TCP throughput approaches the theoretical value. Third, DCTCP is affected by the ECN mechanism and the early congestion control makes the throughput lower than the theoretical value. Finally, CP with no threshold for out-of-order packets works well and matches the theoretical value. In summary, CP with no threshold for duplicate ACKs can completely solve the out-of-order problem.

6.4 Query Completion Time

Experimental parameter settings. The literature [4] describes the PDF of background flow size distribution, the interval time between arrivals of queries, and the interval time between arrivals of background flows in realistic DCNs. According to this information, we generate realistic traffic of DCNs with 12 servers in our testbed. Unless otherwise specified, all switch buffers of each port are set to 128 KB. In the experiment, each server independently selects a time value and data size from identical time interval and data size distributions. One query is immediately sent to the other 11 servers after its arrival, and responses are sent back to the originating server. Both query size and response size are 2 KB. We conducted the experiment using DCTCP with CP, CP, 10-ms-RTO$_{min}$ DCTCP, and 10-ms-RTO$_{min}$ TCP. The experiment lasted 10 minutes and generated over 50,000 queries and background flows separately. We also conducted the previously reported 10x-realistic traffic experiment [4] in which the size of responses and background flows larger than 1 MB were increased tenfold. Furthermore, in the different buffer size experiments, all switch buffers are set to a specific value.

Realistic data center traffic. Fig. 15 shows the 99th and 99.9th percentile of query completion time with 128 KB and 384 KB at the aggregation switch. It can be seen that CP reduces the query completion time of TCP and DCTCP. In Fig. 15(a), compared with TCP, CP achieves a 34.06% reduction at 99th percentile and a 11.74% reduction at 99.9th percentile, respectively. At the 99th and 99.9th percentile, query completion time of DCTCP is 0.56ms and 1.41ms higher than that of DCTCP with CP, respectively. Similarly, in Fig. 15(b), query completion time of CP is 61.91% and 88.60% of that of TCP at the 99th and 99.9th percentile, respectively. Compared with DCTCP, CP with DCTCP achieves a 4.5ms decrease at 99th percentile and a 2.4ms decrease at 99.9th percentile. No query traffic suffered timeouts during the experiments. Thus, the query completion time reduction is due to the rapid packet loss detection using CP.

Expanded traffic or different buffer size. We conducted experiments using 10x-expanded traffic and a different buffer size. As can be seen in Fig. 16(a), compared with TCP, CP achieves a 35.71% reduction at 99th percentile and a 76.55% reduction at 99.9th percentile. Compared with DCTCP, DCTCP with CP achieves a 18.5% reduction at 99th percentile and a 26.2% reduction at 99.9th percentile. These results indicate that CP effectively decreases query completion time even under 10x-expanded traffic condition.

Fig. 16(b) shows the query completion times with CP with different buffer size. Compared with 128 KB and 256 KB buffer, 32 KB buffer achieves 33.07% and 56.49% time reduction at the 99th percentile, respectively. The 99.9th percentile query completion time with a 32 KB buffer is only 6.426 ms, which is 17.77% and 15.34% of that with 128 KB and 256 KB buffer, respectively. It is clear that the combination of CP and a shallow buffer can achieve good performance in DCNs.

7 Conclusion

In this paper, we proposed the cutting payload (CP) approach to solve TCP problems. Analysis indicates that TCP problems are due to three types of issues. To address these imperfections, CP uses payload-cut packets and PACK packets to maintain self-clocking and to

rapidly and precisely inform a sender of packet loss. The experimental results verify that CP can solve the problems with TCP discussed in this paper. In addition, CP is compatible with nearly all existing congestion control protocols in DCNs. In particular, the combination of DCTCP and CP has the best performance across all topologies in experiments.

Acknowledgments

We gratefully appreciate our shepherd Prof. S. Keshav for his constructive suggestions, and acknowledge the anonymous reviewers for their valuable comments. This work is supported in part by National Basic Research Program of China (973 Program) under Grant No. 2012CB315803 and 2014CB347800, and National Natural Science Foundation of China (NSFC) under Grant No. 61225011.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, August 2010.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, August 2010.

[3] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *Allerton CCC*, August 2008.

[4] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. DCTCP: Efficient Packet Transport for the Commoditized Data Center. In *SIGCOMM*, August 2010.

[5] 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*, April 2012.

[6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-Optimal Datacenter Transport. In *SIGCOMM*, August 2013.

[7] M. Allman, H. Balakrishnan, and S. Floyd. RFC 3042: Enhancing TCP's Loss Recovery Using Limited Transmit.

[8] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. In *WREN*, August 2009.

[9] E. Blanton, M. Allman, K. Fall, and L. Wang. RFC 3517: A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, December 2004.

[11] A. Dixit, P. Prakash, and R. R. Kompella. On the Efficacy of Fine-grained Traffic Splitting Protocols in Data Center Networks. In *SIGCOMM*, August 2011.

[12] N. Dukkipati. Tcp: Tail Loss Probe (TLP). `http://lwn.net/Articles/542642/`.

[13] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. RFC 2883: An Extension to the Selective Acknowledgement (SACK) Option for TCP.

[14] 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*, August 2009.

[15] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *SIGCOMM*, August 2009.

[16] C. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, August 2012.

[17] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, August 1988.

[18] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *IMC*, November 2009.

[19] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An Analysis of Traces from a Production Mapreduce Cluster. In *CCGRID*, May 2010.

[20] A. Mankin. Random Drop Congestion Control. In *SIGCOMM*, September 1990.

[21] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. RFC 2018: TCP Selective Acknowledgment Options.

[22] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale Storage Cluster: Delivering scalable high bandwidth storage. In *SC*, November 2004.

[23] J. Postel. RFC 792: Internet Control Message Protocol.

[24] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella. The TCP Outcast Problem: Exposing Unfairness in Data Center Networks. In *NSDI*, April 2012.

[25] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, August 2011.

[26] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP(D^{2}TCP). In *SIGCOMM*, August 2012.

[27] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *SIGCOMM*, August 2009.

[28] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, August 2011.

[29] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *CoNEXT*, December 2010.

[30] 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 *NSDI*, March 2011.

[31] L. Zhang. Why TCP Timers Don't Work Well. In *SIGCOMM*, August 1986.

[32] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *Proceedings of ICNP*, November 2003.

High Throughput Data Center Topology Design

Ankit Singla, P. Brighten Godfrey, Alexandra Kolla
University of Illinois at Urbana–Champaign

Abstract

With high throughput networks acquiring a crucial role in supporting data-intensive applications, a variety of data center network topologies have been proposed to achieve high capacity at low cost. While this work explores a large number of design points, even in the limited case of a network of identical switches, no proposal has been able to claim any notion of *optimality*. The case of *heterogeneous* networks, incorporating multiple line-speeds and port-counts as data centers grow over time, introduces even greater complexity.

In this paper, we present the first non-trivial upper-bound on network throughput under uniform traffic patterns for *any* topology with identical switches. We then show that random graphs achieve throughput surprisingly close to this bound, within a few percent at the scale of a few thousand servers. Apart from demonstrating that homogeneous topology design may be reaching its limits, this result also motivates our use of random graphs as building blocks for design of heterogeneous networks. Given a heterogeneous pool of network switches, we explore through experiments and analysis, how the distribution of servers across switches and the interconnection of switches affect network throughput. We apply these insights to a real-world heterogeneous data center topology, VL2, demonstrating as much as 43% higher throughput with the same equipment.

1 Introduction

Data centers are playing a crucial role in the rise of Internet services and big data. In turn, efficient data center operations depend on high capacity networks to ensure that computations are not bottlenecked on communication. As a result, the problem of designing massive high-capacity network interconnects has become more important than ever. Numerous data center network architectures have been proposed in response to this need [2, 10–15, 20, 23, 25, 26, 30], exploiting a variety of network topologies to achieve high throughput, ranging from fat trees and other Clos networks [2, 13] to modified generalized hypercubes [14] to small world networks [21] and uniform random graphs [23].

However, while this extensive literature exposes several points in the topology design space, even in the limited case of a network of identical switches, it does not answer a fundamental question: *How far are we from throughput-optimal topology design?* The case of *heterogeneous* networks, *i.e.*, networks composed of switches or servers with disparate capabilities, introduces even greater complexity. Heterogeneous network equipment is, in fact, the common case in the typical data center: servers connect to top-of-rack (ToR) switches, which connect to aggregation switches, which connect to core switches, with each type of switch possibly having a different number of ports as well some variations in line-speed. For instance, the ToRs may have both 1 Gbps and 10 Gbps connections while the rest of the network may have only 10 Gbps links. Further, as the network expands over the years and new, more powerful equipment is added to the data center, one can expect more heterogeneity — each year the number of ports supported by non-blocking commodity Ethernet switches increases. While line-speed changes are slower, the move to 10 Gbps and even 40 Gbps is happening now, and higher line-speeds are expected in the near future.

In spite of heterogeneity being commonplace in data center networks, very little is known about heterogeneous network design. For instance, there is no clarity on whether the traditional ToR-aggregation-core organization is superior to a "flatter" network without such a switch hierarchy; or on whether powerful core switches should be connected densely together, or spread more evenly throughout the network.

The goal of this paper is to develop an understanding of how to design high throughput network topologies at limited cost, even when heterogeneous components are involved, and to apply this understanding to improve real-world data center networks. This is nontrivial: Network topology design is hard in general, because of the combinatorial explosion of the number of possible networks with size. Consider, for example, the related[1] *degree-diameter problem* [9], a well-known graph theory problem where the quest is to pack the largest possible number of nodes into a graph while adhering to constraints on both the degree and the diameter. Nontrivial optimal solutions are known for a total of only

[1] Designing for low network diameter is related to designing for high throughput, because shorter path lengths translate to the network using less capacity to deliver each packet; see discussion in [23].

seven combinations of degree and diameter values, and the largest of these optimal networks has only 50 nodes! The lack of symmetry that heterogeneity introduces only makes these design problems more challenging.

To attack this problem, we decompose it into several steps which together give a high level understanding of network topology design, and yield benefits to real-world data center network architectures. First, we address the case of networks of homogeneous servers and switches. Second, we study the heterogeneous case, optimizing the distribution of servers across different classes of switches, and the pattern of interconnection of switches. Finally, we apply our understanding to a deployed data center network topology. Following this approach, our key results are as follows.

(1) Near-optimal topologies for homogeneous networks. We present an upper bound on network throughput for any topology with identical switches, as a function of the number of switches and their degree (number of ports). Although designing *optimal* topologies is infeasible, we demonstrate that random graphs achieve throughput surprisingly close to this bound—within a few percent at the scale of a few thousand servers for random permutation traffic. This is particularly surprising in light of the much larger gap between bounds and known graphs in the related degree-diameter problem [9][2].

We caution the reader against over-simplifying this result to 'flatter topologies are better': Not all 'flat' or 'direct-connect' topologies (where all switches connect to servers) perform equally. For example, random graphs have roughly 30% higher throughput than hypercubes at the scale of 512 nodes, and this gap increases with scale [16]. Further, the notion of 'flat' is not even well-defined for heterogeneous networks.

(2) High-throughput heterogeneous network design. We use random graphs as building blocks for heterogeneous network design by first optimizing the volume of connectivity between groups of nodes, and then forming connections randomly within these volume constraints. Specifically, we first show empirically that in this framework, for a set of switches with different port counts but uniform line-speed, attaching servers to switches in proportion to the switch port count is optimal.

Next, we address the interconnection of multiple types of switches. For tractability, we limit our investigation to two switch types. Somewhat surprisingly, we find that a wide range of connectivity arrangements provides nearly identical throughput. A useful consequence of this result is that there is significant opportunity for clustering switches to achieve shorter cable lengths on average, without compromising on throughput. Jellyfish [23] demonstrated this experimentally. Our results provide the theoretical underpinnings of such an approach.

Finally, in the case of multiple line-speeds, we show that complex bottleneck behavior may appear and there may be multiple configurations of equally high capacity.

(3) Applications to real-world network design. The topology proposed in VL2 [13] incorporates heterogeneous line-speeds and port-counts, and has been deployed in Microsoft's cloud data centers.[3] We show that using a combination of the above insights, VL2's throughput can be improved by as much as 43% at the scale of a few thousand servers simply by rewiring existing equipment, with gains increasing with network size.

While a detailed treatment of other related work follows in §2, the **Jellyfish** [23] proposal merits attention here since it is also based on random graphs. Despite this shared ground, Jellyfish does not address either of the central questions addressed by our work: (a) How close to optimal are random graphs for the homogeneous case? and (b) How do we network *heterogeneous* equipment for high throughput? In addition, unlike Jellyfish, by analyzing how network metrics like cut-size, path length, and utilization impact throughput, we attempt to develop an *understanding* of network design.

2 Background and Related Work

High capacity has been a core goal of communication networks since their inception. How that goal manifests in network topology, however, has changed with systems considerations. Wide-area networks are driven by geographic constraints such as the location of cities and railroads. Perhaps the first high-throughput networks not driven by geography came in the early 1900s. To interconnect telephone lines at a single site such as a telephone exchange, *nonblocking* switches were developed which could match inputs to any permutation of outputs. Beginning with the basic crossbar switch which requires $\Theta(n^2)$ size to interconnect n inputs and outputs, these designs were optimized to scale to larger size, culminating with the Clos network developed at Bell Labs in 1953 [8] which constructs a nonblocking interconnect out of $\Theta(n \log n)$ constant-size crossbars.

In the 1980s, supercomputer systems began to reach a scale of parallelism for which the topology connecting compute nodes was critical. Since a packet in a supercomputer is often a low-latency memory reference (as opposed to a relatively heavyweight TCP connection) traversing nodes with tiny forwarding tables, such

[2] For instance, for degree 5 and diameter 4, the best known graph has only 50% of the number of nodes in the best known upper bound [27]. Further, this gap grows larger with both degree and diameter.

[3] Based on personal exchange, and mentioned publicly at http://research.microsoft.com/en-us/um/people/sudipta/.

systems were constrained by the need for very simple, loss-free and deadlock-free routing. As a result the series of designs developed through the 1990s have simple and regular structure, some based on non-blocking Clos networks and others turning to butterfly, hypercube, 3D torus, 2D mesh, and other designs [17].

In commodity compute clusters, increasing parallelism, bandwidth-intensive big data applications and cloud computing have driven a surge in data center network architecture research. An influential 2008 paper of Al-Fares et al. [2] proposed moving from a traditional data center design utilizing expensive core and aggregation switches, to a network built of small components which nevertheless achieved high throughput — a folded Clos or "fat-tree" network. This work was followed by several related designs including Portland [20] and VL2 [13], a design based on small-world networks [21], designs using servers for forwarding [14, 15, 29], and designs incorporating optical switches [12, 26].

Jellyfish [23] demonstrated, however, that Clos networks are sub-optimal. In particular, [23] constructed a random degree-bounded graph among switch-to-switch links, and showed roughly 25% greater throughput than a fat-tree built with the same switch equipment. In addition, [23] showed quantitatively that random networks are easier to incrementally expand — adding equipment simply involves a few random link swaps. Several challenges arise with building a completely unstructured network; [23] demonstrated effective routing and congestion control mechanisms, and showed that cable optimizations for random graphs can make cable costs similar to an optimized fat-tree while still obtaining substantially higher throughput than a fat-tree.

While the literature on homogeneous network design is sizeable, very little is known about heterogeneous topology design, perhaps because earlier supercomputer topologies (which reappeared in many recent data center proposals) were generally constrained to be homogeneous. VL2 [13] provides a point design, using multiple line-speeds and port counts at different layers of its hierarchy; we compare with VL2 later (§7). The only two other proposals that address heterogeneity are LEGUP [11] and REWIRE [10]. LEGUP uses an optimization framework to search for the cheapest Clos network achieving desired network performance. Being restricted to Clos networks impairs LEGUP greatly: Jellyfish achieves the same network expansion as LEGUP at 60% lower cost [23]. REWIRE removes this restriction by using a local-search optimization (over a period of several days of compute time at the scale of 3200 servers) to continually improve upon an initial feasible network. REWIRE's code is not available so a comparison has not been possible. But more fundamentally, all of the above approaches are either point designs [13] or heuristics [10, 11] which by their blackbox nature, provide neither an *understanding* of the solution space, nor any evidence of near-optimality.

3 Simulation Methodology

Our experiments measure the capacity of network topologies. For most of this paper, our goal is to study topologies explicitly independent of systems-level issues such as routing and congestion control. Thus, we model network traffic using fluid splittable flows which are routed optimally. Throughput is then the solution to the standard maximum concurrent multi-commodity flow problem [18]. Note that by maximizing the minimum flow throughput, this model incorporates a strict definition of fairness. We use the CPLEX linear program solver [1] to obtain the maximum flow. Unless otherwise specified, the workload we use is a random permutation traffic matrix, where each server sends traffic to (and receives traffic from) exactly one other server.

In §8, we revisit these assumptions to address systems concerns. We include results for several other traffic matrices besides permutations. We also show that throughput within a few percent of the optimal flow values from CPLEX can be achieved *after* accounting for packet-level routing and congestion control inefficiencies.

Any comparisons between networks are made using identical switching equipment, unless noted otherwise.

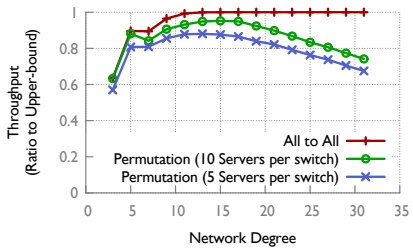
Across all experiments, we test a wide range of parameters, varying the network size, node degree, and oversubscription. A representative sample of results is included here. Most experiments average results across 20 runs, with standard deviations in throughput being $\sim$1% of the mean except at small values of throughput in the uninteresting cases. Exceptions are noted in the text.

Our simulation tools are publicly available [24].

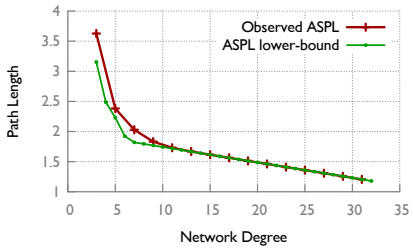
4 Homogeneous Topology Design

In this setting, we have N switches, each with k ports. The network is required to support S servers. The symmetry of the problem suggests that each switch be connected to the same number of servers. (We assume for convenience that S is divisible by N.) Intuitively, spreading servers across switches in a manner that deviates from uniformity will create bottlenecks at the switches with larger numbers of servers. Thus, we assume that each switch uses out of its k ports, r ports to connect to other switches, and $k-r$ ports for servers. It is also assumed that each network edge is of unit capacity.

The design space for such networks is the set of all subgraphs H of the complete graph over N nodes K_N, such that H has degree r. For generic, application-oblivious design, we assume that the objective is to max-

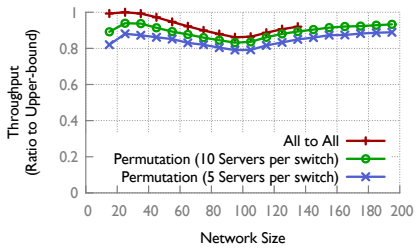


(a)

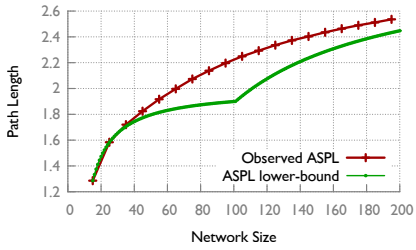


(b)

Figure 1: *Random graphs versus the bounds: (a) Throughput and (b) average shortest path length (ASPL) in random regular graphs compared to the respective upper and lower bounds for any graph of the same size and degree. The number of switches is fixed to* 40 *throughout. The network becomes denser rightward on the x-axis as the degree increases.*



(a)



(b)

Figure 2: *Random graphs versus the bounds: (a) Throughput and (b) average shortest path length (ASPL) in random regular graphs compared to the respective upper and lower bounds for any graph of the same size and degree. The degree is fixed to* 10 *throughout. The network becomes sparser rightward on the x-axis as the number of nodes increases.*

imize throughput under a uniform traffic matrix such as all-to-all traffic or random permutation traffic among servers. To account for fairness, the network's throughput is defined as the maximum value of the minimum flow between source-destination pairs. We denote such a throughput measurement of an r-regular subgraph H of K_N under uniform traffic with f flows by $T_H(N,r,f)$. The average path length of the network is denoted by $\langle D \rangle$.

For this scenario, we prove a simple upper bound on the throughput achievable by *any* hypothetical network.

Theorem 1. $T_H(N,r,f) \leq \frac{Nr}{\langle D \rangle f}$.

Proof. The network has a total of Nr edges (counting both directions) of unit capacity, for a total capacity of Nr. A flow i whose end points are a shortest path distance d_i apart, consumes at least $x_i d_i$ units of capacity in to obtain throughput x_i. Thus, the total capacity consumed by all flows is at least $\sum_i x_i d_i$. Given that we defined network throughput $T_H(N,r,f)$ as the minimum flow throughput, $\forall i, x_i \geq T_H(N,r,f)$. Total capacity consumed is then at least $T_H(N,r,f)\sum_i d_i$. For uniform traffic patterns such as random permutations and all-to-all traffic, $\sum_i d_i = \langle D \rangle f$ because the average source-destination distance is the same as the graph's average shortest path distance. Also, total capacity consumed cannot exceed the network's capacity. Therefore, $\langle D \rangle f T_H(N,r,f) \leq Nr$, rearranging which yields the result. □

Further, [7] proves a lower bound on the average shortest path length of any r-regular network of size N:

$$\langle D \rangle \geq d^* = \frac{\sum_{j=1}^{k-1} jr(r-1)^{j-1} + kR}{N-1}$$

$$\text{where} \quad R = N - 1 - \sum_{j=1}^{k-1} r(r-1)^{j-1} \geq 0$$

and k is the largest integer such that the *inequality* holds.

This result, together with Theorem 1, yields an upper bound on throughput: $T_H(N,r,f) \leq \frac{Nr}{fd^*}$. Next, we show experimentally that random regular graphs achieve throughput close to this bound.

A *random regular graph*, denoted as RRG(N, k, r), is a graph sampled uniform-randomly from the space of all r-regular graphs. This is a well-known construct in graph theory. As Jellyfish [23] showed, RRGs compare favorably against traditional fat-tree topologies, supporting a larger number of servers at full throughput. However, that fact leaves open the possibility that there are network topologies that achieve significantly higher throughput than even RRGs. Through experiments, we compare the throughput RRGs achieve to the upper bound we derived above, and find that our results eliminate this possibility.

Fig. 1(a) and Fig. 2(a) compare throughput achieved by RRGs to the upper bound on throughput for any topology built with the same equipment. Fig. 1(a) shows this comparison for networks of increasing density (*i.e.*, the

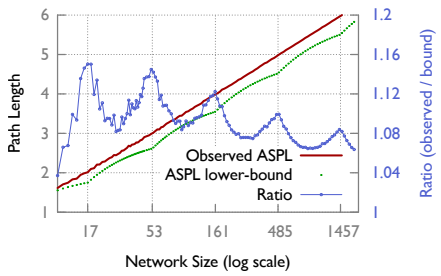


Figure 3: *ASPL in random graphs compared to the lower bound. The degree is fixed to* 4 *throughout. The bound shows a "curved step" behavior. In addition, as the network size increases, the ratio of observed ASPL to the lower bound approaches* 1. *The x-tics correspond to the points where the bound begins new distance levels.*

degree r increases, while the number of nodes N remains fixed at 40) for 3 uniform traffic matrices: a random permutation among servers with 5 servers at each switch, another with 10 servers at each switch, and an all-to-all traffic matrix. For the high-density traffic pattern, *i.e.*, all-to-all traffic, *exact optimal* throughput is achieved by the random graph for degree $r \geq 13$. Fig. 2(a) shows a similar comparison for increasing size N, with $r = 10$. Our simulator does not scale for all-to-all traffic because the number of commodities in the flow problem increases as the square of the network size for this pattern. Fig. 1(b) and 2(b) compare average shortest path length in RRGs to its lower bound. For both large network sizes, and very high network density, RRGs are surprisingly close to the bounds (right side of both figures).

The curve in Fig. 2(b) has two interesting features. First, there is a "curved step" behavior, with the first step at network size up to $N = 101$, and the second step beginning thereafter. To see why this occurs, observe that the bound uses a tree-view of distances from any node — for a network with degree d, d nodes are assumed to be at distance 1, $d(d-1)$ at distance 2, $d(d-1)^2$ at distance 3, etc. While this structure minimizes path lengths, it is optimistic — in general, not all edges from nodes at distance k can lead outward to unique new nodes[4]. As the number of nodes N increases, at some point the lowest level of this hypothetical tree becomes full, and a new level begins. These new nodes are more distant, so average path length suddenly increases more rapidly, corresponding to a new "step" in the bound. A second feature is that as $N \to \infty$, the ratio of observed ASPL to the lower bound approaches 1. This can be shown analytically by dividing an upper bound on the random regular graph's diameter [6] (which also upper-bounds its ASPL) by the lower bound of [7]. For greater clarity, we show in Fig. 3 similar behavior for degree $d = 4$, which makes it easier to show many "steps".

[4]In fact, prior work shows that graphs with this structure do not exist for $d \geq 3$ and diameter $D \geq 3$ [19].

The near-optimality of random graphs demonstrated here leads us to use them as a building block for the more complicated case of heterogeneous topology design.

5 Heterogeneous Topology Design

With the possible exception of a scenario where a new data center is being built from scratch, it is unreasonable to expect deployments to have the same, homogeneous networking equipment. Even in the 'greenfield' setting, networks may potentially use heterogeneous equipment. While our results above show that random graphs achieve close to the best possible throughput in the homogeneous network design setting, we are unable, at present, to make a similar claim for heterogeneous networks, where node degrees and line-speeds may be different. However, in this section, we present for this setting, interesting experimental results which challenge traditional topology design assumptions. Our discussion here is mostly limited to the scenario where there are two kinds of switches in the network; generalizing our results for higher diversity is left to future work.

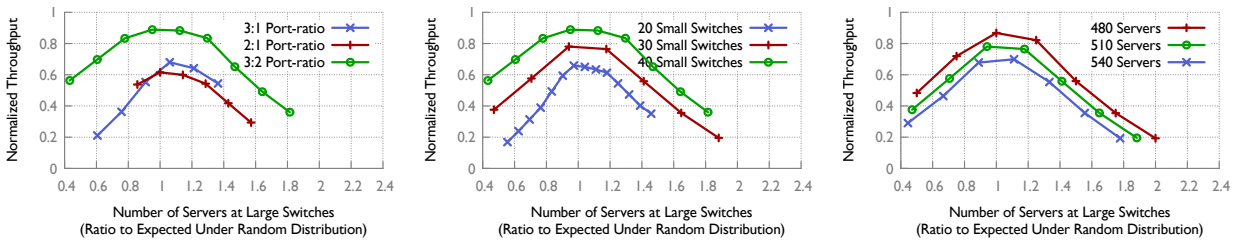
5.1 Heterogeneous Port Counts

We consider a simple scenario where the network is composed of two types of switches with different port counts (line-speeds being uniform throughout). Two natural questions arise that we shall explore here: (a) How should we distribute servers across the two switch types to maximize throughput? (b) Does biasing the topology in favor of more connectivity between larger switches increase throughput?

First, we shall assume that the interconnection is an unbiased random graph built over the remaining connectivity at the switches after we distribute the servers. Later, we shall fix the server distribution but bias the random graph's construction. Finally we will examine the combined effect of varying both parameters at once.

Distributing servers across switches: We vary the numbers of servers apportioned to large and small switches, while keeping the total number of servers and switches the same[5]. We then build a random graph over the ports that remain unused after attaching the servers. We repeat this exercise for several parameter settings, varying the numbers of switches, ports, and servers. A representative sample of results is shown in Fig. 4. The particular configuration in Fig. 4(a) uses 20 larger and 40 smaller switches, with the port counts for the three curves in the figure being 30 and 10 (3:1), 30 and 15 (2:1), and 30 and 20 (3:2) respectively. Fig. 4(b) uses 20 larger switches (30 ports) and 20, 30 and 40 smaller switches

[5]Clearly, across the same type of switches, a non-uniform server-distribution will cause bottlenecks and sub-optimal throughput.



(a) (b) (c)

Figure 4: *Distributing servers across switches: Peak throughput is achieved when servers are distributed proportionally to port counts i.e., x-axis=1, regardless of (a) the absolute port counts of switches; (b) the absolute counts of switches of each type; and (c) oversubscription in the network.*

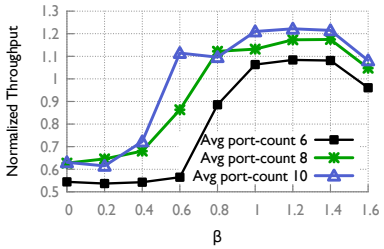


Figure 5: *Distributing servers across switches: Switches have port-counts distributed in a power-law distribution. Servers are distributed in proportion to the β^{th} power of switch port-count. Distributing servers in proportion to degree ($\beta = 1$) is still among the optimal configurations.*

(20 ports) respectively for its three curves. Fig. 4(c) uses the same switching equipment throughout: 20 larger switches (30 ports) and 30 smaller switches (20 ports), with 480, 510, and 540 servers attached to the network. Along the x-axis in each figure, the number of servers apportioned to the larger switches increases. The x-axis label normalizes this number to the *expected* number of servers that would be apportioned to large switches if servers were spread randomly across all the ports in the network. As the results show, distributing servers in proportion to switch degrees (*i.e.*, x-axis$= 1$) is optimal.

This result, while simple, is remarkable in the light of current topology design practices, where top-of-rack switches are the only ones connected directly to servers.

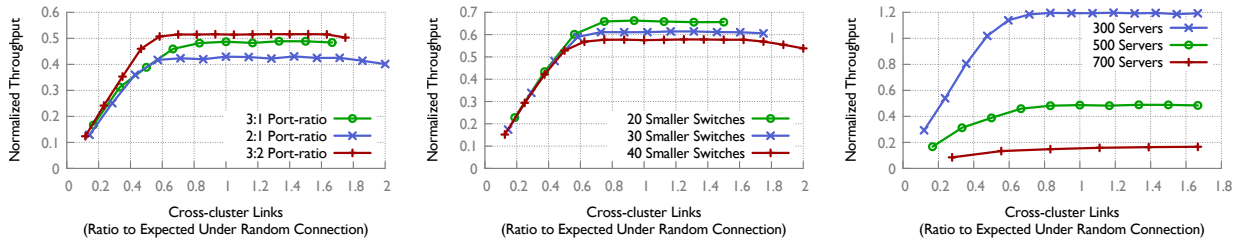
Next, we conduct an experiment with a diverse set of switch types, rather than just two. We use a set of switches such that their port-counts k_i follow a power law distribution. We attach servers at each switch i in proportion to k_i^β, using the remaining ports for the network. The total number of servers is kept constant as we test various values of β. (Appropriate distribution of servers is applied by rounding where necessary to achieve this.) $\beta = 0$ implies that each switch gets the same number of servers regardless of port count, while $\beta = 1$ is the same as port-count-proportional distribution, which was optimal in the previous experiment. The results are shown in Fig. 5. $\beta = 1$ is optimal (within the variance in our data), but so are other values of β such as 1.2 and 1.4. The variation in throughput is large at both extremes of the plot, with the standard deviation being as much as 10% of the mean, while for $\beta \in \{1, 1.2, 1.4\}$ it is $< 4\%$.

Switch interconnection: We repeat experiments similar to the above, but instead of using a uniform random network construction, we vary the number of connections across the two clusters of (large and small) switches[6]. The distribution of servers is fixed throughout to be in proportion to the port counts of the switches.

As Fig. 6 shows, throughput is surprisingly stable across a wide range of volumes of cross-cluster connectivity. x-axis $= 1$ represents the topology with no bias in construction, *i.e.*, vanilla randomness; $x < 1$ means the topology is built with fewer cross-cluster connections than expected with vanilla randomness, etc. Regardless of the absolute values of the parameters, when the interconnect has too few connections across the two clusters, throughput drops significantly. This is perhaps unsurprising – as our experiments in §6.1 will confirm, the cut across the two clusters is the limiting factor for throughput in this regime. What *is* surprising, however, is that across a wide range of cross-cluster connectivity, throughput remains stable at its peak value. Our theoretical analysis in §6.2 will address this behavior.

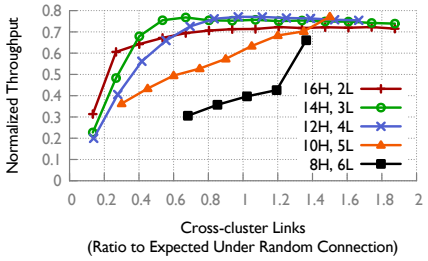
Combined effect: The above results leave open the possibility that joint optimization across the two parameters (server placement and switch connectivity pattern) can yield better results. Thus, we experimented with varying both parameters simultaneously as well. Two representative results from such experiments are included here. All the data points in Fig. 7(a) use the same switching equipment and the same number of servers. Fig. 7(b), likewise, uses a different set of equipment. Each curve in these figures represents a particular distribution of servers. For instance, '16H, 2L' has 16 servers attached to each larger

[6]Note that specifying connectivity across the clusters automatically restricts the remaining connectivity to be within each cluster.

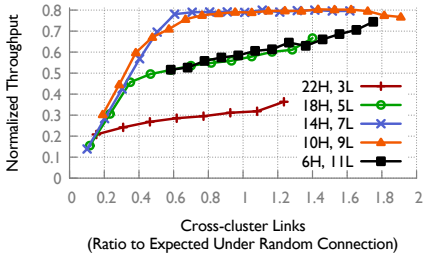


(a) (b) (c)

Figure 6: *Interconnecting switches: Peak throughput is stable to a wide range of cross-cluster connectivity, regardless of (a) the absolute port counts of switches; (b) the absolute counts of switches of each type; and (c) oversubscription in the network.*



(a)



(b)

Figure 7: *Combined effect of server distribution and cross-cluster connectivity: Multiple configurations are optimal, but proportional server distribution with a vanilla random interconnect is among them. (a)* 20 *large,* 40 *small switches, with* 30 *and* 10 *ports respectively. (b)* 20 *large,* 40 *small switches, with* 30 *and* 20 *ports respectively. Results from* 10 *runs.*

switch and 2 to each of the smaller ones. On the *x*-axis, we again vary the cross-cluster connectivity (as in Fig. 6(a)). As the results show, while there are indeed multiple parameter values which achieve peak throughput, a combination of distributing servers proportionally (corresponding to '12H, 4L' and '14H, 7L' respectively in the two figures) and using a vanilla random interconnect is among the optimal solutions. Large deviations from these parameter settings lead to lower throughput.

5.2 Heterogeneous Line-speeds

Data center switches often have ports of different line-speeds, *e.g.*, tens of 1GbE ports, with a few 10GbE ports. How does this change the above analysis change?

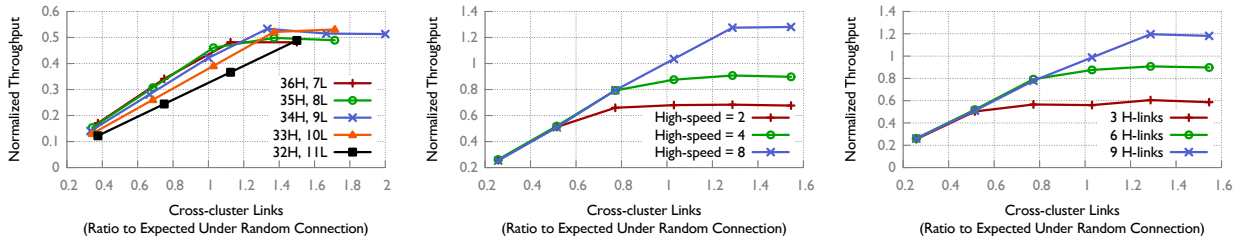
To answer this question, we modify our scenario such that the small switches still have only low line-speed ports, while the larger switches have both low line-speed ports and high line-speed ports. The high line-speed ports are assumed to connect only to other high line-speed ports. We vary both the server distribution and the cross-cluster connectivity and evaluate these configurations for throughput. As the results in Fig. 8(a) indicate, the picture is not as clear as before, with multiple configurations having nearly the same throughput. Each curve corresponds to one particular distribution of servers across switches. For instance, '36H, 7L' has 36 servers attached to each large switch, and 7 servers attached to each small switch. The total number of servers across all curves is constant. While we are unable to make clear qualitative claims of the nature we made for scenarios with uniform line-speed, our simulation tool can be used to determine the optimal configuration for such scenarios.

We also investigate the impact of the number and the line-speed of the high line-speed ports on the large switches. For these tests, we fix the server distribution, and vary cross-cluster connectivity. We measure throughput for various 'high' line-speeds (Fig. 8(b)) and numbers of high line-speed links (Fig. 8(c)). While higher number or line-speed does increase throughput, its impact diminishes when cross-cluster connectivity is too small. This is expected: as the bottlenecks move to the cross-cluster edges, having high capacity between the large switches does not increase the *minimum* flow.

In the following, we attempt to add more than just the intuition for our results. We seek to explain throughput behavior by analyzing factors such as bottlenecks, total network utilization, shortest path lengths between nodes, and the path lengths actually used by the network flows.

6 Explaining Throughput Results

We investigate the cause of several of the throughput effects we observed in the previous section. First, in §6.1,



(a) (b) (c)

Figure 8: *Throughput variations with the amount of cross-cluster connectivity: (a) various server distributions for a network with* 20 *large and* 20 *small switches, with* 40 *and* 15 *low line-speed ports respectively, with the large switches having* 3 *additional* 10× *capacity connections; (b) with different line-speeds for the high-speed links keeping their count fixed at* 6 *per large switch; and (c) with different numbers of the high-speed links at the big switches, keeping their line-speed fixed at* 4 *units.*

we break down throughput into component factors — network utilization, shortest path length, and "stretch" in paths — and show that the majority of the throughput changes are a result of changes in utilization, though for the case of varying server placement, path lengths are a contributing factor. Note that a decrease in utilization corresponds to a saturated bottleneck in the network.

Second, in §6.2, we explain in detail the surprisingly stable throughput observed over a wide range of amounts of connectivity between low- and high-degree switches. We give an upper bound on throughput, show that it is empirically quite accurate in the case of uniform line-speeds, and give a lower bound that matches within a constant factor for a restricted class of graphs. We show that throughput in this setting is well-described by two regimes: (1) one where throughput is limited by a sparse cut, and (2) a "plateau" where throughput depends on two topological properties: total volume of connectivity and average path length $\langle D \rangle$. The transition between the regimes occurs when the sparsest cut has a fraction $\Theta(1/\langle D \rangle)$ of the network's total connectivity.

Note that bisection bandwidth, a commonly-used measure of network capacity which is equivalent to the sparsest cut in this case, begins falling as soon as the cut between two equal-sized groups of switches has less than $\frac{1}{2}$ the network connectivity. Thus, our results demonstrate (among other things) that bisection bandwidth is not a good measure of performance[7], since it begins falling asymptotically far away from the true point at which throughput begins to drop.

6.1 Experiments

Throughput can be exactly decomposed as the product of four factors:

$$T = \frac{C \cdot U}{\langle D \rangle \cdot AS} = C \cdot U \cdot \frac{1}{\langle D \rangle} \cdot \frac{1}{AS}$$

where C is the total network capacity, U is the average link utilization, $\langle D \rangle$ is the average shortest path length, and AS is the average stretch, i.e., the ratio between average length of routed flow paths[8] and $\langle D \rangle$. Throughput may change due to any one of these factors. For example, even if utilization is 100%, throughput could improve if rewiring links reduces path length (this explained the random graph's improvement over the fat-tree in [23]). On the other hand, even with very low $\langle D \rangle$, utilization and therefore throughput will fall if there is a bottleneck in the network.

We investigate how each of these factors influences throughput (excluding C which is fixed). Fig. 9 shows throughput (T), utilization (U), inverse shortest path length ($1/\langle D \rangle$), and inverse stretch ($1/AS$). An increase in any of these quantities increases throughput. To ease visualization, for each metric, we normalize its value with respect to its value when the throughput is highest so that quantities are unitless and easy to compare.

Across experiments, our results (Fig. 9) show that high utilization best explains high throughput. Fig. 9(a) analyzes the throughput results for '480 Servers' from Fig. 4(c), Fig. 9(b) corresponds to '500 Servers' in Fig. 6(c), and Fig. 9(c) to '3 H-links' in Fig. 8(c). Note that it is not obvious that this should be the case: Network utilization would also be high if the flows took long paths and used capacity wastefully. At the same time, one could reasonably expect 'Inverse Stretch' to also correlate with throughput well — if the paths used are close to shortest, then the flows are not wasting capacity. Path lengths do play a role — for example, the right end of Fig. 9(a) shows an increase in path lengths, explaining why throughput falls about 25% more than utilization falls — but the role is less prominent than utilization.

Given the above result on utilization, we examined where in the network the corresponding bottlenecks occur. From our linear program solver, we are able to obtain the link utilization for each network link. We

[7] This result is explored further in followup work [16], where we point out problems with bisection bandwidth as a performance metric.

[8] This average is weighted by amount of flow along each route.

averaged link utilization for each link type in a given network and flow scenario *i.e.*, computing average utilization across links between small and large switches, links between small switches only, etc. The movement of under-utilized links and bottlenecks shows clear correspondence to our throughput results. For instance, for Fig. 6(c), as we move leftward along the x-axis, the number of links across the clusters decreases, and we can expect bottlenecks to manifest at these links. This is exactly what the results show. For example, for the leftmost point ($x = 1.67$, $y = 1.67$) on the '500 Servers' curve in Fig. 6(c), links inside the large switch cluster are on average $< 20\%$ utilized while the links between across clusters are close to fully utilized ($> 90\%$ on average). On the other hand, for the points with higher throughput, like ($x = 1$, $y = 0.49$), all network links show uniformly high utilization ($\sim$100%). Similar observations hold across all our experiments.

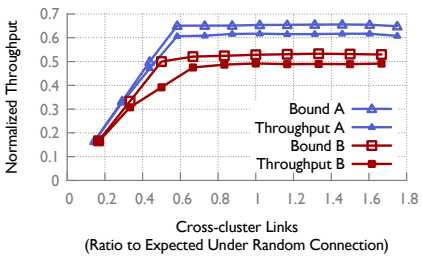
6.2 Analysis

Fig. 6 shows a surprising result: network throughput is stable at its peak value for a wide range of cross-cluster connectivity. In this section, we provide upper and lower bounds on throughput to explain the result. Our upper bound is empirically quite close to the observed throughput in the case of networks with uniform line-speed. Our lower bound applies to a simplified network model and matches the upper bound within a constant factor. This analysis allows us to identify the point (*i.e.*, amount of cross-cluster connectivity) where throughput begins to drop, so that our topologies can avoid this regime, while allowing flexibility in the interconnect.

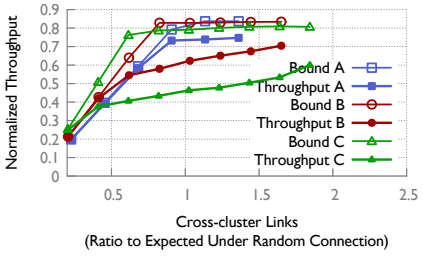
Upper-bounding throughput. We will assume the network is composed of two "clusters", which are simply arbitrary sets of switches, with n_1 and n_2 attached servers respectively. Let C be the sum of the capacities of all links in the network (counting each direction separately), and let $\bar{C}$ be that of the links crossing the clusters. To simplify this exposition, we will assume the number of flows crossing between clusters is exactly the expected number for random permutation traffic: $n_1 \frac{n_2}{n_1+n_2} + n_2 \frac{n_1}{n_1+n_2} = \frac{2n_1n_2}{n_1+n_2}$. Without this assumption, the bounds hold for random permutation traffic with an asymptotically insignificant additive error.

Our upper bound has two components. First, recall our path-length-based bound from §4 shows the throughput of the minimal-throughput flow is $T \leq \frac{C}{\langle D \rangle f}$ where $\langle D \rangle$ is the average shortest path length and f is the number of flows. For random permutation traffic, $f = n_1 + n_2$.

Second, we employ a cut-based bound. The cross-cluster flow is $\geq T \frac{2n_1n_2}{n_1+n_2}$. This flow is bounded above by the capacity $\bar{C}$ of the cut that separates the clusters, so we must have $T \leq \bar{C} \frac{n_1+n_2}{2n_1n_2}$.



(a)



(b)

Figure 10: *Our analytical throughput bound from Eqn. 1 is close to the observed throughput for the uniform line-speed scenario (a) for which the bound and the corresponding throughput are shown for two representative configurations A and B, but can be quite loose with non-uniform line-speeds (b).*

Combining the above two upper bounds, we have

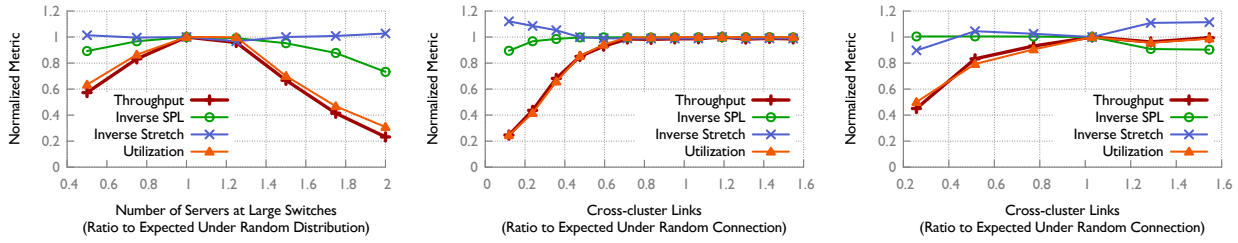
$$T \leq min \left\{ \frac{C}{\langle D \rangle (n_1+n_2)}, \frac{\bar{C}(n_1+n_2)}{2n_1n_2} \right\} \qquad (1)$$

Fig. 10 compares this bound to the actual observed throughput for two cases with uniform line-speed (Fig. 10(a)) and a few cases with mixed line-speeds (Fig. 10(b)). The bound is quite close for the uniform line-speed setting, both for the cases presented here and several other experiments we conducted, but can be looser for mixed line-speeds.

The above throughput bound begins to drop when the cut-bound begins to dominate. In the special case that the two clusters have equal size, this point occurs when

$$\bar{C} \leq \frac{C}{2\langle D \rangle}. \qquad (2)$$

A drop in throughput when the cut capacity is inversely proportional to average shortest path length has an intuitive explanation. In a random graph, most flows have many shortest or nearly-shortest paths. Some flows might cross the cluster boundary once, others might cross back and forth many times. In a uniform-random graph with large $\bar{C}$, near-optimal flow routing is possible with any of these route choices. As $\bar{C}$ diminishes, this flexibility means we can place some restriction on the choice of routes without impacting the flow. However, the flows



(a) (b) (c)

Figure 9: *The dependence of throughput on all three relevant factors: inverse path length, inverse stretch, and utilization. Across experiments, total utilization best explains throughput, indicating that bottlenecks govern throughput.*

which cross clusters must still utilize at least one cross-cluster hop, which is on average a fraction $1/\langle D\rangle$ of their hops. Therefore in expectation, since $\frac{1}{2}$ of all (random-permutation) flows cross clusters, at least a fraction $\frac{1}{2\langle D\rangle}$ of the total traffic volume will be cross-cluster. We should therefore expect throughout to diminish once less than this fraction of the total capacity is available across the cut, which recovers the bound of Equation 2.

However, while Equation 2 determines when the *upper bound* on throughput drops, it does not bound the point at which *observed* throughput drops: since the upper bound might be loose, throughput may drop earlier or later. However, given a peak throughput value, we can construct a bound based on it. Say the peak throughput in a configuration is T^*. $T^* \leq \bar{C}\frac{n_1+n_2}{2n_1n_2}$ implies throughput must drop below T^* when $\bar{C}$ is less than $C^* := T^*\frac{2n_1n_2}{n_1+n_2}$. If we are able to empirically estimate T^* (which is not unreasonable, given its stability), we can determine the value of $\bar{C}^*$ below which throughput *must* drop.

Fig. 11 has 18 different configurations with two clusters with increasing cross-cluster connectivity (equivalently, $\bar{C}$). The one point marked on each curve corresponds to the $\bar{C}^*$ threshold calculated above. As predicted, below $\bar{C}^*$, throughput is less than its peak value.

Lower-bounding throughput. For a restricted class of random graphs, we can lower-bound throughput as well, and thus show that our throughput bound (Eqn. 1), and the drop point of Eqn. 2, are tight within constant factors.

We restrict this analysis to networks $G = (V,E)$ with n nodes each with constant degree d. All links have unit capacity in each direction. The vertices V are grouped into two equal size clusters V_1, V_2, i.e., $|V_1| = |V_2| = \frac{1}{2}n$. Let p, n be such that each node has pn neighbors within its cluster and qn neighbors in the other cluster, so that $p+q = d/n = \Theta(1/n)$. Under this constraint, we choose the remaining graph from the uniform distribution on all d-regular graphs. Thus, for each of the graphs under consideration, the total inter-cluster connectivity is $\bar{C} = 2q\cdot|V_1|\cdot|V_2| = q\cdot\frac{n^2}{2}$. Decreasing q corresponds to decreasing the cross-cluster connectivity and increasing

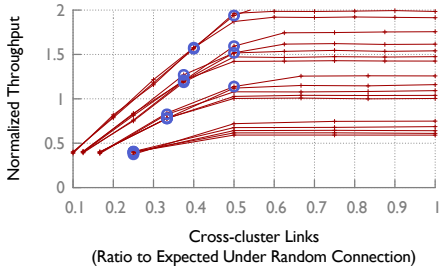


Figure 11: *Throughput shows a characteristics profile with respect to varying levels of cross-cluster connectivity. The one point marked on each curve indicates our analytically determined threshold of cross-cluster connectivity below which throughput must be smaller than its peak value.*

the connectivity within each cluster. Our result below holds with high probability (w.h.p.) over the random choice of the graph. Let $T(q)$ be the throughput with the given value of q, and let T^* be the throughput when $p = q$ (which will also be the maximum throughput).

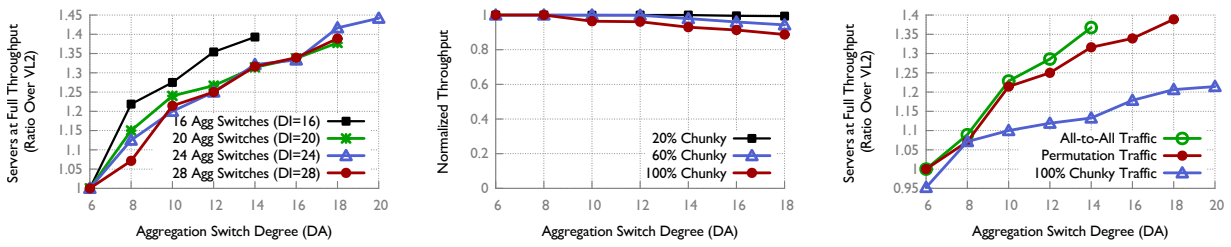
Our main result is the following theorem, which explains the throughput results by proving that while $q \geq q^*$, for some value q^* that we determine, the throughput $T(q)$ is within a constant factor of T^*. Further, when $q < q^*$, $T(q)$ decreases roughly linearly with q. We refer the reader to our technical report [22] for the proof.

Theorem 2. *There exist constants c_1, c_2 such that if $q^* = c_1\frac{1}{\langle D\rangle}p$, then for $q \geq q^*$ w.h.p. $T(q) \geq c_2T^*$. For $q < q^*$, $T(q) = \Theta(q)$.*

7 Improving VL2

In this section, we apply the lessons learned from our experiments and analysis to improve upon a real world topology. Our case study uses the VL2 [13] topology deployed in Microsoft's data centers. VL2 incorporates heterogeneous line-speeds and port-counts and thus provides a good opportunity for us to test our design ideas.

VL2 Background: VL2 [13] uses three types of switches: top-of-racks (ToRs), aggregation switches, and



(a) (b) (c)

Figure 12: *Improving VL2: (a) The number of servers our topology supports in comparison to VL2 by rewiring the same equipment; (b) Throughput under various chunky traffic patterns; and (c) The number of servers our topology can support in comparison to VL2 when we require it to achieve full throughput for all-to-all traffic, permutation traffic, and chunky traffic.*

core switches. Each ToR is connected to 20 1GbE servers, and has 2 10GbE uplinks to different aggregation switches. The rest of the topology is a full bipartite interconnection between the core and aggregation switches. If aggregation switches have *DA* ports each, and core switches have *DI* ports each, then such a topology supports $\frac{DA.DI}{4}$ ToRs at full throughput.

Rewiring VL2: As results in §5.1 indicate, connecting ToRs to only aggregation switches, instead of distributing their connectivity across all switches is sub-optimal. Further, the results on the optimality of random graphs in §4 imply further gains from using randomness in the interconnect as opposed to VL2's complete bipartite interconnect. In line with these observations, our experiments show significant gains obtained by modifying VL2.

In modifying VL2, we distribute the ToRs over aggregation and core switches in proportion to their degrees. We connect the remaining ports uniform randomly. To measure our improvement, we calculate the number of ToRs our topology can support at full throughput compared to VL2. By 'supporting at full throughput', we mean observing full 1 Gbps throughput for each flow in random permutation traffic across each of 20 runs. We obtain the largest number of ToRs supported at full throughput by doing a binary search. As Fig. 12(a) shows, we gain as much as a 43% improvement in the number of ToRs (equivalently, servers) supported at full throughput at the largest size. Note that the largest size we evaluated is fairly small – 2,400 servers for VL2 – and our improvement increases with the network's size.

8 In Practice

In this section, we address two practical concerns: (a) performance with a more diverse set of traffic matrices beyond the random permutations we have used so far; and (b) translating the flow model to packet-level throughput without changing the results significantly.

8.1 Other Traffic Matrices

We evaluate the throughput of our VL2-like topology under other traffic matrices besides random permutations. For these experiments, we use the topologies corresponding to the '28 Agg Switches (*DI*=28)' curve in Fig. 12(a). (Thus, by design, the throughput for random permutations is expected, and verified, to be 1.) In addition to the random permutation, we test the following other traffic matrices: (a) All-to-all: where each server communicates with every other server; and (b) *x*% Chunky: where each of *x*% of the network's ToRs sends all of its traffic to *one* other ToR in this set (*i.e.*, a ToR-level permutation), while the remaining $(100-x)$% ToRs are engaged in a server-level random permutation workload among themselves.

Our experiments showed that using the network to interconnect the same number of servers as in our earlier tests with random permutation traffic, full throughput is still achieved for all but the chunky traffic pattern. In Fig. 12(b), we present results for 5 chunky patterns. Except when a majority of the network is engaged in the chunky pattern, throughput is within a few percent of full throughput. We note that 100% Chunky is a hard to route traffic pattern which is easy to avoid. Even assigning applications to servers randomly will ensure that the probability of such a pattern is near-zero.

Even so, we repeat the experiment from Fig. 12(a) where we had measured the number of servers our modified topology supports at full throughput under random permutations. In this instance, we require our topology to support full throughput under the 100% Chunky traffic pattern. The results in Fig. 12(c) show that the gains are smaller, but still significant, 22% at the largest size, and increasing with size. It is also noteworthy that all-to-all traffic is easier to route than both the other workloads.

8.2 From Flows to Packets

Following the method used by Jellyfish [23], we use Multipath TCP (MPTCP [28]) in a packet level simulation to test if the throughput of our modified VL2-like

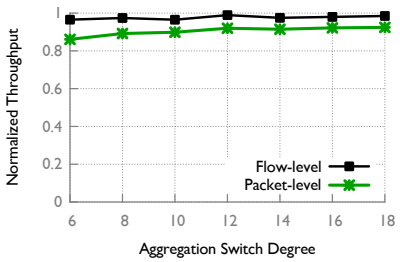


Figure 13: *Packet level simulations of random permutation traffic over our topology show that throughput within a few percent of the optimal flow-level throughput can be achieved using MPTCP over the shortest paths.*

topology is similar to what flow simulations yield. We use MPTCP with the shortest paths, using as many as 8 MPTCP subflows. The results in Fig. 13 show that throughput within a few percent (6% gap at the largest size) of the flow-level simulations is achievable. Note that we deliberately oversubscribed the topologies so that the flow value was close to, but less than 1. This makes sure that we measure the gap between the flow and packet models accurately — if the topology is overprovisioned, then even inefficient routing and congestion control may possibly yield close to full throughput.

9 Discussion

Why these traffic matrices? In line with the design objective of hosting arbitrary applications at high throughput, the approach we have taken is to study *difficult* traffic matrices, rather than TMs specific to particular environments. We show in [16] that an all-to-all workload bounds performance under any workload within a factor of 2. As such, testing this TM is more useful than any other specific, arbitrary choice. In addition, we evaluate other traffic matrices which are even harder to route (Fig. 12(c)). Further, our code is available [24], and is easy to augment with arbitrary traffic patterns to test.

What about latency? We include a rigorous analysis of latency in terms of path length (Fig. 1(b), 2(b)), showing that average shortest path lengths are close to optimal in random graphs. Further, Jellyfish [23] showed that even worst-case path length (diameter) in random graphs is smaller than or similar to that in fat-trees. Beyond path length, latency depends on the transport protocol's ability to keep queues small. In this regard, we note that techniques being developed for low latency transport (such as DCTCP [3], HULL [4], pFabric [5]) are topology agnostic.

But randomness?! 'Random' $\not\Rightarrow$ 'inconsistent performance': the standard deviations in throughput are $\sim$1% of the mean (and even smaller for path length). Also, by 'maximizing the minimum flow' to measure throughput, we impose a strict definition of fairness, eliminating the possibility of randomness skewing results across flows. Further, Jellyfish [23] showed that random graphs achieve flow-fairness comparable to fat-trees under a practical routing scheme. Simple and effective physical cabling methods were also shown in [23].

Limitations: While we have presented here foundational results on the design of both homogeneous and heterogeneous topologies, many interesting problems remain unresolved, including: (a) a non-trivial upper bound on the throughput of heterogeneous networks; (b) theoretical support for our §5.1 result on server distribution; and (c) generalizing our results to arbitrarily diverse networks with multiple switch types.

Lastly, we note that this work does not incorporate functional constraints such as those imposed by middleboxes, for instance, in its treatment of topology design.

10 Conclusion

Our result on the near-optimality of random graphs for homogeneous network design implies that homogeneous topology design may be reaching its limits, particularly when uniformly high throughput is desirable. The research community should perhaps focus its efforts on other aspects of the problem, such as joint optimization with cabling, or topology design for specific traffic patterns (or bringing to practice research proposals on the use of wireless and/or optics for flexible networks that adjust to traffic patterns), or improvements to heterogeneous network design beyond ours.

Our work also presents the first systematic approach to the design of heterogeneous networks, allowing us to improve upon a deployed data center topology by as much as 43% even at the scale of just a few thousand servers, with this improvement increasing with size. In addition, we further the understanding of network throughput by showing how cut-size, path length, and utilization affect throughput.

While significant work remains in the space of designing and analyzing topologies, this work takes the first steps away from the myriad point solutions and towards a theoretically grounded approach to the problem.

Acknowledgments

We would like to thank our shepherd Walter Willinger and the anonymous reviewers for their valuable suggestions. We gratefully acknowledge the support of Cisco Research Council Grant 573665. Ankit Singla was supported by a Google PhD Fellowship.

References

[1] CPLEX Linear Program Solver. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

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

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. DCTCP: Efficient packet transport for the commoditized data center. In *SIGCOMM*, 2010.

[4] 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. *NSDI*, 2012.

[5] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing datacenter packet transport. *HotNets*, 2012.

[6] B. Bollobás and W. F. de la Vega. The diameter of random regular graphs. In *Combinatorica 2*, 1981.

[7] V. G. Cerf, D. D. Cowan, R. C. Mullin, and R. G. Stanton. A lower bound on the average shortest path length in regular graphs. *Networks*, 1974.

[8] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[9] F. Comellas and C. Delorme. The (degree, diameter) problem for graphs. http://maite71.upc.es/grup_de_grafs/table_g.html/.

[10] A. R. Curtis, T. Carpenter, M. Elsheikh, A. Lopez-Ortiz, and S. Keshav. Rewire: An optimization-based framework for unstructured data center network design. In *INFOCOM*, 2012.

[11] A. R. Curtis, S. Keshav, and A. Lopez-Ortiz. LEGUP: using heterogeneity to reduce the cost of data center network upgrades. In *CoNEXT*, 2010.

[12] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *SIGCOMM*, 2010.

[13] 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.

[14] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, 2009.

[15] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[16] S. A. Jyothi, A. Singla, P. B. Godfrey, and A. Kolla. Measuring and Understanding Throughput of Network Topologies. Technical report, 2014. http://arxiv.org/abs/1402.2531.

[17] F. T. Leighton. Introduction to parallel algorithms and architectures: Arrays, trees, hypercubes. 1991.

[18] T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 1999.

[19] M. Miller and J. Siran. Moore graphs and beyond: A survey of the degree/diameter problem. *ELECTRONIC JOURNAL OF COMBINATORICS*, 2005.

[20] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[21] J.-Y. Shin, B. Wong, and E. G. Sirer. Small-world datacenters. *ACM SOCC*, 2011.

[22] A. Singla, P. B. Godfrey, and A. Kolla. High Throughput Data Center Topology Design. Technical report, 2013. http://arxiv.org/abs/1309.7066.

[23] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Network Data Centers Randomly. In *NSDI*, 2012.

[24] A. Singla, S. A. Jyothi, C.-Y. Hong, L. Popa, P. B. Godfrey, and A. Kolla. TopoBench: A network topology benchmarking tool. https://github.com/ankitsingla/topobench, 2014.

[25] A. Singla, A. Singh, K. Ramachandran, L. Xu, and Y. Zhang. Proteus: a topology malleable data center network. In *HotNets*, 2010.

[26] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan. c-through: Part-time optics in data centers. In *SIGCOMM*, 2010.

[27] C. Wiki. The Degree-Diameter Problem for General Graphs. http://goo.gl/iFRJS.

[28] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI*, 2011.

[29] H. Wu, G. Lu, D. Li, C. Guo, and Y. Zhang. Mdcube: A high performance network structure for modular data center interconnection. In *CoNext*, 2009.

[30] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror mirror on the ceiling: flexible wireless links for data centers. In *SIGCOMM*, 2012.

Adtributor: Revenue Debugging in Advertising Systems

Ranjita Bhagwan, Rahul Kumar, Ramachandran Ramjee, George Varghese,
Surjyakanta Mohapatra, Hemanth Manoharan, and Piyush Shah
Microsoft

Abstract

Advertising (ad) revenue plays a vital role in supporting free websites. When the revenue dips or increases sharply, ad system operators must find and fix the root-cause if actionable, for example, by optimizing infrastructure performance. Such revenue debugging is analogous to diagnosis and root-cause analysis in the systems literature but is more general. Failure of infrastructure elements is *only one* potential cause; a host of other *dimensions* (e.g., advertiser, device type) can be sources of potential causes. Further, the problem is complicated by *derived measures* such as costs-per-click that are also tracked along with revenue.

Our paper takes the first systematic look at revenue debugging. Using the concepts of explanatory power, succinctness, and surprise, we propose a new multi-dimensional root-cause algorithm for fundamental and derived measures of ad systems to identify the dimension mostly likely to blame. Further, we implement the attribution algorithm and a visualization interface in a tool called the Adtributor to help troubleshooters quickly identify potential causes. Based on several case studies on a very large ad system and extensive evaluation, we show that the Adtributor has an accuracy of over 95% and helps cut down troubleshooting time by an order of magnitude.

1 Introduction

Many free websites are supported today by revenue generated through advertisements (ads). Website ads can be of two types, namely, search and display. In the case of a search ad, an end user goes to a publisher website such as bing.com and enters a query phrase. The response to the query is a search results page that may contain one or more ads. If the user clicks on one of these ads, the publisher earns revenue. In the case of a display ad, an end user may visit a publisher website, such as cnn.com, where she might see ads at the top or sides of the page. The display of these ads earns revenue for the publisher.

Ad systems facilitate generation and accounting of millions of such search and display ads every day. Apart from *users* and *publishers* noted above, there are two other key constituents who interact with the ad system. The ads shown to the user are the result of an ad auction between various *advertisers* who bid to compete to have their ad displayed to the user. Also in the midst are various *fraud operators* [8] that try to usurp a fraction of the advertising revenue.

Ad systems manage the interaction between users, publishers, advertisers and fraud operators. Ad systems implement various ad-related algorithms that run the real-time ad auctions between the advertisers, return the winning ads to the publisher, monitor the user clicks, detect and remove potential fraudulent activity, compute the revenue from each displayed or clicked ad, charge the advertiser the appropriate bid amount, and pay the publishers. At the core of the ad system is a large-scale distributed system consisting of thousands of servers distributed across several data centers that execute the ad algorithms and manage the serving and accounting of ads.

The focus of this paper is on debugging ad systems. Typically, an ad system monitor issues an alert whenever a measure of interest is identified as anomalous (e.g., revenue or number of searches is down sharply). [1] Our goal is to automatically identify the potential root cause of this anomaly. We term our approach *revenue debugging*, even though it is applicable to several measures of interest to ad system operators, to acknowledge the prominence of the revenue metric. In this paper, we describe a new revenue debugging algorithm that analyses the large amount of data logged by the ad system and narrows down the scope of potential root-cause of an anomaly to a sub-component of the ad system for further investigation by a human troubleshooter.

Root-cause identification and diagnostics is an age-old problem in systems. Various performance root-causing tools have been proposed in the past [1, 2, 3, 10, 14, 15]. But all these solutions have focused on performance/failure debugging. Here, we address a similar yet more general problem: diagnostics in ad systems. While performance/failure of infrastructure systems components can be one possible root-cause for an anomalous measure, there may be various other root-causes that depend on other components that interact with the ad system. Consider the following examples.

1. Papal Election: We noticed that the papal election caused a revenue drop because many searches were made for non-monetizable query terms such as pope or papal

[1] Anomaly detection is a challenging problem in itself but is out of scope of this paper.

election, that advertisers typically do not bid for. The total number of ads shown dropped which resulted in an anomalous revenue drop. While identifying the root-cause as the papal election is not actionable, root-cause identification is still important as it eliminates an actionable root-cause such as the example below.

2. Browser Ad Failure: We found a revenue drop was caused by a manual error in updating a configuration file that had the side-effect of not showing ads on certain browser versions. In this case, quick identification helped rectify the configuration error, thereby restoring advertising revenue. A more extensive set of examples is depicted in Table 1 and discussed in Section 2.1.

The first challenge in ad systems debugging is sheer scale. There are hundreds of millions of searches and clicks every day; performing diagnostics at the level of a search or a click is not scalable (imagine running Magpie [3] or tracking a string of system calls through hundreds of system components for every click). Thus, for scalability reasons, ad system debugging operates over aggregates of various *measures*. These measures are typically counters aggregated over certain time intervals (e.g., revenue generated over the last 1 hour). Root-cause identification can only be triggered by anomalous behavior of these aggregate counters.

A second distinguishing characteristic of ad systems as compared to typical systems trouble shooting is the existence of multiple *dimensions*, and the need to first isolate the dimension that explains the anomaly. Measures such as revenue can be broken down or projected along different dimensions such as advertiser, browser, or data center. For instance, in Example 2, if revenue were projected along the browser dimension, one could observe that some browser versions were not generating their "typical" share of revenue. However, if the same revenue were sliced by the advertiser dimension, perhaps the distribution of revenue would not have changed significantly.

Typical systems root-causing algorithms such as SCORE [11] use succinctness (Occam's razor) and explanatory power (does the root-cause explain the change?) as their main parameters for optimization and do not have to account for multiple dimensions. To isolate anomalous dimensions, we introduce the notion of *surprise*, captured by quantifying the change in distribution of measure values across each dimension. For instance, in Example 2, change in distribution of revenue along the browser dimension is more surprising than the change in distribution of revenue along the advertiser dimension. Thus, *our first contribution in the paper is the root-causing algorithm described in Section 3* that uses surprise in addition to succinctness and explanatory power to identify root-causes in ad systems.

A third unique characteristic of ad systems is the prevalence of derived measures. Consider two fundamental measures: revenue per hour and number of clicks per hour. From these two measures, one can define a derived measure called cost-per-click that is simply revenue divided by number of clicks. Ad system operators monitor and track many such derived measures that are functions of various fundamental measures (see Figure 1). For example, the change in number of clicks and change in revenue may be small by themselves and not anomalous (e.g., less than 10%). However, *correlated* changes (e.g., revenue drops and simultaneously clicks increase, each by say 10%), are anomalous and is captured by the derived cost-per-click measure (20% change). As we discuss in Section 4, attributing a root-cause to derived measures is challenging. To address this, *we propose a novel partial-derivative inspired attribution solution for derived measures, our second contribution of the paper.*

The outcome of our root-cause identification algorithm is a set of candidates that potentially explain an anomaly. However, this is only the first step in the diagnosis process where a troubleshooter may, if appropriate, take actions to fix the issue. To help the troubleshooter quickly identify potential root-cause candidates, *we have implemented our root-cause identification algorithm and a graphical visualizer in a tool called the* Adtributor*, our third contribution of the paper*. Through experiences from a pilot deployment in a production system, we have refined the visualization interface and data representation techniques in Adtributor to further reduce turnaround time for troubleshooters.

Finally, we perform extensive evaluation of our root-causing algorithm. First, we tabulate and discuss a representative set of case studies that highlight the value of our root-causing tool. Second, we evaluate our algorithm on 128 anomaly alerts over 2 weeks of real ad system data and find that our algorithm achieves an accuracy of over 95%. In fact, Adtributor even found root-causes for a few anomalies that were missed by the manual troubleshooters. Further, the tool also speeds up the troubleshooting process by an order of magnitude.

2 Problem Statement

In this section, after providing a system overview, we show examples of real problems and their root-causes. Next, we state the problem more precisely and motivate our solution.

2.1 System Overview

Figure 1 shows a simplified representation of an ad system, and the entities such as users, fraud operators, publishers and advertisers that directly interact with the ad system. The ad system itself has various subcomponents, some of which we show.

While the logging infrastructure does track each

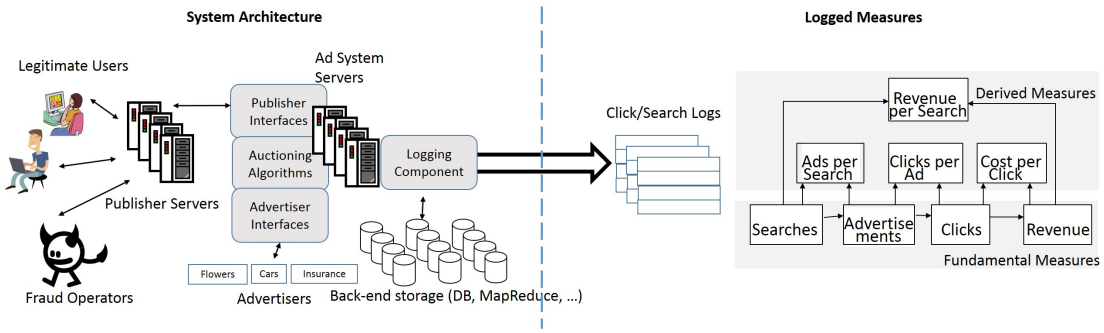


Figure 1: **A simplified representation of an ad system, and the measures it monitors.**

search request or ad-click, the sheer scale makes it hard to track down a problem at the individual request level. Instead, the system monitors a set of *aggregate measures*, as shown in Figure 1. From the raw logs, it first calculates, for each time interval, total searches received, total ads shown, total ad-clicks received, and total revenue from these clicks. These measures are all additive, and can be sliced along different dimensions. For instance, the total revenue is the sum of the revenue made from each advertiser using the system. The total revenue is also the sum of revenue received from different geographical regions where the ad system is active. We term such additive measures *fundamental measures*.

Additionally, the system also monitors a set of non-additive *derived measures*, which are functions of fundamental measures, such as ads-per-search (ads/searches), clicks-per-ad (clicks/ads), cost-per-click (revenue/clicks), and revenue-per-search (revenue/searches).

An anomalous rise or drop in any of these measures is an indication of a problem. Therefore, a diagnostic engine needs to first detect an anomaly, and then perform root-cause analysis. In this paper, we focus on the latter aspect of root-causing, while relying on well-known ARMA model-based methods [4] for anomaly detection. The anomaly detector generates a model-based prediction of measure values based on 8 weeks of historical data, taking into account normal time-of-day and day-of-week fluctuations. It then compares the actual value with the forecasted value – when the actual value of a measure is significantly different from the forecasted value, it generates an anomaly alert. The threshold difference above which we generate an alert is measured in terms of a percentage deviation from the expected value. In the current system, troubleshooters manually set this value based on experience. For each alert, our objective is to attribute the anomaly in a measure to a *dimension* and its corresponding *elements*. We define these terms next.

Dimension: A dimension is an axis along which a measure can be projected. For instance, we can project revenue along the axis of advertisers, and determine how much revenue comes in from each advertiser. The dimension in this case is "Advertiser". Derived measures can be similarly projected across dimensions. Some other dimensions are "Publisher", "Data Center", and "User Location". Typically, an ad system deals with dozens of such dimensions. Note that a dollar of revenue could be added to Advertiser 1 in one dimension, and Publisher 3 in a second dimension.

Element: Every dimension has a domain of values called elements. For instance, the "Advertiser" domain can have the following elements: {*Geico, Microsoft, Toyota, Frito-Lay, ...*}. The Publisher dimension may have elements: {*Bing, Amazon, NetFlix, ...*}.

Table 1 provides a number of problem examples we encountered, both actionable and not actionable, that need to be detected and root-caused to the appropriate dimensions and elements. Column 1 shows that problems can happen at various levels. Column 3 shows the anomalous measure. Column 4 shows the output of the root-cause analysis, the focus of this paper.

Note that Column 4 is only the first step towards root-causing, but it is essential as it gives the troubleshooter the best indication of where the problem actually lies. Other post-processing techniques (correlation engines, NLP techniques, manual investigation) use the output of the multi-dimensional analysis to perform a deeper dive into the issue to arrive at the final root-cause, shown in Column 5, but this aspect of root-causing is outside the scope of this paper. For instance, in row 9, while the multi-dimensional analysis did narrow down the problem to a few query strings, an administrator had to semantically interpret the strings to determine that the papal election was the cause.

2.2 Problem Definition and Scope

The multi-dimensional analysis problem of revenue debugging is to find the dimension and its elements that best explain an anomalous rise or fall in a measure. In this context, we need to define what constitutes the "best explanation" for an anomaly.

Consider the following example. The revenue of an ad system was forecasted to be $100 at a given time. In real-

Category	No.	Symptom	Faulty Dimension and Elements	Final Root-Cause
Infrastructure	1.	Ads shown dropped	Data Center: DC1	Deployment of certain updates to data center DC1 failed.
	2.	Revenue dropped	Log Server: L10, L11, L12	Bug caused abnormally large logs on these logging servers, and they went out of storage.
Ad System	3.	Revenue increased	Bucket: B1, B2	Buckets are A/B tests that are run on disjoint subsets of traffic to test new algorithms. Buckets B1 and B2 were using a different algorithm that increased the number of ads they showed.
	4.	Ads, revenue dropped	Browser: WB1	Configuration file error caused no ads to be shown to users who used web browser WB1. See Section 6.
Advertiser	5.	Cost-per-click, revenue increased	Advertiser: A1, A2, ..., An	These advertisers were all retail companies who increased their budgets during the holiday shopping season. This caused auction prices to go up, thereby increasing cost-per-click and revenue. See Section 6.
	6.	Cost-per-click dropped	Advertiser: Ax	A large advertiser Ax reduced their marketing budget drastically. This caused an overall drop in revenue, and clicks on ads from this advertiser. This made the cost-per-click drop anomalously.
Publisher	7.	Clicks-per-ad increased	Publisher: P1	One publisher launched a new UI with more ads shown on the top of the page than on the side. Users tend to click more on ads at the top of the page, and so this publisher reported more ad-clicks. See Section 6.
	8.	Revenue dropped	Publisher: P2, P3	Publishers P2 and P3 started blocking ads returned by the ad system to make for a cleaner UI. Their revenue dropped.
User	9.	Ads-per-search dropped	Query string: "pope", "papal election"	During the papal election, users searched for "Pope", "Papal election", etc. which are non-monetizable searches. These searches showed no ads, consequently the derived measure ads-per-search dropped.
	10.	Revenue dropped	User Location: New Orleans	A hurricane in New Orleans caused fewer searches from the affected geographical areas.
Fraud	11.	Searches increased	User-agent String	A large number of searches used an identical user-agent string. This was traced to a bot that was spoofing search requests and blindly replicating the user-agent string. See Section 6.

Table 1: **Some example issues that cause anomalies in advertising system measures.**

Data Center	Forecasted Revenue	Actual Revenue	Difference
X	$94	$47	$47
Y	$6	$3	$3
Total	$100	$50	$50

Table 2: **Revenue by Data Center**

Device Type	Forecasted Revenue	Actual Revenue	Difference
A1	$50	$24	$26
A2	$20	$21	-$1
A3	$20	$4	$16
A4	$10	$1	$9
Total	$100	$50	$50

Table 3: **Revenue by Advertiser**

Device Type	Forecasted Revenue	Actual Revenue	Difference
PC	$50	$49	$1
Mobile	$25	$1	$24
Tablet	$25	$0	$25
Total	$100	$50	$50

Table 4: **Revenue by Device Type**

ity, the actual revenue was only $50. An alert is triggered on the revenue measure, which brings a troubleshooters attention to the problem.

To find the root-cause when such problems occur, the ad system continuously tracks the revenue generated across a host of dimensions. For this scenario, consider three such dimensions: Data center (DC), Advertiser (AD), and Device type (DT). Tables 2, 3, 4 show the projection of revenue values along these dimensions, and the values attributed to the individual elements.

We now explain the semantics of these attributions. When the ad system receives a search query, it routes the query to a data center that in turn serves a number of ads in response. The revenue attributed to a data center is the total revenue received from clicks on ads that this data center serves. Each ad has an associated advertiser. When a user clicks an ad, the system charges the advertiser a pre-determined sum of money. The revenue attributed to the advertiser is the total cost of all such clicks made on the advertiser's ads. Users make search queries using a host of devices, which could be phones, tablets, or PCs. The revenue attributed to a device type is the sum total of all revenue that the ad system obtains from ad-clicks from that specific device-type.

The question that we seek to answer is: how do we pinpoint the revenue drop to the right dimensions and their elements? We restate the problem as follows:

"Find a Boolean expression, in terms of dimensions and their elements, such that the revenue drop attributed to the expression best explains the total drop in revenue."

While we examine how to determine "best" shortly, consider the following expressions that could explain the $50 revenue drop:

$$Revenue\_Drop(DC == X) = \$47 \quad (1)$$

$$Revenue\_Drop(AD == A1 \vee AD == A3 \vee AD == A4) = \$51 \quad (2)$$

$$Revenue\_Drop(DT == Mobile \vee DT == Tablet) = \$49 \quad (3)$$

For example, equation 2 states that the sum of the differences between the forecasted and actual revenues for rows 1, 3, and 4 of the advertiser table is $51, which is very close to the total revenue drop of $50.

In general, such expressions could include multiple

dimensions such as $Revenue\_Drop$ $(DT == PC \wedge DC == X)$ which refers to a revenue drop across PC users served ads from data center X. Based on about one year of monitoring alerts in ad systems we have observed, through manual study as well as through using an attribution algorithm that blames anomalies on multiple dimensions, that such cases where multiple dimensions contribute together to a root-cause are very rare. Therefore, for simplicity of exposition, in this paper, we limit our discussions to finding a Boolean expression that involves a single dimension and a set of its elements that explains the anomalous change.

To understand what constitutes the *"best"* dimension and a set of its elements, we studied several criteria. Consider the following strawman approach that motivates our final problem statement.

Strawman: Find the dimension and a set of its elements whose revenue drop is at least a threshold fraction, T_{EP}, of the total revenue drop, and is most succinct.

We quantify the *explanatory power* (EP) of a set of elements as the fraction of the measure change that it explains. We quantify *succinctness* (P) of a set of elements as the total number of elements in the expression. Therefore, the strawman will find the expression that has explanatory power of at least T_{EP}, and uses the smallest number of elements.

Occam's razor suggests that the most succinct set, as long as it explains the drop within a certain margin of error (T_{EP}), is the best explanation. By this argument, if T_{EP} is set to 0.9, the best dimension and set of elements among the three equations is in Equation 1, since the data center X alone can explain 94% of the total drop.

This approach, however, has deficiencies for root-causing in the presence of multiple dimensions. Though data center X's revenue drop is a high 94% of the total revenue drop, notice that both the forecasted and actual revenue are equally spread between the two data centers X and Y. Data center X provided 94% of the forecasted revenue ($94 out of $100), and actual revenue ($47 out of $50). Data center Y contributed 6% across both values. By comparison, in the device type dimension, device type *PC* contributed 50% of forecasted revenue ($50 out of $100), but 98% of actual revenue ($49 out of $50). The contributions of *Mobile* and *Tablet* device types also varies widely from 25% of forecasted revenue to 0% of actual revenue. The contributions vary along the advertiser dimensions as well, but not as much as they do along the device type dimension.

This large change in the contributions between forecasted and actual revenue from the different elements of the device type dimension is, in general, *surprising and unexpected*. Consequently, we propose that surprise is a better indication of a problem than if we only used succinctness and explanatory power of an expression. Say the root-cause of this revenue drop was due to a configuration file error which caused no ads to be shown on mobiles and tablets. While data center X would still show a huge drop in revenue because it provides 94% of all ads shown across devices, the actual root-cause is better explained by the device type dimension, and the elements *Mobile* and *Tablet*. In other words, the expression in Equation 3 is the best one, even though it is not the most succinct.

To capture this observation, our approach includes a notion of "surprise" (S) associated with an expression (Section 3 has the precise definition). Therefore, generalizing to any measure, our final revenue debugging problem statement can be captured in three steps:

- For a dimension, find all sets of elements that explain at least a threshold fraction, T_{EP}, of the change in the measure (have high explanatory power).
- Among all such sets for each dimension, find the sets that are *most succinct* in that dimension.
- Across all such sets for all dimensions, find the one set that is the *most surprising* in terms of changes in contribution.

Again, for the mock example, with T_{EP} = 0.9, the first step will narrow down the sets to {X} for Data Center, {A1, A3, A4} for Advertiser, and {Mobile, Tablet} and {PC, Mobile, Tablet} for Device Type. Step 2 will narrow down the sets for each dimension to {X}, {A1, A3, A4}, and {Mobile, Tablet}. Step 3 will then use the surprise metric to pick the Device Type dimension and its set {Mobile, Tablet} as the best explanation of the drop.

Our algorithm use a *per-element* threshold of the change in the measure, T_{EEP}, to add to the idea of succinctness. Not only do we want the smallest set of elements, we also want only those elements that contribute *at least* a fraction of T_{EEP} to the anomaly.

We show in Section 3.4 that solving this problem can take exponential time (in number of elements) in the worst case. Therefore, we use a greedy approach that solves this problem approximately.

3 Root-Cause Identification Algorithm

We start with some notation and use it to formally define explanatory power and surprise. We then describe the root-cause identification algorithm. While the algorithm remains the same for fundamental and derived measures, the way explanatory power and surprise are computed for derived measures is more complex and is discussed separately in Section 4.

3.1 Notation

The list of important terms used in this section and their notation are summarized in Table 5. Let the set of measures (e.g., revenue, number of searches)

Term	Notation	Example
Dimensions	$D = \{D_1, D_2, ..., D_n\}$	{Advertiser,Data center,...}
Cardinality of Dimension D_i	C_i	1000's for advertiser, 10's for Data center,...
Elements of Dimension D_i	$E_i = \{E_{i1}, E_{i2}, ..., E_{iC_i}\}$	{Flower123,...} for Advertisers
Measures	$M = \{m_1, m_2, ..., m_k\}$	{Revenue, Number of Searches,...}
Forecasted and Actual Values of measure m for element E_{ij}	$F_{ij}(m)$, $A_{ij}(m)$	Revenue for Flowers123: forecast = $100, actual = $90
Overall forecasted and actual values of measure m	$F(m)$, $A(m)$	Total revenue: forecast = $1,000,000 and actual = $900,000

Table 5: **Notation**

be denoted as $M = \{m_1, m_2, ..., m_k\}$ and let the set of dimensions (e.g., advertisers, data centers) be $D = \{D_1, D_2, ..., D_n\}$. Further, let the set of elements of a given dimension D_i be denoted as $E_i = \{E_{i1}, E_{i2}, ..., E_{iC_i}\}$ where C_i is the cardinality of dimension i. For example, E_{21} may be "Flowers123", an element of the advertiser dimension.

For each of the measures $m \in M$ of interest (including the fundamental and derived measures) and for each of the elements E_{ij}, we have access to the predicted or forecasted values, F_{ij}, as well as the actual observed values, A_{ij}. Note that, as discussed earlier, these values are aggregates corresponding to some time interval of interest (e.g., $100 revenue forecast, $90 revenue actual for element Flower123, dimension advertiser).

For fundamental measures such as revenue or number of searches, both the overall forecasted value for the measure, $F(m)$, as well as the overall actual value, $A(m)$, remain identical across all the dimensions (e.g., $100 forecasted and $50 actual revenue in the example in the previous section). For fundamental measures, the overall measure is simply the summation of value of the measures of the elements of the respective dimensions, but the same is not true for derived measures as they are not additive (Section 4).

Thus, given $F(m)$ and $A(m)$, the algorithm needs to output a potential root cause to explain the difference between the two. For this, it uses explanatory power and surprise, defined next.

3.2 Explanatory power

Explanatory power of an element can be defined as the percentage of change in the overall value of the measure that is explained by change in the given element's value. For fundamental measures, the explanatory power of an element j in dimension i is simply

$$EP_{ij} = (A_{ij}(m) - F_{ij})/(A(m) - F(m)) \quad (4)$$

For example, the total number of searches at a given hour deviates from a forecasted value of 1 million to 0.8 million, and the number of searches at the same hour at a particular data center, DC1, differs from its forecasted value of 0.5 million to 0.4 million, the explanatory power for element DC1 is (0.4-0.5)/(0.8-1) = 50%.

Note that, explanatory power for an element can be more than 100% or even negative, if the change in element is in opposite direction to overall change. However, the sum of explanatory powers of all elements of any dimension should sum up to 100%. Thus, explanatory power fully explains the change in the overall measure.

3.3 Surprise

As discussed in the example in Section 2, a dimension that has large change in its distribution (e.g., Device Type) is more likely to be a root-cause than the dimension that does not exhibit such a change (e.g., Data Center). We now formally define a measure of surprise to capture this notion.

For each element E_{ij}, let $p_{ij}(m)$ be the forecasted or prior probability value given by

$$p_{ij}(m) = F_{ij}(m)/F(m), \forall E_{ij} \quad (5)$$

Given a new anomalous observation, let $q_{ij}(m)$ be the actual or posterior probability value

$$q_{ij}(m) = A_{ij}(m)/A(m), \forall E_{ij} \quad (6)$$

Intuitively, the new observations for a given dimension are surprising if the posterior probability distribution is significantly different from the prior probability distribution. This difference between two probability distributions P and Q can be captured by the relative entropy or Kullback-Leibler (KL) divergence [12]. However, the use of KL divergence in our context has two issues. First, KL divergence is not symmetric. Second, KL divergence is only defined if, for all i, $q_i = 0$ only if $p_i = 0$, which does not hold in our setting (e.g., advertiser pauses his campaign).

Thus, instead of KL Divergence, we use a related measure called the Jensen-Shannon (JS) divergence [12] for computing surprise, defined as

$$D_{JS}(P,Q) = 0.5(\Sigma_i p_i \, log\frac{2p_i}{p_i+q_i} + \Sigma_i q_i \, log\frac{2q_i}{p_i+q_i})$$

Observe that $D_{JS}(P,Q)$ is symmetric and is finite even when $q_i = 0$ and/or $p_i = 0$. Further, $0 \leq D_{JS}(P,Q) \leq 1$, where 0 denotes no change in distribution between P and Q, with higher values denoting greater differences.

Thus, to compute surprise S_{ij} for element E_{ij}, we use $p = p_{ij}(m)$ and $q = q_{ij}(m)$ to compute

$$S_{ij}(m) = 0.5\,(p\,log(\frac{2p}{p+q}) + q\,log(\frac{2q}{p+q})) \quad (7)$$

3.4 Algorithm

```
1  Foreach m ∈ M // Compute surprise for all measures
2      Foreach E_ij // all elements, all dimensions
3              p = F_ij(m)/F(m) // Equation 5
4              q = A_ij(m)/A(m) // Equation 6
5              S_ij(m) = D_JS(p, q) // Equation 7
6  ExplanatorySet = {}
7  Foreach i ∈ D
8      SortedE = E_i.SortDescend(S_ij(m)) //Surprise
9      Candidate = {}, Explains = 0, Surprise = 0
10     Foreach E_ij ∈ SortedE
11         EP = (A_ij(m) − F_ij(m))/(A(m) − F(m))
12         if (EP > T_EEP) // Occam's razor
13             Candidate.Add += E_ij
14             Surprise += S_ij(m)
15             Explains += EP
16         if (Explains > T_EP) // explanatory power
17             Candidate.Surprise = Surprise
18             ExplanatorySet += Candidate
19             break
20 //Sort Explanatoryset by Candidate.Surprise
21 Final = ExplanatorySet.SortDescend(Surprise)
22 Return Final.Take(3) // Top 3 most surprising
```

Figure 2: **Root-Cause Identification Algorithm**

The root-cause identification algorithm seeks to solve the optimization problem specified in Section 2 using the above definitions of explanatory power and surprise.

Note that, obtaining the optimal solution to the problem in the worst case will take exponential time. This can be shown through a simple example: consider a set of size n where each element has an identical explanatory power and we require $n/2$ elements of the set to explain T_{EP}. In this case, every possible subset of cardinality $n/2$ is of minimum size possible (succinct) and has explanatory power of T_{EP}. Thus, we have to compare the surprise values of all these subsets (whose count is exponential in n) in order to find the subset that has the maximum surprise, the optimal solution.

Instead of enumerating various minimum cardinality subsets that have explanatory power of at least T_{EP} , our algorithm (Figure 2) uses the following greedy heuristic. In each dimension, after computing the surprise for all elements (lines 1–5), it first sorts the elements in descending order of surprise (line 8). It then adds each element to a candidate set as long as the element explains at least T_{EEP} of the total anomalous change by itself (lines 12–15). The parameter T_{EEP} helps control the cardinality of the set *(Occam's razor)*. For example, if T_{EEP} is 10% and T_{EP} is 67%, we can have at most 7 elements that explain anomalous change. Further, by examining elements in descending order of surprise, we greedily seek to maximize the surprise of the candidate set. The algorithm adds at most one candidate set per dimension (lines 16–19), as long as the set is able to explain a majority (T_{EP}) of the anomalous change *(explanatory power)*. Finally, the algorithm sorts the various candidate sets by their surprise value and returns the *top three most surprising* candidate sets as potential root-cause candidates (lines 21–22).

4 Derived Measures

Derived measures are functions of fundamental measures that are tracked by troubleshooters since they reveal more information than if one simply tracked the fundamental measures. In this section, we discuss how we compute explanatory power and surprise for derived measures.

4.1 Explanatory Power

While attributing contribution of an individual element to the overall value of a derived measure is important for root-cause identification, this is not as straightforward as computing the same for fundamental measures. In this section, we first start with a illustrative example that helps define explanatory power for derived measures and then present our solution to the derived measure attribution problem.

Example. Consider the hypothetical example in Tables 6 and 7 that shows revenue and number of clicks, respectively, for four different advertisers during an anomalous period. For these two fundamental measures, attribution of the overall change to each of the advertisers is simple using the explanatory power (equation 4) and is shown in the column labelled EP. Thus, for the revenue drop, one can attribute it to advertiser A1 (400%) while for the increase in clicks, one can attribute it to advertiser A2 (200%).

Let us assume that an anomaly is thrown on a measure if it differs from its expected value by at least 20%. Note that the overall revenue has gone down by 10% while the number of clicks is up 16%, neither of which exceeds the anomalous threshold. The corresponding cost-per-click values are shown in Table 8 and using the same 20% threshold, the overall cost-per-click (22.5% decrease) can be labelled anomalous Thus, one can see that derived measures can be useful in surfacing anomalies that are not surfaced by just examining fundamental measures. We confirm this quantitatively in Section 6.

The derived measure attribution problem is the following: how does one attribute the *drop* in overall cost-per-click from 0.2 (expected) to 0.155 (actual) to each of the advertisers? If one examines the individual cost-per-clicks of the advertisers in Table 8, we see that cost-per-click for advertisers A1, A2, A4, are *unchanged* while the cost-per-click for advertiser A3 has *increased*. Thus,

Adver-tiser	Forecasted Revenue	Actual Revenue	EP %
Overall	100	90	-10
A1	50	10	400
A2	0	0	0
A3	40	70	-300
A4	10	10	0

Table 6: **Revenue**

Adver-tiser	Forecasted Clicks	Actual Clicks	EP %
Overall	500	580	16
A1	100	20	-100
A2	200	360	200
A3	100	100	0
A4	100	100	0

Table 7: **Clicks**

Adver-tiser	Forecasted Cost/Click	Actual Cost/Click	EP %
Overall	0.2	0.155	-22.5
A1	0.5	0.5	125
A2	0	0	106
A3	0.4	0.7	-131
A4	0.1	0.1	0

Table 8: **Cost-per-click**

at first glance, it appears that none of the advertisers can be blamed for the overall drop but surely one or more of them must be responsible! Given this situation, how do we go about assigning explanatory power values for the change in cost-per-click to these advertisers?

Examining the fundamental measures does help shed more light. For example, even though cost-per-click of A1 is unchanged, A1 had a 5X drop compared to its forecasted values for both revenue and clicks. Given A1's cost-per-click (0.5) was higher than the overall value (0.2), the 5X reduction implies that A1 was indeed pulling down the overall cost-per-click. The fact that A1 explains some of the decrease in the overall derived measure can be further validated by observing that if we used A1's actual values but assume that the rest of the advertisers delivered their respective forecasted values, then the overall cost-per-click goes down to 60/420 = 0.143 for an impact of -29%.

Similarly, while A2 had 0 revenue as forecasted, A2 had a large increase in clicks, which ends up decreasing the overall cost-per-click. Again, if we used A2's actual values but keep the rest of the advertisers' measures to their forecasted value, the overall cost-per-click goes down to 100/660 = 0.152 for an impact of -24%. The above exercise of changing one advertiser's value at a time also suggests that A1 was more responsible for pulling down overall cost-per-click than A2 (since use of A1's actual values resulted in lower overall value than for A2).

Now consider A3. A3 had a higher revenue than forecasted without change in clicks, so A3 was clearly not contributing to the overall drop. Using A3's actual values in the above exercise would in fact increase the overall cost-per-click to 0.26, for an impact of +30%.

Finally, A4 had no change in either revenue or clicks. Therefore, A4 had no impact in overall cost-per-click.

Normalizing the individual impact values so that all the elements in total explain 100% of the overall change, the above exercise would give A1's explanatory power as 125%, A2's as 106%, A3's as -131% and A4's as 0%.

Summarizing the observations in the above example, one can see that an element's explanatory power for derived measures can be determined by computing a new derived measure value, where the actual value of the given element and forecasted values of all other elements are used, and comparing this derived measure value to the expected value of the derived measure.

Now, the question is how do we formalize this intuition in order to determine the explanatory power for arbitrary derived measures? We describe this next.

Derived measure attribution. Our solution to the derived measure attribution problem is adapted from *partial derivatives and finite-difference calculus*. Recall that a partial derivative is a measure of how a function of several variables changes when one of its variable changes. However, since we operate in the discrete domain, we use partial derivative equivalents from finite-difference calculus [13].

We formally define explanatory power of an element i for a derived measure, which is function $h(m_1,...,m_k)$ of fundamental measures $m_1, ..., m_k$, as the partial derivative with respect to i in finite-differences of $h(.)$, normalized so that the value across all elements of the dimension sum up to 100%.

While the above definition is general and applicable to derived measures that are arbitrary functions of fundamental measures (as long as they are differentiable in finite-differences), we now illustrate it through the specific example of derived functions of the form $A(m_1)/A(m_2)$, which make up many of the derived measures in ad systems (Figure 1). For example, for the cost-per-click derived measure, we have m_1 = revenue and m_2 = clicks.

The partial derivative in finite-differences of f(.)/g(.) is of the form $(\Delta f * g - \Delta g * f)/(g * (g + \Delta g))$, and is similar to continuous domain partial derivative, except for the extra Δg in the denominator.

Thus, explanatory power of element j for dimension i for derived measures of the form m_1/m_2 is given by

$$\begin{aligned} EP_{ij} \quad = \quad & ((A_{ij}(m_1) - F_{ij}(m_1)) * F(m_2) \\ & -(A_{ij}(m_2) - F_{ij}(m_2)) * F(m_1)) \\ & /(F(m_2) * \\ & (F(m_2) + A_{ij}(m_2) - F_{ij}(m_2))) \end{aligned} \quad (8)$$

We compute EP_{ij} for each of the elements using the above equation and normalize it so that they add up to 100%.

Table 8 shows the explanatory power computed using the above formula for each of the advertisers. We can see that the rank ordering of A1, A2, A4, and A3 and their respective explanatory power values for the attribution to the overall change agrees with the intuitive observations made earlier.

4.2 Surprise

Recall that we defined surprise for fundamental measures in Section 3.3 based on the relative entropy (specifically, JS divergence) between the prior and posterior mass functions of values for measure m. In this section, we seek to extend the notion of surprise to derived functions of multiple measures.

Consider the cost-per-click example in the previous section. A simple approach for computing surprise for derived measure is as follows. Just as for fundamental measures, one could compute prior and posterior probability values for cost-per-click for each element E_{ij}, say p_{ij}(cost-per-click) and q_{ij}(cost-per-click) and compute the surprise just as in Section 3.3.

However, such an approach will not work. Consider the example of advertiser A2 in Table 8. A2's cost-per-click was forecasted to be zero and the actual value was also 0. Thus, if one used the above approach to compute surprise for element A2, it would have a value of 0 (no surprise). However, we found that A2 had a high explanatory power of 106% for the overall change in cost-per-click due to changes in A2's number of clicks.

Examining the problem from the perspective of relative-entropy, given several measures, we first need to compute the joint probability distribution of the measures and then compute relative entropy of the joint probability distribution function. If the measures are independent, then the relative entropy (JS divergence as well) of the joint probability distribution is simply the sum of the relative entropy of the individual measure's probability distributions. In ad systems, the measures are not always strictly independent since some of them can be correlated (e.g., as the number of searches increase, revenue can be expected to increase). However, as an approximation, we assume that measures are independent, and compute the *surprise for derived measures as the summation of the surprise of the individual measures that are part of the derived function*.

5 Implementation and Experience

In this section, we describe our implementation of the above algorithms in the Adtributor tool and outline our experience with a pilot deployment in a production ad system.

5.1 Implementation

In our implementation, a database records, in real-time, counters for all measures, dimensions, and elements and exposes them as an OLAP service that supports multi-dimensional analytical queries [19]. When the system triggers an anomalous event, the Adtributor toolchain first gathers data relevant to the anomaly such as time of anomaly, measure, data for various measures, dimensions and elements. After the data has been queried from the database, Adtributor employs the root-cause algorithm to discover potential root-causes for the given anomaly.

Figure 3: **An example output of the Adtributor. Note: certain sensitive fields are masked.**

Recall that measures are not necessarily independent of each other. An anomaly on a certain measure could be correlated with changes in value of another. Therefore, we build a *dependency graph* of measures, and for a given anomalous measure, run the root-causing algorithm for every measure that correlates with it.

Adtributor filters the candidate set of root-causes (as described in Section 3) to produce the final list of root-causes. We use a T_{EP} value of 67% and a T_{EEP} value of 10%. These threshold values are driven by what the troubleshooters already use in the manual process. Also, our current implementation singles out a list of the top three dimensions. The troubleshooting experts recommended this number based on their own requirements and also on ease of visualization. With a smaller number they could miss useful information, while a larger number would lead to too much information for them to sift through.

The final output is a self-contained HTML5 application. Figure 3 shows an example of the output produced by the Adtributor toolchain. The visualization of the root-causes contains the following information:

- **Dependency Graph**: A graphical representation of dependencies between the different measures in the system (left half).
- **Measure Historical Graph**: A graph depicting the historical behavior of a measure (top right graph).
- **Element Root-Causes and Historical Graph**: For a given measure and dimension, the top elements that are root-causes. The element root-causes are grouped by dimensions with their historical graphs (under top right graph).

We arrived at the visualization requirements through iterative discussions with the troubleshooting experts. The dependency graph allows them to observe causality between the values of different measures, and the histori-

cal graphs per-dimension help them in making a more informed choice on what exactly was the root-cause. The entire Adtributor toolchain is implemented using .NET Framework 4 using 12,500 lines of code and executed automatically for each anomaly.

5.2 Deployment Experience

We conducted a pilot deployment of Adtributor between May 1, 2013 and May 10, 2013 with the troubleshooters who work with the production system on root-causing anomalies to understand the usefulness of Adtributor. This deployment was partially successful in helping the troubleshooters with their current processes. The findings of this pilot resulted in a set of improvements to our algorithm and visualization which led to significantly better performance as we show in our evaluation in Section 6.

Volatile dimensions: Various dimensions can be extremely volatile, and unexpected changes can occur in measures along these axes even though they are not necessarily the root-cause of the problem. Consider the example of an advertiser who frequently changes the budget allotment to their ads. When there is a revenue anomaly, this can sometimes cause the root-causing algorithm to pick the advertiser as a culprit even though the change coincidentally occurred just a little before the anomaly event. This drove us to improve our prediction algorithm for measures associated with elements of volatile dimensions by increasing the weightage given to large changes in the near-past in our prediction model, thereby fixing this problem to a large extent.

Visualization enhancements: The dependency graph of related measures was found to be very useful by the troubleshooters. However, the current view in the tool is limited to a small set of measures. There are hundreds of other measures being monitored within the ad system for which the dependencies are not known. We have therefore used a Bayesian structure learning algorithm [5] to infer a subset of these dependencies and plan to enhance the visualization of the dependency graph with these additional measures.

6 Evaluation

In the Section, we first describe four case studies in which multi-dimensional analysis is key to arriving at the final root-cause. Next, we provide a quantified evaluation of the accuracy of Adtributor, and the time savings we achieve with the tool.

6.1 Case Studies

Case 1: This was triggered by an anomalous drop in revenue. On performing the multi-dimensional analysis, we found that the dimension *Browser* was responsible. Figure 4 helps explain how Adtributor arrived at this result. It shows the percentage contribution to revenue along three dimensions – *Browser*, *Data Center*, and *Bucket* – for predicted revenue and actual revenue (see Table 1, example 3 for the definition of a bucket.). Notice that Browser 3's revenue contribution was predicted to be 12%, but its actual revenue was 0%! Similarly, Browser 1's contribution was predicted to be 60%, but was actually much higher at 74%. Neither the Data Center dimension nor the Bucket dimension show such surprising changes in contribution. This problem was actionable, since a further investigation revealed that a configuration error had caused no ads to be shown to users on Browser 3. Correcting the error fixed the problem and further loss in revenue.

Case 2: We noted an anomalous revenue increase at a particular time, which Adtributor attributed to a certain set of six advertisers. Two of these advertisers were airline ticket vendors, two were car rental agencies, and the remaining two were hotels. In aggregate, they fully explained the change in revenue. Delving into the issue, we noticed that these advertisers had deliberately increased their budgets for a certain period of time. The ads were appearing in a geographic region which had a long-weekend holiday approaching. Thus, we inferred that the advertisers were trying to capitalize and capture the attention of users as they performed vacation-related searches. Clearly, in this case, the sudden rise in revenue was attributed to advertiser behavior and not due to an actionable bug in the system. Several such anomalies also occur when advertisers deliberately drop their budgets as well.

Case 3: The total number of searches went anomalously high, and an analysis showed that most of the increase was attributed along the *User-agent string* dimension. From post-processing on this result, it was inferred that a majority of the searches with the repeated user-agent string were coming from a small range of IP addresses, and therefore, suspiciously characteristic of bot-traffic. In particular, the goal of this bot was to perform queries and collect information for search-engine optimization (SEO). This was an actionable issue which was fixed promptly by filtering the contribution of this traffic to the various metrics.

Case 4: We notice that sometimes, publishers change the placement of advertisements on their page, which make ads more (or less) conspicuous. This in turn causes a corresponding increase or decrease in revenue. These show up as revenue changes along the dimension of *Ad Position* on the page. For instance, if the publisher moves an ad meant to be shown on the side of the page to the top of the page, this presents itself as a surprising increase in revenue attributed to ads shown on the top. This is because users tend to click more on ads shown on the top of a page than they do on ads shown on the side.

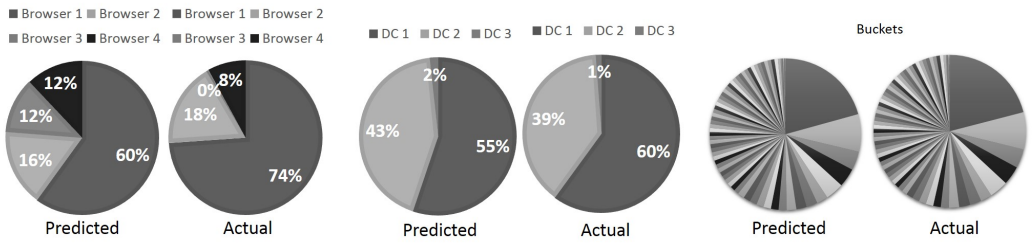


Figure 4: **Predicted and actual revenues for the Browser, Data center, and Bucket dimensions (Case Study 1).**

6.2 Comparative Study

Our quantitative evaluation is based on using Adtributor to root-cause problems in a widely deployed ad system. We evaluate all anomalies generated on a total of 12 measures, both fundamental and derived, across 33 dimensions. The results we present here use a subset of 128 valid anomalies generated by over a billion searches between September 1, 2013 and September 15, 2013 across 8 populations: PC and Mobile ad systems for USA, UK, France and Germany. For the purpose of this study, we do not consider false-positives in the anomaly generation process as they are weeded out by troubleshooters before applying the root-causing process[2]. 50% of the tested anomalies were generated solely on derived measures, with no related anomalies being generated on the respective fundamental measures that constitute the derived measure. This shows that using derived measures in aggregate root-cause analysis is extremely important.

We compare the output of Adtributor's multi-dimensional analysis with the output of the troubleshooting team that performs an in-depth and detailed analysis of these anomalies through manual means with the assistance of other tools (not Adtributor). Manually analyzing the cause of the anomalies has a number of advantages. The troubleshooters are aware of a large amount of information and domain-knowledge, and they frequently use this knowledge in the troubleshooting process. An automated tool such as Adtributor cannot possibly have an understanding of all of this. Further, Adtributor only narrows the scope of the root-cause (Column 4 of Table 1) – a manual process may still be necessary in many cases to identify the final root-cause (Column 4 of Table 1) since some of the data necessary to do this next step may not be available for the automated process (e.g., verifying whether the publisher indeed changed the position of ads).

However, the advantage of using Adtributor is that it aids the manual troubleshooting process by 1) using the multi-dimensional root-cause analysis to *exhaustively* check all possible dimensions (as we show, in a few cases, the manual process may overlook a dimension, leading to erroneous conclusions) and 2) Significantly faster processing to bubble up the top suspect candidates. For example, there are dozens of dimensions and some dimensions can have thousands of elements.

As described in Section 5, Adtributor displays the top three dimensions and their elements as potential suspects. We say that Adtributor matches the output of the manual root-causing process if it shows the same dimension and exactly the same elements as the manual process at any one of these three positions.

No. of anomalies	128
No. of matches	118 (1:81, 2:27,3:10)
Manual errors found	4
Adtributor's errors	5
Ambiguous	1
Adtributor accuracy	(118+4)/128=95.3%
Strawman accuracy (no surprise)	20.0%

Table 9: **Results summary from our comparison of Adtributor with manual scrutiny (and Strawman).**

Table 9 shows the results of the comparison between the output of Adtributor and the manual investigation. Of the 128 anomalies, Adtributor matched the results of the manual analysis in 118 cases. Of these, 81 (69%) matched in position 1, 27 (23%) matched only in position 2 and not in position 1, and 10 (8%) matched in position 3, and not in position 1 or 2. Of the 10 anomalies for which we did not match the manual output, we performed a deeper dive with the troubleshooting expert. On careful scrutiny, we found that out of the 10, 4 of the manual root-causes were erroneous, and Adtributor's output in position 1 was, in fact, the correct root-cause. This shows the utility of using a systematic algorithm, as in Adtributor, that exhaustively searching all dimensions to perform multi-dimensional root-cause analysis.

Out of the remaining 6 anomalies, the manual output was correct in 5 of them while the output of Adtributor was erroneous. In all of these cases, Adtributor suffered from a lack of domain knowledge, or the lack of knowledge of events external to the system which the troubleshooters were explicitly aware of. In one case (labelled ambiguous), however, the troubleshooter felt the dimension and elements blamed by Adtributor was

[2]Evaluating the number of false-positives and negatives would be to evaluate the anomaly detection algorithm which, as mentioned earlier, is out of scope of this paper.

as likely to be the true root-cause as the one obtained through manual analysis. In this case, he felt a further drill-down would be required to determine the correct root-cause. Taking the manual errors into account, Adtributor's *overall accuracy was (118+4)/128, or 95%*.

We also compare the potential time that could be saved using Adtributor compared to the first step in the manual troubleshooting process that identifies the dimension and elements that may be potential root-causes. Adtributor uses a multi-threaded implementation and caching to speed up the process of studying every dimension and every measure. It has a turnaround of approximately 3-5 minutes for each anomaly. The manual process of troubleshooting took between 13 minutes for the fastest anomaly to up to 231 minutes, with an average turnaround time of 73 minutes. Therefore, we conclude that Adtributor *speeds up the initial root-causing process by an order of magnitude*.

Finally, we show the value of using surprise by comparing our algorithm to the Strawman discussed in Section 2 that only uses succinctness and explanatory power. Compared to Adtributor's accuracy of 95%, we found that Strawman had an accuracy of only 20%. This clearly demonstrates the value of using surprise to identify the right dimension and elements as the root-cause.

7 Applicability beyond Ad Systems

We believe that the techniques introduced in this paper are general enough to be useful in other settings. For example,

Multi-dimensional analysis: Consider a web-server with a global audience that suddenly sees the number of hits drop sharply. Many of the dimensions considered in this paper such as data centers or CDNs, browsers, user locations, fraud operators/bots, etc. may all be potential root-causes that a multi-dimensional analysis can help disambiguate.

Derived measure attribution: Consider the following problem. The Mean-opinion-score (MOS) for VoIP calls has dropped and the investigators would like to understand which of the links in the route of the call is most responsible for this drop. Each link may have different amounts of delay, jitter, and loss percentages, and the MOS is a complex function of measures such as delay, loss, and jitter [6]. The use of the derived attribution technique can help compute the explanatory power of the drop in MOS for each of links.

8 Related Work

System and Network Root-Cause Analysis: Previous research has extensively studied root-causing performance and failure problems in systems and networks [14, 2, 10, 1, 21, 15, 3, 20, 11, 18]. Some of these use traces across individual requests through systems [3, 14] to diagnose problems, while others use aggregate counters of system performance or configuration values [15, 10, 20] to diagnose problems.

Distalyzer [14] is an example of the former category. It uses individual event logs and learns anomalous patterns between events that indicate a performance problem in a system component. Ganesha [15] is an example of the latter. It uses clustering approaches across aggregate measures, such as CPU usage, to build distinct profiles of MapReduce nodes. While our approach too uses aggregate measures, we intend to find more than performance problems or diagnose failures.

Q-Score [18] uses machine-learning to arrive at root-causes. We tried similar approaches and decided against them because selecting the right set of features to input to a stock machine-learning algorithm turned out to be a non-trivial task. Instead, we found that building a customized algorithm was simpler and better suited to analysis and feedback by our domain experts.

SCORE [11] localizes IP faults to underlying components using succinctness of explanation. Given a set of link failures as observation, it determines the smallest set of risk groups that explain failures. However, as we show, this approach is not enough to perform attribution across dimensions and a notion of surprise is essential to complete our solution.

Data Mining for Summarization: Previous work in data mining [17, 16, 7] has concentrated on summarizing multi-dimensional data in OLAP products. The objective is to provide an easily interpretable summary of the differences in data values across multiple dimensions. Such summarization techniques have been applied to network traffic summarization as well [9]. While data summarization across multiple dimensions is related to our work, it does not match our objective of finding surprising changes to perform root-cause analysis. In fact, our approach to root-cause analysis is complementary to these approaches and can be applied on the summaries that they generate.

9 Conclusion

We have described an algorithm, implementation, and evaluation of an approach that uses multi-dimensional analysis for root-causing problems in large-scale ad systems. We found that our approach has high accuracy (95%), helped identify more accurate root causes than the manual investigation in a few cases, and was able to reduce troubleshooting time significantly.

10 Acknowledgments

We would like to thank our shepherd VYAS SEKAR for his valuable comments and suggestions. We would also like to thank MURALI KRISHNA for helping us validate the output of Adtributor and determine its accuracy.

References

[1] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. Padmanabhan, and G. Voelker. NetPrints: Diagnosing Home Network Misconfigurations using Shared Knowledge. In *NSDI*, 2009.

[2] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies. In *SIGCOMM*, 2007.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of USENIX OSDI*, 2004.

[4] G. Box, G. M. Jenkins, and C. Gregory. *Time Series Analysis: Forecasting and Control*. Prentice-Hall, 1994.

[5] D. M. Chickering. The winmine toolkit. Technical Report MSR-TR-2002-103, Microsoft, Redmond, WA, 2002.

[6] R. Cole and J. Rosenbluth. Voice over IP performance monitoring. *CCR*, Apr 2001.

[7] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data. In *Proceedings of ACM SIGMOD*, 2004.

[8] V. Dave, S. Guha, and Y. Zhang. Measuring and fingerprinting click-spam in ad networks. In *Proceedings of ACM SIGCOMM*, 2012.

[9] C. Estan, S. Savage, and G. Varghese. Mining anomalies using traffic feature distributions. In *Proceedings of ACM SIGCOMM*, 2003.

[10] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, and J. Padhye. Detailed diagnosis in computer networks. In *Sigcomm*. ACM, 2010.

[11] R. R. Kompella, J. Yates, A. Greenberg, and A. Snoeren. IP fault localization via risk modelling. In *Proceedings of USENIX NSDI*, 2005.

[12] J. Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991.

[13] L. Milne-Thomson. *The calculus of Finite Differences*. Macmillan, 1933.

[14] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of USENIX NSDI*, 2012.

[15] X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: blackBox diagnosis of MapReduce systems. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):8–13, 2009.

[16] S. Sarawagi. Explaining differences in multidimensional aggregates. In *Proceedings of VLDB*, 1999.

[17] S. Sarawagi. iDiff: Informative Summarization of Differences in Multidimensional Aggregates. *Data Mining and Knowledge Discovery*, 5(4):255–276, 2001.

[18] H. H. Song, Z. Ge, A. Mahimkar, J. Wang, J. Yates, Y. Zhang, A. Basso, and M. Chen. Q-score: Proactive service quality assessment in a large IPTV system. In *Proceedings of ACM IMC*, 2011.

[19] E. Thomsen, G. Spofford, and D. Chase. *Microsoft OLAP solutions*. John Wiley & Sons, Inc., 1999.

[20] H. Wang, J. Platt, Y. Chen, R. Zhang, and Y. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI*, 2004.

[21] H. Yan, L. Breslau, D. Massey, D. Pei, and J. Yates. G-RCA: A Generic Root Cause Analysis Platform for Service Quality Management in Large IP Networks. In *Proceedings of ACM CoNext*, 2010.

DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps

Bin Liu[*], *Suman Nath*[‡], *Ramesh Govindan*[*], *Jie Liu*[‡]
[*]*University of Southern California,* [‡]*Microsoft Research*

Abstract

Ad networks for mobile apps require inspection of the visual layout of their ads to detect certain types of *placement frauds*. Doing this manually is error prone, and does not scale to the sizes of today's app stores. In this paper, we design a system called DECAF to automatically discover various placement frauds scalably and effectively. DECAF uses automated app navigation, together with optimizations to scan through a large number of visual elements within a limited time. It also includes a framework for efficiently detecting whether ads within an app violate an extensible set of rules that govern ad placement and display. We have implemented DECAF for Windows-based mobile platforms, and applied it to 1,150 tablet apps and 50,000 phone apps in order to characterize the prevalence of ad frauds. DECAF has been used by the ad fraud team in Microsoft and has helped find many instances of ad frauds.

1 Introduction

Several recent studies have pointed out that advertising in mobile (smartphones and tablets) apps is plagued by various types of frauds. Mobile app advertisers are estimated to lose nearly 1 billion dollars (12% of the mobile ad budget) in 2013 due to these frauds [4]. The frauds fall under two main categories: (1) *Bot-driven frauds* employ bot networks or paid users to initiate fake ad impressions and clicks [4] (more than 18% impressions/clicks come from bots [13]), and (2) *Placement frauds* manipulate visual layouts of ads to trigger ad impressions and unintentional clicks from real users (47% of user clicks are reportedly accidental [12]). Mobile app publishers are incentivized to commit such frauds since ad networks pay them based on impression count [10, 7, 8], click count [7, 8], or more commonly, combinations of both [7, 8]. Bot-driven ad frauds have been studied recently [4, 20, 40], but placement frauds in mobile apps have not received much attention.

Contributions. In this paper, we make two contributions. First, we present the design and implementation of a scalable system for automatically detecting ad placement fraud in mobile apps. Second, using a large collection of apps, we characterize the prevalence of ad placement fraud and how these frauds correlate with app ratings, app categories, and other factors.

Figure 1: *Placement Fraud Examples*

Detecting ad fraud. In Web advertising, most fraud detection is centered around analyzing server-side logs [51] or network traffic [38, 39], which are mostly effective for detecting bot-driven ads. These can also reveal placement frauds to some degree (e.g., an ad not shown to users will never receive any clicks), but such detection is possible only *after* fraudulent impressions and clicks have been created. While this may be feasible for mobile apps, we explore a qualitatively different approach: to detect fraudulent behavior by *analyzing the structure* of the app, an approach that can detect placement frauds more effectively and *before* an app is used (e.g., before it is released to the app store). Our approach leverages the highly specific, and legally enforceable, *terms and conditions* that ad networks place on app developers (Section 2). For example, Microsoft Advertising says developers must not "edit, resize, modify, filter, obscure, hide, make transparent, or reorder any advertising" [11]. Despite these prohibitions, app developers continue to engage in fraud: Figure 1 shows (on the left) an app in which 3 ads are shown at the bottom of a page while ad networks restrict developers to 1 per page, and (on the right) an app in which an ad is hidden behind UI buttons.

The key insight in our work is that manipulation of the visual layout of ads in a mobile app can be programmatically detected by combining two key ideas: (a) a UI automation tool that permits automated traversal of all the "pages" of a mobile app, and (b) extensible fraud checkers that test the visual layout of each "page" for compliance with an ad network's terms and conditions.

While we use the term ad fraud, we emphasize that our work deems as fraud any violation of published terms and conditions, and *does not attempt to infer whether the violations are intentional or not*.

We have designed a system called DECAF that leverages the insight discussed above (Section 3). First, it employs an automation tool called a *Monkey* that, given a mobile app binary, can automatically execute it and navigate to various parts (i.e., states) of the apps by simulating user interaction (e.g., clicking a button, swiping a page, etc.). The idea of using a Monkey is not new [44, 35, 33, 47]. The key optimization goals of a Monkey are good coverage and speed—the Monkey should be able to traverse a good fraction of app states within a short time. However, even for relatively simpler apps, naïve state traversal based on a UI automation framework can take several hours per app, as reported in a recent work [33]. Combined with the massive sizes of popular app stores, this clearly motivates for scalable traversals. Recent works therefore propose optimization techniques, many of which require instrumenting apps [44, 47] or the OS [35].

DECAF *treats apps and the underlying OS as black boxes* and relies on a UI automation framework. The advantage of this approach is that DECAF can scan apps written in multiple languages (e.g., Windows Store apps can be written in C#, HTML/JavaScript, and C++) and potentially from different platforms (e.g., Windows and Android). However, this flexibility comes at the cost of limited information from the UI automation framework; existing automation frameworks do not provide information on callback functions, system events, z-coordinate of UI elements, etc. that are required by optimizations proposed in [44], [35], and [47]. To cope with this, DECAF employs several novel techniques: a *fuzzy matching*-based technique to robustly identify structurally similar pages with similar ad placement (so that it suffices for the Monkey to visit only one of them), a machine learning-based predictor to avoid visiting equivalent pages, an app usage based technique to prioritize app navigation, and a resource usage based technique for fast and reliable detection of page load completion.

The second component of DECAF is to efficiently identify fraudulent behavior in a given app state. We find that, rather surprisingly, analyzing visual layout of an app page to detect possible ad fraud is nontrivial. This is due to complex UI layouts of app pages (especially in tablet apps), incomplete UI layout information from the UI automation framework (e.g., missing z-coordinate), mismatch between device's screen size and app's page size (e.g., panoramic pages), and variable behavior of ad networks (e.g., occasionally not serving any ad due to unavailability of specific types of ads), etc. We develop novel techniques to reliably address these challenges.

We have implemented DECAF to run on Windows 8 (tablet) apps and Windows Phone 8 apps (Section 5). Experiments show that DECAF achieves a coverage of 94% (compared to humans) in 20 minutes of execution per app and is capable of detecting many types of ad frauds in existing apps (Section 6).

Characterizing Ad Fraud. Using DECAF we have also analyzed 50,000 Windows Phone 8 apps and 1,150 Windows tablet apps, and discovered many occurrences of various types of frauds (Section 7). Many of these frauds were found in apps that have been in app stores for more than two years, yet the frauds remained undetected. We have also correlated the fraud data with various app metadata crawled from the app store and observed interesting patterns. For example, we found that fraud incidence appears independent of ad rating on both phone and tablet, and some app categories exhibit higher incidence of fraud than others but the specific categories are different for phone and tablet apps. Finally, we find that few publishers commit most of the frauds. These results suggest ways in which ad networks can selectively allocate resources for fraud checking.

DECAF has been used by the ad fraud team in Microsoft and has helped detect many fraudulent apps. Fraudulent publishers were contacted to fix the problems, and the apps whose publishers did not cooperate with such notices have been blacklisted and denied ad delivery. To our knowledge, DECAF is the first tool to automatically detect ad fraud in mobile app stores.

2 Background, Goals and Challenges

Background. Many mobile app publishers use in-app advertisements as their source of revenue; more than 50% of the apps in major app stores show ads [28]. To embed ads in an app, an *app publisher* registers with a mobile *ad network* such as AdMob[6], iAd [8], or Microsoft Mobile Advertising [9]. In turn, ad networks contract with *advertisers* to deliver ads to apps. Generally speaking, the ad network provides the publisher with an ad control (i.e., a library with some visual elements embedded within). The publisher includes this ad control in her app, and assigns it some screen real estate. When the app runs and the ad control is loaded, it fetches ads from the ad network and displays it to the user.

Ad networks pay publishers based on the number of times ads are seen (called *impressions*) or clicked by users, or some combination thereof. For example, Microsoft Mobile Advertising pays in proportion to total impression count $\times$ the overall click probability.

To be fair to advertisers, ad networks usually impose strict guidelines (called *prohibitions*) on how ad controls should be used in apps, documented in lengthy *Publisher Terms and Conditions*. We call all violations of these prohibitions *frauds*, regardless of whether they are vio-

lated intentionally or unintentionally.

Goals. In this paper, we focus on automated detection of a special category of mobile app frauds that we call *placement frauds*. Such frauds are different from *interaction frauds* such as illegitimate clicks by bots and *content frauds* such as modifying ad contents [34].

Placement Fraud. These frauds relate to how and where the ad control is placed. Ad networks impose placement restrictions to prevent impression or click inflation, while the advertiser may restrict what kinds of content (i.e., ad context) the ads are placed with. For instance, Microsoft Mobile Advertising stipulates that a publisher must not "edit, resize, modify, filter, obscure, hide, make transparent, or reorder any advertising" and must not "include any Ad Inventory or display any ads ... that includes materials or links to materials that are unlawful (including the sale of counterfeit goods or copyright piracy), obscene,..." [11]. Similarly, Google AdMob's terms dictate that "Ads should not be placed very close to or underneath buttons or any other object which users may accidentally click while interacting with your application" and "Ads should not be placed in areas where users will randomly click or place their fingers on the screen" [1].

We consider two categories of placement frauds.

Structural frauds: These frauds relate to how the ad controls are placed. Violators may manipulate the UI layout to inflate *impressions*, or to reduce ad's foot print on screen. This can be done in multiple ways:

- An app page contains **too many ads** (Microsoft Advertising allows at most 1 ad per phone screen and 3 ads per tablet screen [11]).
- Ads are **hidden** behind other controls (e.g., buttons or images) or placed **outside the screen**. (This violates the terms and conditions in [11, 1]). Developers often use this trick to give users the feel of an "ad-free app", or to accommodate many ads in a page while evading manual inspection.
- Ads are resized and made **too small** for users to read.
- Ads are overlapped with or placed next to actionable controls, such as buttons, to capture accidental clicks.

Contextual frauds: These frauds place ads in inappropriate contexts. For example, a *page context* fraud places ads in pages containing inappropriate (e.g., adult) content. Many advertisers who try to increase brand image via display ads, do not want to show such ads in pages. Ad networks therefore prohibit displaying ads in pages containing "obscene, pornographic, gambling related or religious" contents [11]. Publishers may violate these rules in an attempt to inflate impression counts.

Detecting placement frauds manually in mobile apps can be extremely tedious and error prone. This is because an app can have a large number of pages and some violations (*e.g.,* ads hidden behind UI controls) cannot often be detected visually. This, combined with the scale of popular app stores clearly suggests the need for automation in mobile app ad fraud detection.

Beyond detecting fraud, a second goal of this paper is to characterize the prevalence of ad fraud by type, and correlate ad fraud with app popularity, app type, or other measures. Such a characterization provides an initial glimpse into the incidences of ad fraud in today's apps, and, if tracked over time, can be used to access the effectiveness of automated fraud detection tools.

Challenges. The basic approach to detecting placement fraud automatically is to programmatically inspect the visual elements and content in an app. But, because of the large number of apps and their visual complexity (especially on tablets), programmed visual inspection of apps requires searching a large, potentially unbounded. space. In this setting, inspection of visual elements thus faces two competing challenges: *coverage*, and *speed*. A more complete search of the visual elements can yield high coverage at the expense of requiring significant computation and therefore sacrificing speed. A key research contribution in this paper is to address the tension between these challenges.

Beyond searching the space of all visual elements, the second key challenge is to accurately identify ad fraud within a given visual element. Detecting structural frauds in an app page requires analyzing the structure of the page and ads in it. This analysis is more challenging than it seems. For example, checking if a page shows more than one ad (or k ads in general) in a screen at any given time might seem straightforward, but can be hard on a *panoramic page* that is larger than the screen size and that the user can horizontally pan and/or vertically scroll. Such a page may contain multiple ads without violating the rule, as long as no more than one ad is visible in any scrolled/panned position of the screen (this is known as the "sliding screen" problem). Similarly whether an ad is hidden behind other UI controls is not straightforward if the underlying framework does not provide the depths (or, z-coordinates) of various UI controls. Finally, detecting contextual fraud is fundamentally more difficult as it requires analyzing the content of the page (and hence not feasible in-field when real users use the apps).

3 DECAF Overview

DECAF is designed to be used by app stores or ad networks. It takes a collection of apps and a set of fraud compliance rules as input, and outputs apps/pages that violate these rules. DECAF runs on app binaries and does not assume any developer input.

One might consider using static analysis of an app's UI to detect structural fraud. However, a fraudulent app can dynamically create ad controls or change their properties during run time and bypass such static analysis.

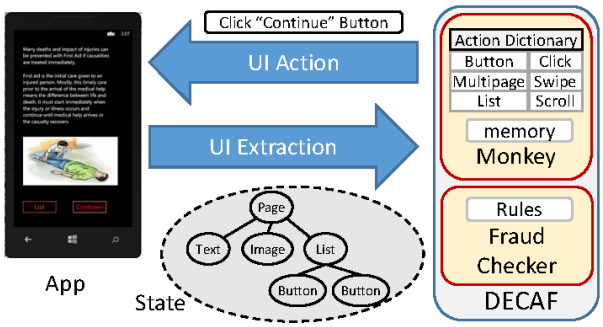


Figure 2: *The architecture of* DECAF *includes a Monkey that controls the execution and an extensible set of fraud detection policies.*

Static analysis also fails to detect contextual fraud. DECAF therefore performs dynamic checking (analogous to [44, 35, 33, 43]) in which it checks the implementation of an app by directly executing it in an emulator.

Unlike [44, 35], DECAF uses a *black-box* approach and does not instrument the app binary or the OS. This choice is pragmatic: Windows 8 tablet apps are implemented in multiple languages (C#, HTML/Javascript, C++)[1], and our design allows us to be language-agnostic. However, as we discuss later, this requires novel techniques to achieve high coverage and speed.

Figure 2 shows the architecture of DECAF. DECAF runs mobile apps in an emulator and interacts with the app through two channels: a *UI Extraction* channel for extracting UI elements and their layout in the current page of an app (shown as a Document Object Model (DOM) tree in Figure 2), and a *UI Action* channel for triggering an action on a given UI element (such as clicking on a button). In Section 5, we describe how these channels are implemented. DECAF itself has two key components: (1) a Monkey that controls the app execution using these channels and (2) a fraud checker that examines page contents and layout for ad fraud.

3.1 The Monkey

The execution of an app by a Monkey can be viewed as traversal on a state-transition graph that makes transitions from one *state* to the next based on UI inputs, such as clicking, swiping, and scrolling. Each state corresponds to a page in the app, and the Monkey is the program that provides UI inputs (through the UI Action channel) for each visited state.

At each state that it visits, the Monkey uses the UI extraction channel to extract *page information*, which includes (1) structural metadata such as size, location, visibility, layer information of each ad and non-ad control in the current page, and (2) content such as the text, images, and urls in the page. The information is extracted from the DOM tree of the page; the DOM tree contains all UI elements on a given page along with contents of the elements. The Monkey also has a dictionary of actions associated with each UI type, such as clicking a button, swiping a multi-page, and scrolling a list, and uses this dictionary to generate UI inputs on the UI action channel.

Starting from an empty state and a freshly loaded app, the Monkey iterates through the UI controls on the page to the next state, until it has no transitions to make (either because all its transitions have been already explored, or it does not contain any actionable UI control). Before making a transition, the Monkey must wait for the current page to be completely loaded; page load times can be variable due to network delays, for example. After visiting a state, it uses one of two strategies. If a (hardware or software) back button is available, it retracts to a previous (in depth-first order) state. If no back button is available (e.g., many tablets do not have a physical back button and some apps do not provide a software back button), the Monkey restarts the app, navigates to the previous state through a shortest path from the first page, and starts the exploration process.

In order to explore a large fraction of useful states within a limited time, the Monkey needs various optimizations. For example, it needs to determine if two states are equivalent so that it can avoid exploring states that have already been visited. It also needs to prioritize states, so that it can explore more important or useful states within the limited time budget. We discuss in Section 4 how DECAF addresses these. The Monkey also needs to address many other systems issues such as dealing with non-deterministic transitions and transient crashes, detecting transition to an external program (such as a browser), etc. DECAF incorporates solutions to these issues, but we omit the details here for brevity.

3.2 Fraud Checker

At each quiescent state, DECAF invokes the fraud checker. The checker has a set of *detectors*, each of which decides if the layout or page context violates a particular rule. While DECAF's detectors are extensible, our current implementation includes the following detectors.

Small Ads: The detector returns true if any ad in the given page is smaller than the minimal valid size required by the ad network. The operation is simple as the automation framework provides widths and heights of ads.

Hidden Ads: The detector returns true if any ad in the given page is (partially) hidden or unviewable. Conceptually, this operation is not hard. For each ad, the detector first finds the non-ad GUI elements, then checks if any of these non-ad elements is rendered above the ad. In practice, however, this is nontrivial due to the fact that existing automation frameworks (e.g., for Windows and for Android) do not provide z-coordinates of GUI elements,

[1] In a sample of 1,150 tablet apps, we found that about 56.5% of the apps were written in C#, 38.3% in HTML/Javascript, and 5.2% in C++.

complicating the determination of whether a non-ad element is rendered *above* an ad. We describe in Section 4.4 how DECAF deals with this.

Intrusive Ads: The detector returns true if the distance between an ad control and a clickable non-ad element is below a predefined threshold or if an ad control partially covers a clickable non-ad control. Detecting the latter can also be challenging since the automation framework does not provide z-coordinates of UI elements. We describe in Section 4.4 how DECAF deals with this.

Many Ads: The detector returns true if the number of viewable ads in a screen is more than k, the maximum allowed number of ads. This can be challenging due to the mismatch of apps' page size and device's screen size. To address the sliding screen problem discussed before, a naïve solution would check all possible screen positions in the page and see if there is any violation at any position. We propose a more efficient solution in Section 4.4.

Inappropriate Context: The detector returns true if an ad-containing page has inappropriate content (e.g., adult content) or if the app category is inappropriate. Detecting whether or not page content is inappropriate is outside the scope of the paper; DECAF uses an existing system[2] that employs a combination of machine-classifiers and human inputs for content classification.

4 Optimizations for Coverage and Speed

The basic system described in Section 3 can explore most states of a given app[3]. However, this may take a long time: as [33] reports, this can take several hours for apps designed to have simple UIs for in-vehicle use, and our work considers content-rich tablet apps for which naïve exploration can take significantly longer. This is not practical when the goal is to scan thousands of apps. In such cases, the Monkey will have a limited *time budget*, say few tens of minutes, to scan each app; indeed, in DECAF, users specify a time budget for each app, and the Monkey explores as many states as it can within that time. With limited time, naïve exploration can result in poor *coverage* of the underlying state transition graph, and consequent inaccuracy in ad fraud detection. In this section, we develop various techniques to address this problem. The techniques fall under three general categories that we describe next.[4]

[2]Microsoft's internal system, used by its online services.

[3]Without any human involvement, however, the Monkey can fail to reach states that require human inputs such as a login and a password.

[4]In a companion technical report [34], we discuss an alternative approach to detecting ad fraud using "smart" ad controls that are aware of, and enforce, ad network placement policies. Such ad controls pose many research challenges, since they need to permit communication among ad controls on the same page (to collaboratively detect placement violations) and this can compromise security.

4.1 Detecting Equivalent States

To optimize coverage, a commonly used idea is that after the Monkey detects that it has already explored a state equivalent to the current state, it can backtrack without further exploring the current state (and other states reachable from it). Thus, a key determinant of coverage is the definition of state equivalence. Prior work [33] points out that using a strict definition, where two states are equivalent if they have an identical UI layout, may be too restrictive; it advocates a heuristic for UI lists that defines a weaker form of equivalence.

DECAF uses a different notion of state equivalence, dictated by the following requirements. First, *the equivalence should be decided based on fuzzy matching* rather than exact matching. This is because even within the same run of the Monkey, the structure and content of the "same" state can slightly vary due to dynamic page content and variability in network conditions.

Second, *the equivalence function should be tunable* to accommodate a wide range of fraud detection scenarios. For detecting contextual frauds, the Monkey may want to explore all (or as many as possible within a given time budget) distinct pages of an app, so that it can check appropriateness of all contents of the app. In such a case, two states are equivalent only if they have the same content. For detecting structural frauds, on the other hand, the Monkey may want to explore only the pages that have unique structure (i.e., layout of UI elements). In such cases, two states with the same structure are equivalent even if their contents differ. How much fuzziness to tolerate for page structure and content should also be tunable: the ad network may decide to scan some "potentially bad" apps more thoroughly than the others (e.g., because their publishers have bad histories), and hence can tolerate less fuzziness on those potentially bad apps.

DECAF achieves the first requirement by using a flexible equivalence function based on cosine similarity of feature vectors of states. Given a state, it extracts various features from the visible elements in the DOM tree of the page. More specifically, the name of a feature is the concatenation of a UI element type and its level in the DOM tree, while its value is the count and total size of element contents (if the element contains text or image). For example, the feature `(TextControl@2, 100, 2000)` implies that the page contains 100 `Text` UI elements of total size 2000 bytes at level 2 of the DOM tree of the page. By traversing the DOM tree, DECAF discovers such features for all UI element types and their DOM tree depths. This gives a feature vector for the page that looks like: [(`Image`@2, 10, 5000), (`Text`@1, 10, 400), (`Panel`@2, 100, `null`),...].

To compare if two states are equivalent, we compute cosine similarity of their feature vectors and consider them to be equivalent if the cosine similarity is

above a threshold. This configurable threshold achieves our second requirement; it acts as a tuning parameter to configure the strictness of equivalence. At one extreme, a threshold of 1 specifies *content* equivalence of two states[5]. A smaller threshold implies a more relaxed equivalence, fewer states to be explored by the Monkey, and faster exploration of states with less fidelity in fraud detection. To determine *structural* equivalence of two states, we ignore the size values in feature vectors and use a smaller threshold to accommodate slight variations in page structures. Our experiments indicate that a threshold of 0.92 strikes a good balance between thoroughness and speed while checking for structural frauds.

4.2 Path Prioritization

Many (especially tablet) apps contain too many states for a Monkey to explore within a limited time budget. Indeed, some apps may even contain a practically infinite number of pages to explore, e.g., a *news* app whose content can dynamically change *while* the monkey is exploring the app. Given that the Monkey can explore only a fraction of app pages, without careful design, the Monkey can waste its time exploring states that do not add value to ad fraud detection, and so may not have time to explore useful states. To address this, DECAF uses a novel *state equivalence prediction* method to prioritize which paths to traverse in the UI graph for detecting structural fraud, and a novel *state importance assessment* for detecting contextual fraud.

4.2.1 State Equivalence Prediction

To motivate state equivalence prediction, consider exploring all structurally distinct pages of a news-serving app. Assume that the Monkey is currently in state P_0, which contains 100 news buttons (leading to structurally equivalent states $P_{0,0} \cdots P_{0,99}$) and one video button (leading to $P_{0,100}$). The Monkey could click the buttons in the same order as they appear in the page. It would first recursively explore state $P_{0,0}$ and its descendent states, then visit all the $P_{0,1-99}$ states, realize that that they are all equivalent to already visited state $P_{0,1}$, return to P_0. Finally, it will explore $P_{0,100}$ and its descendant states. This is clearly sub-optimal, since the time required to (1) go from P_0 to each of the states $P_{0,1-99}$ (forward transition) and (2) then backtracking to P_0 (backward transition) is wasted. The forward transition time includes the time for the equivalent page to completely load (we found this to be as large as 30 secs).

Backward transitions can be expensive. The naïve strategy above can also be pathologically sub-optimal in some cases. Most mobile devices do not have a physical back button, so apps typically include software back buttons and our Monkey uses various heuristics based on their screen location and name to identify them. However, in many apps, the Monkey can fail to automatically identify the back button (e.g., if they are placed in unusual locations in the page and are named differently). In such cases the Monkey does not have any obvious way to directly go back to the previous page, creating unidirectional edges in the state graph. In our example, if the transition between P_0 and $P_{0,1}$ is unidirectional, the backward transition would require the Monkey to restart the app and traverse through all states from the root to P_0, while waiting for each state to load completely before moving to the next state. Overall, the wasted time per button is as high as *3 minutes* in some of our experiments, and this can add up to a huge overhead if there are many such pathological traversals.

The net effect of above overheads is that the Monkey can run out of time before it gets a chance to explore the distinct state $P_{0,100}$. A better strategy would be to first explore pages with different UI layouts ($P_{0,0}$ and $P_{0,100}$ in previous example), and then if the time budget permits, to explore remaining pages.

Minimizing state traversal overhead using prediction. These overheads could have been minimized if there was a way to *predict* whether a unidirectional edge would take us to a state equivalent to an already visited state. Our *state equivalence prediction* leverages this intuition, but in a slightly different way. On a given page, it determines which buttons would likely lead to the same (or similar) states, and then explores more than one of these buttons *only if the time budget permits*. Thus, in our example, if the prediction were perfect, it would click on the button leading to the video page $P_{0,100}$ before clicking on the second (and third and so on) news button.

One might attempt to do such prediction based on event handlers invoked by various clickable controls, assuming that buttons leading to equivalent states will invoke the same event handler and those leading to different states will invoke different handlers. The event handler for a control can be found by static analysis of code. This, however, does not always work as event handlers can be bound to controls during run time. Even if the handlers can be reliably identified, different controls may be bound to the same handler that acts differently based on runtime arguments.

DECAF uses a language-agnostic approach that only relies on the run-time layout properties of the various UI elements. The approach is based on the intuition that UI controls that lead to equivalent states have similar "neighborhoods" in the DOM tree: often their parents and children in the UI layout hierarchy are of similar type or have similar names. This intuition, formed by exam-

[5] On rare occasions, two pages with different content can be classified as equivalent if their text (or image) content has exactly the same count and total size. This is because we rely on count and size, instead of contents, of texts and images to determine equivalence of pages.

Control Features	Do they have the same name? Do they have the same ID? Are they with the same UI element type?
Parent Features	Do they have a same parent name path? Do they have a same parent ID path? Do they have a same parent UI element type path?
Child Features	Do their children share a same name set? Do their children share a same ID set? Do their children share a same UI element type set?

Table 1: *SVM classifier features*

ining a number of apps, suggests that it might be possible to use machine-classification to determine if two UI controls are likely to lead to the same state.

Indeed, our approach uses supervised learning to construct a binary classifier for binary feature vectors. Each feature vector represents a pair of UI controls, and each element in the feature vector is a Boolean answer to the questions listed in Table 1. For any two UI controls, these questions can be answered from the DOM tree of the page(s) they are in. We construct a binary SVM classifier from a large labelled dataset; the classifier takes as input the feature vector corresponding to two UI controls, and determines whether they are likely to lead to equivalent states (if so, the UI controls are said to be equivalent).

In constructing the classifier, we explored various feature definitions, and found ones listed in Table 1 to be most accurate. For instance, we found that features directly related to a control's appearance (e.g., color and size) are not useful for prediction because they may be different even for controls leading to equivalent states.

Our Monkey uses the predictor as follows. For every pair of UI controls in a page, the Monkey determines whether that pair is likely to lead to the same state. If so, it clusters the UI controls together, resulting in a set of clusters each of which contains equivalent controls. Then, it picks one control (called the *representative control*) from each cluster and explores these; the order in which they are explored is configurable (e.g., increasing/decreasing by their cluster size, or randomly). The Monkey then continues its depth-first state exploration, selecting only representative controls in each state traversed. After all pages have been visited by exploring only representative controls, the Monkey visits the non-representative controls if the time budget permits. Note that the SVM-based clustering is also robust to dynamically changing pages—since the Monkey explores controls based on their clusters, it can simply choose whatever control is available during exploration and can ignore the controls that have disappeared between the time clusters were computed and when the Monkey is ready to click on a control.

4.2.2 State Importance Assessment

State prediction and fuzzy state matching does not help with state equivalence computed based on page content, as is required for contextual fraud detection. In such cases, the Monkey needs to visit all content-wise distinct pages in an app, and apps may contain too many pages to be explored within a practical time limit.

DECAF exploits the observation that not all pages within an app are equally important. There are pages that users visit more often and spend more time than others. From ad fraud detection point, it is more important to check those pages first, as those pages will show more ads to users. DECAF therefore prioritizes its exploration of app states based on their "importance"—more important pages are explored before less important ones.

Using app usage for estimating state importance. The importance of a state or page is an input to DECAF and can be obtained from app usage statistics from real users, either by using data from app analytic libraries such as Flurry [5] and AppInsight [46] or by having users use instrumented versions of apps.

From this kind of instrumentation, it is possible to obtain a collection of *traces*, where each trace is a path from the root state to a given state. The importance of a state is determined by the number of traces that terminate at that state. Given these traces as input, DECAF combines the traces to generate a *trace graph*, which is a subgraph of the state transition graph. Each node in the trace graph has a *value* and a *cost*, where value is defined as the importance of the node (defined above) and cost is the average time for the page to load.

Prioritizing state traversal using state importance. To prioritize Monkey traversal for contextual fraud, DECAF solves the following optimization problem: given a cost budget B (e.g., total time to explore the app), it determines the set of paths that can be traversed within time B such that total value of all nodes in the paths is maximized. The problem is NP-Hard, by reduction from Knapsack, so we have evaluated two greedy heuristics for prioritizing paths to explore: (1) *Best node*, which chooses the next unexplored node with the best value to cost ratio, and (2) *Best path*, which chooses the next unexplored path with the highest total value-total cost ratio. We evaluate these heuristics in Section 6.

Since app content can change dynamically, it is possible that a state in a trace disappears during the Monkey's exploration of an app. In that case, DECAF uses the trained SVM to choose another state similar to the original state. Finally, traces can be useful not only to identify important states to explore, but also to navigate to states that require human inputs. For example, if navigating to a state requires username/password or special text inputs that the Monkey cannot produce, and if traces incorporate such inputs, the Monkey can use them during its exploration to navigate to those states.

4.3 Page Load Completion Detection

Mobile apps are highly asynchronous but UI extraction channels typically do not provide any callback to an

external observer when the rendering of a page completes. Therefore, DECAF has no way of knowing when the page has loaded in order to check state equivalence. Fixed timeouts, or timeouts based on a percentile of the distribution of page load times, can be too conservative since these distributions are highly skewed. App instrumentation is an alternative, but has been shown to be complex even for managed code such as C# or Java [46].

DECAF uses a simpler and app language-agnostic technique. It monitors all I/O (Networking, Disk and Memory) activities of the app process, and maintains their sum over a sliding window of time T. If this sum goes below a configurable threshold ε, the page is considered to be loaded; the intuition here is that as long as the page is loading, the app should generate non-negligible I/O traffic. The method has the virtue of simplicity, but comes at a small cost of latency, given by sliding window length, to detect the page load.

4.4 Fraud Checker Optimizations

DECAF incorporates several scalability optimizations as part of its fraud checkers.

Detecting too many ads. As mentioned in Section 3, detecting whether a page contains more than k ads in any screen position can be tricky. DECAF uses an efficient algorithm whose computational complexity depends only on the total number of ads N in the page and not on the page or screen size. The algorithm uses a vertical moving window across the page whose width is equal to the screen width and height is equal to the page height; this window is positioned successively at the right edges of rectangles representing ads. Within each such window, a horizontal strip of height equal to the screen height is moved from one ad rectangle bottom-edge to the next; at each position, the algorithms computes the number of ads visible inside the horizontal strip, and exits if this number exceeds a certain threshold. The complexity of this algorithm (see [34] for pseudocode), is $O(N^2 \log(N))$.

Detecting hidden and intrusive ads. As discussed in Section 3, determining if an ad is completely hidden or partially overlapped by other GUI elements is challenging due to missing z-coordinates of the elements. To deal with that, DECAF uses two classifiers described below.

Exploiting DOM-tree structure. This classifier predicts relative z-coordinates of various GUI elements based on their rendering order. In Windows, rendering order is the same as the depth-first traversal order of the DOM tree; i.e., if two elements have the same x- and y-coordinates, the one at the greater depth of the DOM tree will be rendered over the one at the smaller depth. The classifier uses this information, along with x- and y-coordinates of GUI elements as reported by the automation framework, to decide if an ad element is hidden or partially overlapped by a non-ad element.

This classifier is not perfect. It can occasionally classify a visible ad as hidden (i.e., false positives) when GUI elements on top of the ad are invisible and their visibility status is not available from the DOM information.

Analyzing screenshots. This approach uses image processing to detect if a target ad is visible in the app's screenshots. It requires addressing two challenges: (1) *knowing what the ad looks like*, so that the image processing algorithm can search for target ad, and (2) *refocusing*, i.e., making sure that the screenshot captures the region of the page containing the ad.

To address the first challenge, we use a proxy that serves the apps with *fiducials*: dummy ads with easily identifiable (e.g., checker-board) patterns. The proxy intercepts all requests to ad servers and replies with fiducials without affecting normal operation of the app. The image processing algorithm then looks for the specific pattern in screenshots. To address the refocusing challenge, the Monkey scrolls and pans app pages and analyzes screenshots only when the current screen of the page contains at least one ad. The classifier, like the previous one, is not perfect. It can classify hidden ads as visible and vice versa due to errors in image processing and to the failure of the Monkey to refocus.

Combining the classifiers. The two classifiers described above can be combined. In our implementation, we take a conservative approach and declare an ad to be hidden if it is classified as hidden by both the classifiers.

5 Implementation

Tablet/Phone differences. We have implemented DECAF for Windows Phone apps (hereafter referred to as *phone apps*) and Windows Store apps (referred to as *tablet apps*). One key difference between our prototypes for these two platforms is how the Monkey interacts with apps. Tablet apps run on Windows 8, which provides Windows UI automation framework (similar to Android MonkeyRunner [2]); DECAF uses this framework directly. Tablet apps can be written in C#, HTML/JavaScript, or C++ and the UI framework allows interacting with them in a unified way. On the other hand, DECAF runs phone apps in Windows Phone Emulator, which does not provide UI automation, so we use techniques from [44] to extract UI elements in current page and manipulate mouse events on the host machine running the phone emulator to interact with apps. Unlike Windows Phone, tablets do not have any physical back button, so DECAF uses various heuristics to identify software back button defined within the app.

Other implementation details. We use Windows performance counter [16] to implement the page load monitor. To train the SVM classifier for state equivalence, we

manually generated 1,000 feature vectors from a collection of training apps and used grid search with 10-fold cross validation to set the model parameters. The chosen parameter set had a highest cross-validation accuracy of 98.8%. For the user study reported in the next section, we use Windows Hook API [14] and Windows Input Simulator [15] to record and replay user interactions.

6 Evaluation

In this section, we evaluate the overall performance of DECAF optimizations. For lack of space, we limit the results to tablet apps only, since they are more complex than phone apps. Some of our results are compared with ground truth, which we collect from human users; but since the process is not scalable, we limit our study to the 100 top free apps (29 HTML/JavaScript apps and 71 C# apps) from the Windows Store. In the next section, we run DECAF on a larger set of phone and tablet apps to characterize ad frauds.

6.1 State Equivalence Prediction

To measure accuracy of our SVM model, we use humans to find the ground truth. We gave all the 100 apps to real users, who explored as many unique pages as possible[6]. For each app, we combined all pages visited by users and counted the number of structurally distinct pages. Since apps typically have a small number of structurally distinct pages (mean 9.15, median 5), we found humans to be effective in discovering all of them.

Accuracy of SVM model. We evaluated our SVM model on the ground truths and found that it has a false positive rate of 8% and false negative rate of 12%[7]. Note that false negatives do not affect the accuracy of the Monkey; it only affects the performance of the Monkey by unnecessarily sending it to equivalent states. However, false positives imply that the Monkey ignores some distinct states by mistakenly assuming they are equivalent states. To deal with this, we keep the Monkey running and let it explore the remaining states in random order until the time budget is exhausted. This way, the Monkey gets a chance to explore some of those missed states.

Benefits of using equivalence prediction. To demonstrate this, we use an SVM Monkey, with prediction enabled, and a Basic Monkey, that does not do any prediction and hence realizes a state is equivalent only after visiting it. We run each app twice, once with each version of the Monkey for 20 minutes. We measure the Monkey's state exploration performance using a *structural coverage* metric, defined as the fraction of structurally distinct states the Monkey visits, compared with the ground truth found from real users.

[6]This also mimics manual app inspection for frauds.

[7]We emphasize that these rates are not directly for fraud detection, but for the optimization of state equivalence prediction in § 4.

Figure 3(a) shows the structural coverage of the basic and the SVM Monkey, when they are both given 20 minutes to explore each app. In this graph, lower is better: the SVM Monkey achieves perfect coverage for 71% of the apps, while the basic Monkey achieves perfect coverage for only 30% of the apps. Overall, the mean and median coverages of the SVM Monkey are 92% and 100% respectively, and its mean and median coverage improvements are 20.37% and 26.19%, respectively.

Figure 3(b) shows median coverage of the SVM and the basic Monkey as a function of exploration time per app (the graph for mean coverage looks similar, and hence is omitted). As shown, the SVM monkey achieves better coverage for other time limits as well, i.e., for a given target coverage, the SVM Monkey runs much faster than the basic Monkey. For example, the basic Monkey achieves a median coverage of 66% in 20 minutes, while the SVM Monkey achieves a higher median coverage of (86%) in only 5 mins.

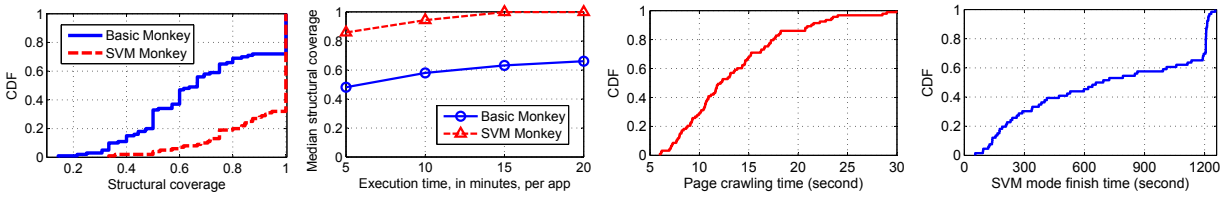
The SVM Monkey fails to achieve perfect coverage for 29 of the 100 apps we tried, for these reasons: the Windows Automation Framework occasionally fails to recognize a few controls; some states require app-specific text inputs (e.g., a zipcode for location-based search) that our Monkey cannot handle; and some apps just have a large state transition graph. Addressing these limitations is left to future work, but we are encouraged by the coverage achieved by our optimizations. Below, we demonstrate that DECAF can be scaled to a several thousand apps, primarily as a result of these optimizations.

6.2 Assessing State Importance

We now evaluate the best node and path strategies (Section 4.2.2) with two baselines that do not exploit app usage statistics: *random node*, where the Monkey chooses a random unexplored next state and *random path*, where the Monkey chooses a random path in the state graph.

To assign values and costs of various app states, we conducted a user study to obtain app usage information. We picked five random apps from five different categories and for each app, we asked 16 users to use it for 5 minutes each. We instrumented apps to automatically log which pages users visit, how much time they spend on each page, how long each page needs to load, etc. Based on this log, we assigned each state a value proportional to the total number of times the page is visited and a cost proportional to the average time the page took to load. To compare various strategies, we use the metric *value coverage*, computed as the ratio of the sum of values of visited states and that of all states.

We ran the Monkey with all four path prioritization strategies for total exploration time limits of 5, 10, 20 and 40 minutes. As Figure 4 shows, value coverage increases monotonically with exploration time. More im-



(a) Structural coverage in 20 minutes of exploration per app

(b) Median coverage as a function of exploration time

(c) Page crawling speed

(d) App crawling speed

Figure 3: *CDF of evaluation result*

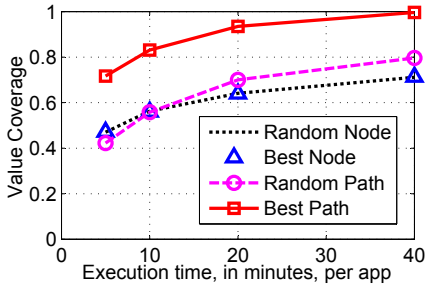


Figure 4: *Value coverage vs. exploration time per app, for various prioritization algorithms.*

portantly, path-based algorithms outperform node-based algorithms, as they can better exploit global knowledge of values of entire paths; the best-path algorithm significantly outperforms the random-path algorithm (average improvement of 27%), highlighting the value of exploiting app usage. Furthermore, exploring all (content-wise) valuable states of an app can take longer than exploring only structurally distinct states. For the set of apps we use in this experiment, achieving a near-complete coverage takes the Monkey 40 minutes.

6.3 Overall Speed and Throughput

Figure 3(c) shows the CDF of time the Monkey needs to explore one app state, measured across all 100 apps. This time includes all the operations required to process the corresponding page: waiting for the page to be completely loaded, extracting the DOM tree of the page, detecting structural fraud in the state, and deciding the next transition. The mean and median times to explore a page is 13.5 and 12.1 sec respectively; a significant component of this time is the additional 5-second delay in detecting page load completion as discussed in Section 4.3. We are currently exploring methods to reduce this delay. Figure 3(d) shows the CDF of time DECAF needs to explore one app. The CDF is computed over 71 apps that the Monkey could finish exploring within 20 minutes. The mean and median time for an app is 11.8 minutes and 11.25 minutes respectively; at this rate, DECAF can scan around 125 apps on a single machine per day.

Table 2: *Occurrences of various fraud types among all fraudulent apps*

Fraud type	Phone Apps	Tablet Apps
Too many (Structural/impression)	13%	4%
Too small (Structural/impression)	40%	54%
Outside screen (Structural/impression)	19%	4%
Hidden (Structural/impression)	39%	32%
Structural/Click	11%	18%
Contextual	2%	20%

7 Characterizing Ad Fraud

In this section, we characterize the prevalence of ad frauds, compare frauds by type across phone and tablet apps, and explore how ad fraud correlates with app rating, size, and several other factors. To obtain these results, we ran DECAF on 50,000 Windows Phone apps (*phone apps*) and 1,150 Windows 8 apps (*tablet apps*).[8] The Windows 8 App Store prevents programmatic app downloads, so we had to manually download the apps before running them through DECAF, hence the limit of 1,150 on tablet apps. Phone apps are randomly chosen from all SilverLight apps in the app store in April 2013. Microsoft Advertising used DECAF after April 2013 to detect violations in apps and force publishers into compliance, and our results include these apps. Tablet apps were downloaded in September 2013, and were taken from top 100 free apps in 15 different categories.

Fraud by Type. Our DECAF-based analysis reveals that ad fraud is widespread both in phone and in tablet apps. In the samples we have, we discovered more than a thousand phone apps, and more than 50 tablet apps, with at least one instance of ad fraud; the precise numbers are proprietary, and hence omitted.

Table 2 classifies the frauds by type (note that an app may include multiple types of frauds). Apps exhibit all of the fraud types that DECAF could detect, but to varying degrees; manipulating the sizes of ads, and hiding ads under other controls seem to be the most prevalent both on the phone and tablet. There are, however, interesting differences between the two platforms. Contextual

[8] For Windows Phone, we consider SilverLight apps only. Some apps especially games are written in XNA that we ignore. Also, the Monkey is not yet sophisticated enough to completely traverse games such as Angry Birds. For Tablet apps, we manually sample simple games from the Games category.

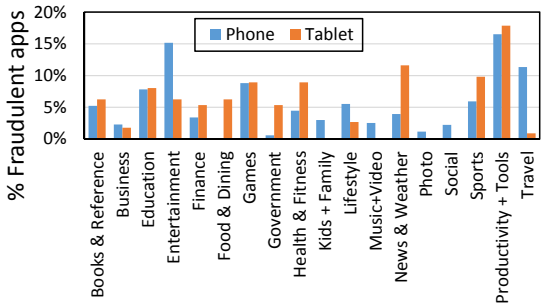


Figure 5: *Fraudulent apps in various categories.*

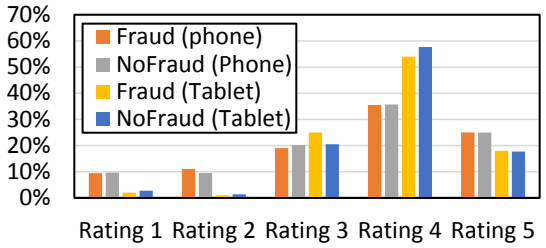


Figure 6: *Distribution of ratings for fraudulent and non-fraudulent phone and tablet apps*

fraud is significantly more prevalent on the tablet, because tablet apps are more content-rich (due to the larger form factor). Ad count violations are more prevalent on the phone, which has a stricter limit (1 ad per screen) compared to the tablet (3 ads per screen).

Fraud by App Category. App stores classify apps by category, and Figure 5 depicts distribution of ad fraud frequency across app categories for both phone and tablet apps. In some cases, fraud is equally prevalent across the two platforms, but there are several instances where fraud is more prevalent in one platform than the other. For instance, navigation and entertainment (movie reviews/timings) based apps exhibit more fraud on the phone, likely because they are more frequently used on these devices and publishers focus their efforts on these categories. For a similar reason, tablets show a significantly higher proportion of fraud than phones in the Health, News and Weather, and Sports categories.

Frauds by rating. We also explore the prevalence of fraud by two measures of the utility of an app. The first measure is its rating value, rounded to a number from 1-5, and we seek to understand if fraud happens more often at one rating level than at another. Figure 6 plots the frequency of different rating values across both fraudulent and non-fraudulent apps, both for the phone and the tablet. One interesting result is that the distribution of rating values is about the same for fraudulent and non-fraudulent apps; i.e., for a given rating, the proportion of fraudulent and non-fraudulent apps is roughly the same. Fraudulent and non-fraudulent phone apps have average ratings of 1.8 and 1.98. For tablet apps, the average ratings are 3.79 and 3.8, for fraudulent and non-fraudulent apps respectively[9]. If rating is construed as a proxy for utility, this suggests that the prevalence of fraud seems to be independent of the utility of an app.

A complementary aspect of apps is *popularity*. While we do not have direct measures of popularity, Figure 7 plots the cumulative distribution of rating counts (the number of ratings an app has received) for phone apps, which has been shown to be weakly correlated with downloads [21] and can be used as a surrogate for popularity (the graphs look similar for tablet apps as well). This figure suggests that there are small distributional differences in rating counts for fraudulent and non-fraudulent apps; the mean rating counts for phone apps is 83 and 118 respectively, and for tablet apps is 136 and 157 respectively. However, these differences are too small to make a categorical assertion of the relationship between popularity and fraud behavior.

We had expected to find at least that less popular apps, or those with lower utility would exhibit fraud behavior, since they have more incentive to do so. These results are a little inconclusive and suggest that our intuitions are wrong, or that we need more direct measures of popularity (actual download counts) to establish the relationship.

The propensity of publishers to commit fraud. Each app in an app store is developed by a *publisher*. A single publisher may publish more than one app, and we now examine how the instances of fraud are distributed across publishers. Figure 8 plots the compliance rate for phone and tablet apps for publishers who have more than one app in the app store. A compliance rate of 100% means that no frauds were detected across all of the publisher's apps, while a rate of 0% means all the publisher's apps were fraudulent. The rate of compliance is much higher in tablet apps, but that may also be because our sample is much smaller. The phone app compliance may be more reflective of the app ecosystem as a whole: a small number of publishers never comply, but a significant fraction of publishers commit fraud on some of their apps. More interesting, the distribution of the number of frauds across publishers who commit fraud exhibits a heavy tail (Figure 9): a small number of publishers are responsible for most of the fraud.

Takeaways. These measurement results are actionable in the following way. Given the scale of the problem, an ad network is often interested in selectively investing resources in fraud detection, and taken together, our results suggest ways in which the ad network should, and should not, invest resources wisely. The analysis of fraud prevalence by type suggests that ad networks could preferentially devote resources to different types of fraud on different platforms; for instance, the ad count

[9] Average ratings for tablet apps are higher than that for phone apps because we chose top apps for tablet.

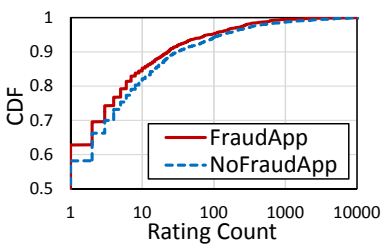


Figure 7: *CDF of of rating counts for phone apps*

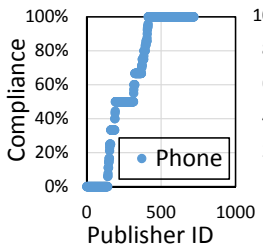
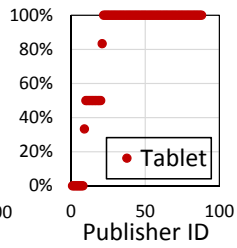


Figure 8: *Compliance rate of publishers with multiple apps*



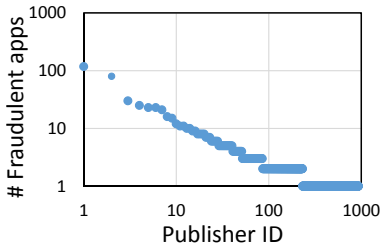


Figure 9: *Fraudulent app count per phone app publisher*

and contextual frauds constitute the lion's share of frauds on tablets, so an ad network may optimize fraud detection throughput by running only these checkers. Similarly, the analysis of fraud by categories suggests categories of apps to which ad networks can devote more resources, and points out that these categories may depend on the platforms. The analysis also points out that ad networks should not attempt to distinguish by rating or rating count. Finally, and perhaps most interesting, the distribution of fraud counts by publisher suggests that it may be possible to obtain significant returns on investment by examining apps from a small set of publishers.

8 Related Work

DECAF is inspired by prior work on app automation and ad fraud detection.

App Automation. Today, mobile platforms provide UI automation tools [3, 2] to test mobile apps. But these tools rely on the developer to provide automation scripts, and do not provide any visibility into the app runtime, so cannot be easily used for detecting ad frauds.

Recent research efforts have built upon these tools to provide full app automation, but their focus has been on different applications: automated testing [47, 35, 50, 18, 19, 26, 30, 41] and automated privacy and security detection [24, 27, 36, 45]. Automated testing efforts evaluate their system only on a handful of apps and many of their UI automation techniques are tuned to those apps. Systems that look for privacy and security violations execute on a large collection of apps but they only use basic UI automation techniques. Closest to our work is AMC [33], which uses automated app navigation to verify UI properties for vehicular Android apps, but reported exploration times of several hours per app and has been evaluated on 12 apps. In contrast to all of these, DECAF is designed for performance and scale to automatically discover ad frauds violations in several thousand apps.

Ad Fraud. Existing works on ad fraud mainly focus on the click-spam behaviors, *characterizing* the features of click-spam, either targeting specific attacks [17, 20, 40, 42], or taking a broader view [22]. Some work has examined other elements of the click-spam ecosystem: the quality of purchased traffic [49, 52], and the spam profit model [32, 37]. Very little work exists in exploring click-spam in mobile apps. From the controlled experiment, authors in [22] observed that around one third of the mobile ad clicks may constitute click-spam. A contemporaneous paper [25] claimed that they are not aware of any mobile malware in the wild that performs advertising click fraud. Unlike these, DECAF focuses on detecting violations to ad network terms and conditions, and even before potentially fraudulent clicks have been generated.

With regard to detection, most existing works focus on bot-driven click spam, either by analyzing search engine query logs to identify outliers in query distributions [51], characterizing networking traffic to infer coalitions made by a group of bot-driven fraudsters [38, 39], or authenticating normal user clicks to filter out bot-driven clicks [29, 31, 48]. A recent work, Viceroi [23], designed a more general framework that is possible to detect not only bot-driven spam, but also some non-bot driven ones (like search-hijacking). DECAF is different from this body of work and focuses on user-based ad fraud in the mobile app setting rather than the click-spam fraud in the browser setting – to the best of our knowledge, ours is the first work to detect ad fraud in mobile apps.

9 Conclusion

DECAF is a system for detecting placement fraud in mobile app advertisements. It efficiently explores the UI state transition graph of mobile apps in order to detect violations of terms and conditions laid down by ad networks. DECAF has been used by Microsoft Advertising to detect ad fraud and our study of several thousand apps reveals interesting variability in the prevalence of fraud by type, category, and publisher. In the future, we plan to explore methods to increase the coverage of DECAF's Monkey, expand the suite of frauds that it is capable of detecting, evaluate other metrics for determining state importance, and explore attacks designed to evade DECAF and develop countermeasures for these attacks.

Acknowledgements. We thank the anonymous referees and our shepherd Aruna Balasubramanian for their comments. Michael Albrecht, Rich Chapler and Navneet Raja provided valuable feedback on DECAF.

References

[1] AdMob Publisher Guidelines and Policies. http://support.google.com/admob/answer/1307237?hl=en&ref_topic=1307235.

[2] Android Monkeyrunner. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[3] Android UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html.

[4] Bots Mobilize. http://www.dmnews.com/bots-mobilize/article/291566/.

[5] Flurry. http://www.flurry.com/.

[6] Google Admob. http://www.google.com/ads/admob/.

[7] Google Admob: What's the Difference Between Estimated and Finalized Earnings? http://support.google.com/adsense/answer/168408/.

[8] iAd App Network. http://developer.apple.com/support/appstore/iad-app-network/.

[9] Microsoft Advertising. http://advertising.microsoft.com/en-us/splitter.

[10] Microsoft Advertising: Build your business. http://advertising.microsoft.com/en-us/splitter.

[11] Microsoft pubCenter Publisher Terms and Conditions. http://pubcenter.microsoft.com/StaticHTML/TC/TC_en.html.

[12] The Truth About Mobile Click Fraud. http://www.imgrind.com/the-truth-about-mobile-click-fraud/.

[13] Up To 40% Of Mobile Ad Clicks May Be Accidents Or Fraud? http://www.mediapost.com/publications/article/182029/up-to-40-of-mobile-ad-clicks-may-be-accidents-or.html#axzz2ed63eE9q.

[14] Windows Hooks. http://msdn.microsoft.com/en-us/library/windows/desktop/ms632589(v=vs.85).aspx.

[15] Windows Input Simulator. http://inputsimulator.codeplex.com/.

[16] Windows Performance Counters. http://msdn.microsoft.com/en-us/library/windows/desktop/aa373083(v=vs.85).aspx.

[17] S. Alrwais, A. Gerber, C. Dunn, O. Spatscheck, M. Gupta, and E. Osterweil. Dissecting Ghost Clicks: Ad Fraud Via Misdirected Human Clicks. In *ACSAC*, 2012.

[18] D. Amalfitano, A. Fasolino, S. Carmine, A. Memon, and P. Tramontana. Using GUI Ripping for Automated Testing of Android Applications. In *IEEE/ACM ASE*, 2012.

[19] S. Anand, M. Naik, M. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In *ACM FSE*, 2012.

[20] T. Blizard and N. Livic. Click-fraud monetizing malware: A survey and case study. In *MALWARE*, 2012.

[21] P. Chia, Y. Yamamoto, and N. Asokan. Is this App Safe? A Large Scale Study on Application Permissions and Risk Signals. In *WWW*, 2012.

[22] V. Dave, S. Guha, and Y. Zhang. Measuring and Fingerprinting Click-Spam in Ad Networks. In *ACM SIGCOMM*, 2012.

[23] V. Dave, S. Guha, and Y. Zhang. ViceROI: Catching Click-Spam in Search Ad Networks. In *ACM CCS*, 2013.

[24] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX OSDI*, 2010.

[25] A. Felt, Porter, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A Survey of Mobile Malware in the Wild. In *ACM SPSM*, 2011.

[26] S. Ganov, C. Killmar, S. Khurshid, and D. Perry. Event Listener Analysis and Symbolic Execution for Testing GUI Applications. In *ICFEM*, 2009.

[27] P. Gilbert, B. Chun, L. Cox, and J. Jung. Vision: Automated Security Validation of Mobile apps at App Markets. In *MCS*, 2011.

[28] M. Grace, W. Zhou, X. Jiang, and A. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *ACM WiSec*, 2012.

[29] H. Haddadi. Fighting Online Click-Fraud Using Bluff Ads. *ACM Computer Communication Review*, 40(2):21–25, 2010.

[30] C. Hu and I. Neamtiu. Automating GUI Testing for Android Applications. In *AST*, 2011.

[31] A. Juels, S. Stamm, and M. Jakobsson. Combating Click Fraud via Premium Clicks. In *USENIX Security*, 2007.

[32] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamalytics: An Empirical Analysis of Spam Marketing Conversion. In *ACM CCS*, 2008.

[33] K. Lee, J. Flinn, T. Giuli, B. Noble, and C. Peplin. AMC: Verifying User Interface Properties for Vehicular Applications. In *ACM MobiSys*, 2013.

[34] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *University of Southern California Technical Report 13-938*, 2013.

[35] A. MacHiry, R. Tahiliani, and M. Naik. Dynodroid: An Input Generation System for Android Apps. In *ACM FSE*, 2013.

[36] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A Whitebox Approach for Automated Security Testing of Android Applications on the Cloud. In *AST*, 2012.

[37] D. McCoy, A. Pitsillidis, G. Jordan, N. Weaver, C. Kreibich, B. Krebs, G. Voelker, S. Savage, and K. Levchenko. PharmaLeaks: Understanding the Business of Online Pharmaceutical Affiliate Programs. In *USENIX Security*, 2012.

[38] A. Metwally, D. Agrawal, and A. El Abbadi. DETECTIVES: DETEcting Coalition hiT Inflation attacks in adVertising nEtworks Streams. In *WWW*, 2007.

[39] A. Metwally, F. Emekci, D. Agrawal, and A. El Abbadi. SLEUTH: Single-pubLisher attack dEtection Using correlaTion Hunting. In *PVLDB*, 2008.

[40] B. Miller, P. Pearce, C. Grier, C. Kreibich, and V. Paxson. What's Clicking What? Techniques and Innovations of Today's Clickbots. In *IEEE DIMVA*, 2011.

[41] N. Mirzaei, S. Malek, C. Pasareanu, N. Esfahani, and R. Mahmood. Testing Android Apps through Symbolic Execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[42] T. Moore, N. Leontiadis, and N. Christin. Fashion Crimes: Trending-Term Exploitation on the Web. In *ACM CCS*, 2011.

[43] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: a Pragmatic Approach to Model Checking Real Code. In *USENIX OSDI*, 2002.

[44] Suman Nath, Felix Lin, Lenin Ravindranath, and Jitu Padhye. SmartAds: Bringing Contextual Ads to Mobile Apps. In *ACM MobiSys*, 2013.

[45] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic Security Analysis of Smartphone Applications. In *ACM CODASPY*, 2013.

[46] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile App Performance Monitoring in the Wild. In *USENIX OSDI*, 2012.

[47] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. Automatic and Scalable Fault Detection for Mobile Applications. Technical Report MSR-TR-2013-98, Microsoft Research, 2013.

[48] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. Wang, and C. Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *IEEE S & P*, 2012.

[49] K. Springborn, , and P. Barford. Impression Fraud in Online Advertising via Pay-Per-View Networks. In *USENIX Security*, 2013.

[50] W. Yang, M. Prasad, and T. Xie. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *FASE*, 2013.

[51] F. Yu, Y. Xie, and Q. Ke. SBotMiner: Large Scale Search Bot Detection. In *ACM WSDM*, 2010.

[52] Q. Zhang, T. Ristenpart, S. Savage, and G. Voelker. Got Traffic? An Evaluation of Click Traffic Providers. In *WebQuality*, 2011.

I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks

Nikhil Handigol[†]*, Brandon Heller*[†]*, Vimalkumar Jeyakumar*[†]*, David Mazières, Nick McKeown*
{*nikhilh,brandonh*}*@cs.stanford.edu,* {*jvimal,nickm*}*@stanford.edu,* `http://www.scs.stanford.edu/~dm/addr/`
Stanford University, Stanford, CA USA
[†] *These authors contributed equally to this work*

Abstract

The complexity of networks has outpaced our tools to debug them; today, administrators use manual tools to diagnose problems. In this paper, we show how *packet histories*—the full stories of every packet's journey through the network—can simplify network diagnosis. To demonstrate the usefulness of packet histories and the practical feasibility of constructing them, we built NetSight, an extensible platform that captures packet histories and enables applications to concisely and flexibly retrieve packet histories of interest. Atop NetSight, we built four applications that illustrate its flexibility: an interactive network debugger, a live invariant monitor, a path-aware history logger, and a hierarchical network profiler. On a single modern multi-core server, NetSight can process packet histories for the traffic of multiple 10 Gb/s links. For larger networks, NetSight scales linearly with additional servers and scales even further with straightforward additions to hardware- and hypervisor-based switches.

1 Introduction

Operating networks is hard. When networks go down, administrators have only a rudimentary set of tools at their disposal (traceroute, ping, SNMP, NetFlow, sFlow) to track down the root cause of the outage. This debugging toolkit has remained essentially unchanged, despite an increase in distributed protocols that modify network state. Network administrators have become "masters of complexity" [40] who use their skill and experience to divine the root cause of each bug. Humans are involved almost every time something goes wrong, and we are still far from an era of automated troubleshooting.

We could easily diagnose many network problems if we could ask the network about suspect traffic and receive an immediate answer. For example:

1. "Host A cannot talk to Host B. Show me where packets from A intended for B are going, along with any header modifications."
2. "I don't want forwarding loops in my network, even transient ones. Show me every packet that passes the same switch twice."
3. "Some hosts are failing to grab IP addresses. Show me where DHCP traffic is going in the network."
4. "One port is experiencing congestion. Show me the traffic sources causing the congestion."

Today, we cannot "just ask" these questions. Our network diagnosis tools either provide no way to pose such a question, or lack access to the information needed to provide a useful answer. But, these questions could be answered with an omniscient view of every packet's journey through the network. We call this notion a *packet history*. More specifically,

Definition A packet history is the route a packet takes through a network plus the switch state and header modifications it encounters at each hop.

A single packet history can be the "smoking gun" that reveals *why*, *how*, and *where* a network failed, evidence that would otherwise remain hidden in gigabytes of message logs, flow records [8, 34], and packet dumps [15, 18, 32].

Using this construct, it becomes possible to build network analysis programs to diagnose problems. We built four such applications: (1) *ndb*, an interactive network debugger, (2) *netwatch*, a live network invariant monitor, (3) *netshark*, a network-wide packet history logger, and (4) *nprof*, a hierarchical network profiler. The problems described above are a small sample from the set of problems these applications can help solve.

These four applications run on top of a prototype platform we built, called NetSight. With a view of

every packet history in the network, NetSight supports both real-time and postmortem analysis. Applications use *Packet History Filter*, a regex-like language that we developed, to concisely specify paths, switch state, and packet header fields for packet histories of interest. The fact that each application is less than 200 lines of code demonstrates the power of Packet History Filter in NetSight.

NetSight assembles packet histories using *postcards*—event records created whenever a packet traverses a switch. Each postcard contains a copy of the packet header, the switch ID, the output ports, and a version number for the switch forwarding state. To generate postcards, our prototype transparently interposes on the control channel between switches and controllers, and we have tested it with both hardware and software OpenFlow switches.[1]

The challenge for any system offering packet histories is to efficiently and scalably process a stream of postcards into archived, queryable packet histories. Surprisingly, a *single* NetSight server suffices to assemble and store packet histories from packet headers collected at each hop, for every packet that crosses 14 routers in the Stanford backbone network. To support larger networks, NetSight scales out on general-purpose servers—increasing its assembly, query, and storage capabilities linearly with the number of processing cores, servers, and disks. To scale further to bandwidth-heavy enterprise and data center networks, we present two additional NetSight variants. NetSight-SwitchAssist proposes new switch hardware changes to reduce postcard bandwidth, while NetSight-HostAssist spreads postcard and history processing among virtualized servers. In contrast to the naïve method of collecting packet headers that requires 31% bandwidth overhead in the average case (§8), the SwitchAssist and HostAssist designs drastically reduce the bandwidth overhead to 7% and 3%, respectively (§8).

To summarize, our contributions include:

- **Language:** Packet History Filter concisely represents packet histories of interest. (§3)
- **Applications:** a suite of network diagnosis apps built atop the NetSight API. (§4)
- **Platform:** the design (§5), implementation (§6), and evaluation (§7) of NetSight.
- A discussion of the two other designs, NetSight-SwitchAssist and NetSight-HostAssist (§8).

This method of network analysis complements techniques that model network behavior [23, 24]. Rather than *predict* the forwarding behavior of *hypothetical* packets, NetSight shows the *actual* forwarding behavior of *real* packets. NetSight makes no assumptions about the correctness of network control software. However, it assumes that the hardware correctly forwards postcards to the NetSight server(s). If it doesn't, NetSight can flag it as a hardware error, but the information might not be helpful in accurately homing in on the root cause. Thus, NetSight helps network operators, control program developers, and switch implementers to discover and fix errors in firmware or control protocols that cause network elements to behave in unexpected ways.

The source code of our NetSight prototype is publicly available with a permissive license [1]. We encourage the readers to download, use, extend, and contribute to the codebase.

[1] Our prototype uses OpenFlow but the design does not require it.

```
packet [dl_src: 0x123, ...]:
     switch 1: { inport: p0, outports: [p1]
                 mods: [dl_dst -> 0x345]
                 matched flow: 23 [...]
                 matched table version: 3 }
     switch 2: { inport: p0, outports: [p2]
                 mods: []
                 matched flow: 11 [...]
                 matched table version: 7 }
     ...
     switch N: { inport: p0
                 table miss
                 matched table version: 8 }
```

Figure 1: A packet history shows the path taken by a packet along with the modifications and switch state encountered by it at each hop.

2 Motivating Packet Histories

In this section, we define packet histories, show an example, note their challenges, and describe where Software-Defined Networking (SDN) can help.

Packet History Definition. A *packet history* tells the full story of a packet's journey through the network. More precisely, a packet history describes:

- *what* the packet looked like as it entered the network (headers)
- *where* the packet was forwarded (switches + ports)
- *how* it was changed (header modifications)
- *why* it was forwarded that way (matched flow/actions + flow table).

Figure 1 shows an example packet history.

Why Packet Histories? Put simply, packet histories provide direct evidence to diagnose network problems. For example, consider a WiFi handover problem we recently encountered [26]. To diagnose the problem, our network admins started with pings. Then they collected and manually inspected forwarding rules. Then they visually parsed control

plane logs, looking for the problem. After hours of debugging, they diagnosed the (surprisingly simple) cause: upon handover to a new AP, forwarding rules in the upstream wired switch were improperly updated, sending incoming packets to the original AP.

Instead, our admins might simply ask NetSight: "Show me all packet histories for packets to the client when the handover occurred." Each history would have shown a packet going to the wrong AP along with the upstream flow table state that caused the error, enabling an immediate diagnosis.

This example shows how *just one packet history* can single-handedly confirm or disprove a hypothesis about a network problem, by showing events that *actually* transpired in the network, along with all relevant state. *Access to the stream of all packet histories* enables diagnostics that would otherwise be impractical, time-consuming, or impossible for a network administrator using conventional tools.

Challenges. Generating, archiving, and querying packet histories in operational networks requires:

(1) Path Visibility: we must somehow view and record the paths taken by *every* packet.

(2) State Visibility: we must reconstruct the *exact* switch state encountered at each packet hop.

(3) Modification Visibility: we must know where and how each packet has changed.

(4) Collection Scale: all of the above must run at the maximum observed traffic rate.

(5) Storage Scale: querying histories requires storing everything, for some time.

(6) Processing Scale: query processing should keep up with collection and storage.

Observing switch states from an external vantage point, by either logging the control messages or querying the switches for their state, will not guarantee precise state-packet correlation. The only place where packets can be correlated with the exact switch state is the data plane itself [20].

Opportunities with SDN. SDN offers a path to the correlated visibility we need: logically centralized control provides a natural location to modify forwarding rules, while a common representation of switch state enables us to reason about any changes. Later, in §6, we show how to precisely correlate packets with the state used to forward them. We solve the remaining scale concerns with careful system architecture, aggressive packet header compression, and an optimized implementation. Next, we describe our API for specifying packet histories.

3 The NetSight API

NetSight exposes an API for applications to specify, receive, and act upon packet histories of interest. NetSight provides a regular-expression-like language—Packet History Filter (PHF)—to express interest in packet histories with specific trajectories, encountered switch state, and header fields. The main function of the NetSight API is:[2]

`add_filter(packet_history_filter, callback)`

For every packet history matching the PHF `packet_history_filter` , the `callback` function is called along with the complete packet history.

Postcard Filters. The atomic element in a PHF is the *postcard filter* (PF). A PF is a filter to match a packet at a hop. Syntactically, a PF is a conjunction of filters on various qualifiers: packet headers, switch ID (known as datapath ID, or dpid), input port, output port, and the switch state encountered by the packet (referenced by a "version" as described in §5). A PF is written as follows:

```
--bpf [not] <BPF> --dpid [not] <switch ID>
--inport [not] <input port> --outport [not]
 <output port> --version [not] <version>
```

where, `<BPF>` is a Berkeley Packet Filter [30] expression. The `not`s are optional and negate matches. A PF must have at least one of the above qualifiers. For example, a PF for a packet with source IP A, entering switch S at any input port other than port P is written as:

`--bpf "ip src A" --dpid S --inport not P`.

Packet History Filter Examples. A PHF is a regular expression built with PFs, where each PF is enclosed within double braces. The following sample PHFs use X and Y as PFs to match packets that:

- start at X: `^{{X}}`
- end at X: `{{X}}$`
- go through X: `{{X}}`
- go through X, and later Y: `{{X}}.*{{Y}}`
- start at X, never reach Y: `^{{X}}[^{{Y}}]*$`
- experience a loop: `(.).*(\1)`

4 Applications

The ability to specify and receive packet histories of interest enables new network-diagnosis applications. This section demonstrates the utility of the NetSight API by presenting the four applications we built upon it.

[2] The other important function is `delete_filter`.

4.1 ndb: Interactive Network Debugger

The original motivating application for NetSight is ndb, an interactive network debugger. The goal of ndb is to provide interactive debugging features for networks, analogous to those provided by gdb for software programs. Using ndb, network application developers can set PHFs on errant network behavior. Whenever these occur, the returned packet histories will contain the sequence of switch forwarding events that led to the errant behavior, helping to diagnose common bugs like the following:

Reachability Error: Suppose host A is unable to reach host B. Using ndb, the developer would use a PHF to specify packets from A destined for B that never reach the intended final hop:
`^{{--bpf "ip src A and dst B" --dpid X --inport p1}}[^{{--dpid Y --outport p2}}]*$`
where, (X, p1) and (Y, p2) are the (switch, port) tuples at which hosts A and B are attached. Recall that the regular expression '`^X`' matches any string that starts with character X, but '`[^X]`' matches any character except 'X'. Thus, the above PHF matches all packet histories with (source,destination)-IP addresses (A,B) that start at (X,p1) but never traverse (Y,p2).

Race condition: A controller may insert new flow entries on multiple switches in response to network events such as link failures or new flow arrivals. If a controller's flow entry insertions are delayed, packets can get dropped, or the controller can get spurious 'packet-in' notifications. To query such events, NetSight inserts a forwarding rule at the lowest priority in all switches at switch initialization time. This rule generates postcards and performs the default action (by sending to either outport NULL that would drop the packet, or to outport CONTROLLER that would send the packet to the controller). Since this rule is hit only when there is no other matching flow entry, the following PHF captures such events, by matching on packet histories that do not match any flow entry at switch X:
`{{--dpid X --outport NULL}}$`

Incorrect packet modification: Networks with many nodes and rules can make it difficult to see where and why errant packet modifications occurred. Packets reaching the destination with unexpected headers can be captured by the following PHF:
`^{{--bpf "BPF1"}}.*{{--bpf "BPF2"}}$`
Where `BPF1` matches the packet when it enters the network and `BPF2` matches the modified packet when it reached the destination.

4.2 netwatch: Live Invariant Monitor

The second application is `netwatch`, a live network invariant monitor. `netwatch` allows the operator to specify desired network behavior in the form of invariants, and triggers alarms whenever a packet violates any invariant (e.g., freedom from traffic loops). `netwatch` is a library of invariants written using PHFs to match packets that violate those invariants. Once PHFs are pushed to NetSight, the callback returns the packet history that violated the invariant(s). The callback not only notifies the operator of an invariant violation, but the PHF provides useful context around *why* it happened. `netwatch` currently supports the following network invariants:

Isolation: Hosts in group A should not be able to communicate with hosts in group B. Raise an alarm whenever this condition is violated. The function `isolation(a_host_set, b_host_set, topo)` pushes down two PHFs:
`^{{ GroupA }}.*{{ GroupB }}$`
`^{{ GroupB }}.*{{ GroupA }}$`
GroupA and GroupB can be described by a set of host IP addresses or by network locations (switch, port) of hosts. This PHF matches packets that are routed from GroupA to GroupB.

Loop Freedom: The network should have no traffic loops. The function `loop_freedom()` pushes down one PHF: `(.).*(\1)`

Waypoint routing: Certain types of traffic should go through specific waypoints. For example, all HTTP traffic should go through the proxy, or guest traffic should go through the IDS and Firewall. The function `waypoint_routing(traffic_class, waypoint_id)` installs a PHF of the form: `{{--bpf "traffic_class" --dpid not "waypoint_id"}}{{--dpid not "waypoint_id "}}*$`
This PHF catches packet histories of packets that belong to `traffic_class` and never go through the specified waypoint.

Max-path-length: No path should ever exceed a specified maximum length, such as the diameter of the network. The function `max_path_length(n)` installs a PHF of the form: `.{n+1}`
This PHF catches all paths whose lengths exceed n.

4.3 netshark: Network-wide Path-Aware Packet Logger

The third application is `netshark`, a `wireshark`-like application that enables users to set filters on the entire history of packets, including their paths and header values at each hop. For example, a user could look for all HTTP packets with src IP A and dst IP

B arriving at (switch X, port p) that have also traversed through switch Y. `netshark` accepts PHFs from the user, returns the collected packet histories matching the query, and includes a `wireshark` dissector to analyze the results. The user can then view properties of a packet at a hop (packet header values, switch ID, input port, output port, and matched flow table version) as well as properties of the packet history to which it belongs (path, path length, etc.).

4.4 `nprof`: Hierarchical Network Profiler

The fourth application is `nprof`, a hierarchical network profiler. The goal of `nprof` is to 'profile' any collection of network links to understand the traffic characteristics and routing decisions that contribute to link utilization. For example, to profile a particular link, `nprof` first pushes a PHF specifying the link of interest:

```
{{--dpid X --outport p}}
```

`nprof` combines the resulting packet histories with the topology information to provide a live hierarchical profile, showing which switches are sourcing traffic to the link, and how much. The profile tree can be further expanded to show which particular flow entries in those switches are responsible.

`nprof` can be used to not only identify end hosts (or applications) that are congesting links of interest, but also identify how a subset of traffic is being routed across the network. This information can suggest better ways to distribute traffic in the network, or show packet headers that cause uneven load distributions on routing mechanisms such as equal-cost or weighted-cost multi-path.

5 How NetSight Works

In this section, we present NetSight, a platform to realize the collection, storage, and filtering of *all* packet histories, upon which one can build a range of applications to troubleshoot networks.

The astute reader is likely to doubt the scalability of *any* system that attempts to store every header traversing a network, along with its corresponding path, state, and modifications, as well as apply complex filters. This is a lot of data to forward, let alone process and archive.

Hence, NetSight is designed from the beginning to scale out and see linear improvements with increasing numbers of servers. The design implements all software processing, such as table lookups, compression operations, and querying, in ways that are simple enough to enable hardware implementations. As an existence proof that such a system is indeed feasible, the implementation described in §6 and evaluated in §7 can perform all packet history processing and storage steps for a moderately-sized network like the Stanford University backbone network on a single server. For networks with higher aggregate bandwidths, processing capabilities increase linearly with the number of servers.

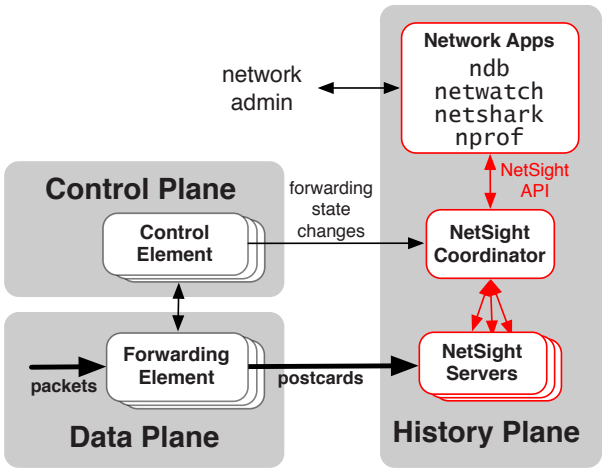


Figure 2: NetSight architecture.

5.1 NetSight Philosophy

NetSight assembles packet histories using *postcards*, event records sent out whenever a packet traverses a switch. This approach decouples the fate of the postcard from the original packet, helping to troubleshoot packets lost down the road, unlike approaches that append to the original packet. Each postcard contains the packet header, switch ID, output port, and current version of the switch state. Combining topology information with the postcards generated by a packet, we can reconstruct the complete packet history: the exact path taken by the packet along with the state and header modifications encountered by it at each hop along the path.

We first explain how NetSight works in the *common case*, where: (1) the network does not drop postcards, (2) the network does not modify packets, and (3) packets are all unicast. Then, in §5.4, we show how NetSight handles these edge cases.

5.2 System Architecture

Figure 2 sketches the architectural components of NetSight. NetSight employs a central coordinator to manage multiple workers (called NetSight servers). NetSight applications issue PHF-based triggers and queries to the coordinator, which then returns a stream or batch of matching packet histories. The coordinator sets up the transmission of postcards from switches to NetSight servers and the transmission of state change records from the network control plane to the coordinator. Finally, the coordi-

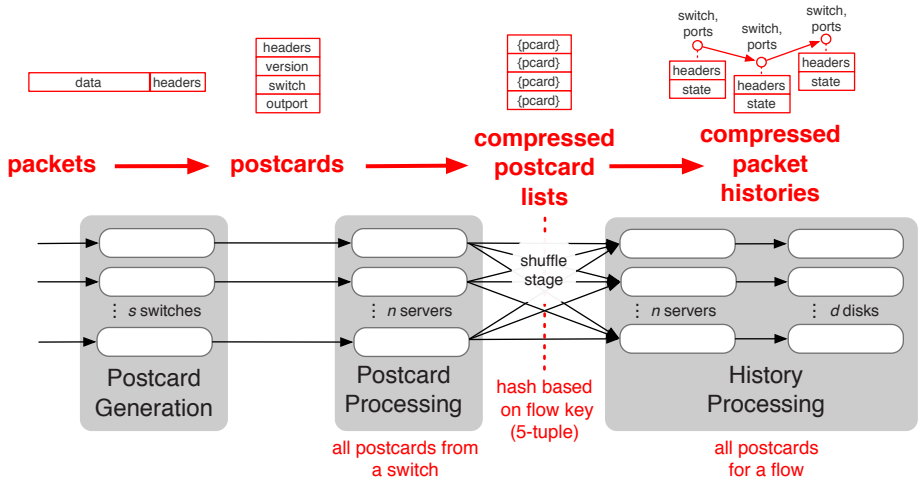


Figure 3: Processing flow used in NetSight to turn packets into packet histories across multiple servers.

nator performs periodic liveness checks, broadcasts queries and triggers, and communicates topology information for the workers to use when assembling packet histories.

5.3 Life Of a Postcard

NetSight turns postcards into packet histories. To explain this process, we now follow the steps performed inside a NetSight server, shown in Figure 3.

Postcard Generation. *Goal: record all information relevant to a forwarding event and send for analysis.* As a packet enters a switch, the switch creates a postcard by duplicating the packet, truncating it to the minimum packet size, marking it with relevant state, and forwarding it to a NetSight server. The marked state includes the switch ID, the output port to which this packet is about to be forwarded, and a version ID representing the exact state of this switch when the packet was forwarded. The original packet remains untouched and continues on its way. Switches today already perform similar packet duplication actions to divert packets for monitoring (e.g. RSPAN [7] and sFlow). Postcard generation should be much faster than normal packet forwarding, because it does not require any expensive IP lookups. It requires encapsulating the packet to a known port and duplicating the packet output; both of these are cheap operations relative to typical IP lookups. Newer switches [17] also support hardware-accelerated encapsulation for tunneling traffic at line-rate (e.g., MPLS, GRE, VXLAN, etc.).

Postcard Collection. *Goal: to send all postcards for a packet to one server, so that its packet history can be assembled.* In order to reconstruct packet histories, NetSight needs to collect all postcards corresponding to a single packet at a single server. To scale processing, NetSight needs to ensure that these groups of postcards are load balanced across servers. NetSight achieves this by shuffling postcards between NetSight servers, using a hash on the flow ID (5-tuple) to ensure postcard locality.

Postcard shuffling is batched into time-based "rounds." At the end of a round, servers send postcards collected during the round to their final destination, where the corresponding packet histories can be assembled and archived. This stage provides an opportunity to compress postcard data before shuffling, by exploiting the redundancy of header values, both within a flow, and between flows. Section 6 details NetSight's fast network-specific compression technique to reduce network bandwidth usage.

History Assembly. *Goal: to assemble packet histories from out-of-order postcards.* Packet histories must be ordered, but postcards can arrive out-of-order due to varying propagation and queuing delays from switches to NetSight servers. NetSight uses topology information, rather than fine-grained timestamps, to place postcards in order.

When a NetSight server has received the complete round of postcards from every other server, it decompresses and merges each one into the Path Table, a data structure that helps combine all postcards for a single packet into a group. To identify all postcards corresponding to a packet, NetSight combines immutable header fields such as IP ID, fragment offset, and TCP sequence number fields into a "packet ID," which uniquely identifies a packet within a flow. To evaluate the strategy of using immutable header fields to identify packets, we analyzed a 400k-packet

trace of enterprise packet headers [28]. Nearly 11.3% of packets were indistinguishable from at least one other packet within a one-second time window. On closer inspection, we found that these were mostly UDP packets with IPID 0 generated by an NFS server. Ignoring these UDP packets removed all IP packet ambiguity, leaving only seven ambiguous ARPs. This initial analysis suggests that most of the packets have enough entropy in their header fields to be uniquely identified. The Path Table is simply a hash table indexed by packet ID, where values are lists of corresponding postcards.

The NetSight server extracts these postcard groups, one-at-a-time, to assemble them into packet histories. For each group, NetSight then performs a topological sort, using switch IDs and output ports, along with topology data.[3] The resulting sorted list of postcards is the packet history.

Filter triggers. *Goal: to immediately notify applications of fresh packet histories matching a pre-installed PHF.* Once the packet history is assembled, NetSight matches it against any "live" PHFs pre-installed by applications such as `netwatch`, and immediately triggers notifications back to the application on a successful match.

History archival. *Goal: to efficiently store the full set of packet histories.* Next, the stream of packet histories generated in each round is written to a file. NetSight compresses these records using the same compression algorithm that is used before the shuffle phase to exploit redundancy between postcards of a packet and between packets of a flow.

Historical queries. *Goal: to enable applications to issue PHF queries against archived packet histories.* When an application issues a historical PHF query to a specified time region, that query runs in parallel on all NetSight servers. Compression helps to improve effective disk throughput here, and hence reduces query completion times.[4]

5.4 Relaxing the Assumptions

We now describe how NetSight handles corner cases.

Dropped Postcards. When postcard drops occur (e.g., due to congestion), packet histories become incomplete, causing NetSight to return errantly matched histories as well as to miss histories that should have matched the installed PHFs. NetSight delegates the responsibility for handling these events to apps. For example, `ndb` returns partial histories to the user, who can often resolve the omission by using the topology information and filling the missing postcards.[5] Out-of-band control links and highest-priority queues for postcards can help to minimize postcard drops.

Non-unicast Packets. For broadcast and multicast traffic, NetSight returns packet histories as directed graphs, rather than lists. For loops, NetSight returns the packet history with an arbitrary starting point and marks it as a loop.

Modified Packets. When Network Address Translation (NAT) boxes modify the header fields in the flow key, the postcards for one packet may arrive at different NetSight servers, preventing complete packet history assembly. Using immutable headers or hashes of packet contents in the shuffle phase would ensure that all postcards for one packet arrive at the same server.[6] However, with such keys, packet histories of packets belonging to a single flow will be evenly spread among servers, reducing opportunities for storage compression: each of n servers will see packet histories of $1/n$-th of the packets of each flow.

Adding a second shuffle stage can ensure both correctness and storage efficiency. In the first stage, packet histories are shuffled for assembly using their packet ID, while in the second stage, they are shuffled for storage using a hash of the 5-tuple flow key of their *first packet*. The reduced storage comes at a cost of additional network traffic and processing.

6 NetSight Implementation

Our NetSight implementation has two processes: one interposes between an OpenFlow controller and its switches to record configuration changes, while another does all postcard and history processing. To verify that it operates correctly on physical switches, we ran it on a chain topology of 6 NEC IP8800 switches [31]. To verify that it ran with unmodified controllers, we tested it on the Mininet emulation environment [27] with multiple controllers (OpenFlow reference, NOX [19], POX [35], RipL-POX [36]) on multiple topologies (chains, trees, and fat trees). This section describes the individual pieces of our prototype, which implements all postcard and history processing in C++ and implements the control channel proxy in Python.

6.1 Postcard Generation

The NetSight prototype is for SDN, leveraging the fact that network state changes are coordinated by

[3] In the current implementation the topology data needs to be externally fed into NetSight. Alternatively, with the SDN implementation described in §6, the proxy can dynamically learn the topology.

[4] Ideally the filesystem is log-structured, to restore individual rounds at the full disk throughput, with minimal seeking [37].

[5] These can indicate an unexpected switch configuration too, as we saw the first time using NetSight on a network (§6.4).

[6] That is, if middleboxes don't mess with packet *contents*.

a controller. This provides an ideal place to monitor and intercept switch configuration changes. It uses a transparent proxy called the *flow table state recorder* (recorder for short) that sits on the control path between the controller and OpenFlow switches.

When a controller modifies flow tables on a switch, the recorder intercepts the message and stores it in a database. For each OpenFlow rule sent by the controller to the switch, the recorder *appends* new actions to generate a postcard for each packet matching the rule in addition to the controller-specified forwarding.

Specifically, the actions create a copy of the packet and tag it with the switch ID,[7] the output port, and a version number for the matching flow entry. The version number is simply a counter that is incremented for every flow modification message. The tag values overwrite the original destination MAC address (the original packet header is otherwise unchanged). Once assembled, postcards are sent to a NetSight server over a separate VLAN. Postcard forwarding can be handled out-of-band via a separate network, or in-band over the regular network; both methods are supported. In the in-band mode, switches recognize postcards using a special VLAN tag to avoid generating postcards for postcards.

6.2 Compression

NetSight compresses postcards in two places: (1) before shuffling them to servers, and (2) before archiving assembled packet histories to disk. While we can use off-the-shelf compression algorithms (such as LZMA) to compress the stream of packets, we can do better by leveraging the structure in packet headers and the fact that all the packets in a flow—identified by the 5-tuple flow id `(srcip, dstip, srcport, dstport, protocol)`—look similar.

NetSight compresses packets by computing diffs between successive packets in the same stream. A diff is a `(Header,Value)` pair, where `Header` uniquely identifies the field that changed and `Value` is its new value. Certain fields (e.g. IPID and TCP Sequence numbers) compress better if we just store the successive *deltas*. Compression works as follows: the first packet of each flow is stored verbatim. Subsequent packets are only encoded as the `(Header,Value)` tuples that change, with a back-reference to the previous packet in the same stream. Finally, NetSight pipes the stream of encoded diffs through a standard fast compression algorithm (e.g. gzip at level 1). Our compression algorithm is a generalization of Van Jacobson's compression of TCP packets over slow links [21].

[7] To fit into the limited tag space, NetSight uses a locally created "pseudo switch ID" (PSID) and maintains an internal mapping from the 8 B datapath ID to the PSID.

6.3 PHF Matching

The PHF matching engine in NetSight is based on the RE1 regex engine [9] and uses the Linux x86 BPF compiler [5] to match packet headers against BPF filters. RE1 compiles a subset (concatenation, alternation and the Kleene star) of regular expressions into byte codes. This byte code implements a Nondeterministic Finite Automaton which RE1 executes on an input string. In RE1, character matches trigger state machine transitions; we modify RE1 and "overload" the character equality check to match postcards against postcard filters.

6.4 Test Deployment Anecdote

NetSight helped to uncover a subtle bug during our initial test deployment. While connectivity between hosts seemed normal, the packet histories returned by `ndb` for packets that should have passed through a particular switch were consistently returned as two partial paths on either side of the switch. These packet histories provided all the context our administrator needed to immediately diagnose the problem: due to a misconfiguration, the switch was behaving like an unmanaged layer-2 switch, rather than an OpenFlow switch as we intended.

With no apparent connectivity issues, this bug would have gone unnoticed, and might have manifested later in a much less benign form, as forwarding loops or security policy violations. This unexpected debugging experience further highlights the power of packet histories.

7 Evaluation

This section quantifies the performance of the server-side mechanisms that comprise NetSight, to investigate the feasibility of collecting and storing every packet history. From measurements of each step, including compression, assembly, and filtering, we can determine the data rate that a single core can handle. For switch-side mechanisms and scaling them, skip to §8.

7.1 Compression

NetSight compresses postcards before the shuffle phase to reduce network bandwidth, then compresses packet histories again during the archival phase to reduce storage costs. We investigate three questions:

Compression: how tightly can NetSight compress packet headers, and how does this compare to off-the-shelf options?

Compression Type	Description
Wire	Raw packets on the wire
PCAP	All IP packets, truncated up to Layer-4 headers
gzip	PCAP compressed by gzip level 6
NetSight (NS)	Van Jacobson-style compression for all IP 5-tuples
NetSight + gzip (NS+GZ)	Compress packet differences with gzip level 1

Table 1: Compression techniques.

Speed: how expensive are the compression and decompression routines, and what are their time vs. size tradeoffs?

Duration: how does the round length (time between data snapshots) affect compression properties, and is there a sweet spot?

Traces. While we do not have a hardware implementation of the compression techniques, we answer the performance questions using thirteen packet capture (pcap) data sets: one from a university enterprise network (UNIV), two from university data centers (DCs), and nine from a WAN. We preprocessed all traces and removed all non-IPv4, non-TCP and non-UDP packets, then stripped packet headers beyond the layer 4 TCP header, which accounted for less than 1% of all traffic. UNIV is the largest trace, containing 31 GB of packet headers collected over an hour on a weekday afternoon. The average *flow* size over the duration of this trace is 76 packets. The data center traces DC1 and DC2 have average flow sizes of about 29 and 333 packets per flow respectively. However, in the WAN traces, we observed that flows, on average, have less than 3 packets over the duration of the trace. We do not know why; however, this extreme point stresses the efficiency of the compression algorithm.

The UNIV trace contains packets seen at one core router connecting Clemson University to the Internet. The data center traces—DC1 and DC2—are from [4] whose IP addresses were anonymized using SHA1 hash. And finally, each WAN trace (from [43]) accounts for a minute of packet data collected by a hardware packet capture device. IP addresses in this trace are anonymized using a CryptoPan prefix-preserving anonymization.

Storage vs CPU costs. Figure 4 answers many of our performance questions, showing the tradeoff between compression storage costs and CPU costs, for different traces and compression methods. This graph compares four candidate methods, listed in Table 1: (a) PCAP: the uncompressed list packet headers, (b) gzip compression directly on the pcap file, (c) NS: the adaptation of Van Jacobson's compression algorithm, (d) NS+GZ: output of (c) followed by gzip compression (level 1, fastest). Each one is lossless with respect to headers; they recover all header fields and timestamps and maintain the packet ordering.

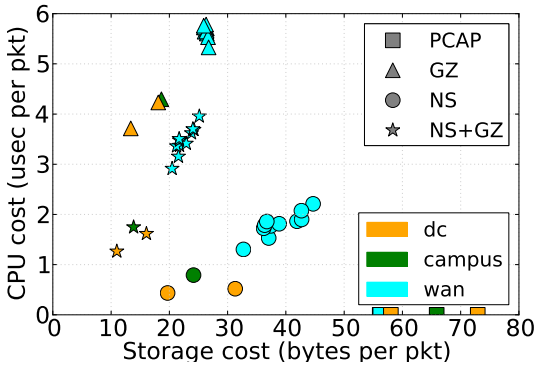


Figure 4: NetSight reduces storage relative to PCAP files, at a low CPU cost. Combining NS with gzip (NS+GZ) reduces the size better than gzip, at a fraction of gzip's CPU costs. The WAN traces compress less as we observe fewer packets in a flow compared to other traces.

We find that all candidates reduce storage relative to PCAP files, by up to 4x, and as expected, their CPU costs vary. GZ, an off-the-shelf option, compresses well, but has a higher CPU cost than both NS and NS+GZ, which leverage knowledge of packet formats in their compression. *NetSight uses NS+GZ, because for every trace, it compresses better than pure GZ, at a reasonably low CPU cost.*

We also find that the compressed sizes depend heavily on the average flow size of the trace. Most of the benefits come from storing differences between successive packets of a flow, and a smaller average flow size reduces opportunities to compress. We see this in the WAN traces, which have shorter flows and compress less. Most of the flow entropy is in a few fields such as IP identification, IP checksums and TCP checksums, and the cost of storing diffs for these fields is much lower than the cost of storing a whole packet header.

To put these speeds in perspective, consider our most challenging scenario, NS+GZ in the WAN, shown by the blue stars. The average process time per packet is 3.5μs, meaning that one of the many cores in a modern CPU can process 285K postcards/sec. Assuming an average packet size of 600 bytes, this translates to about 1.37 Gb/s of network traffic, and this number scales linearly with the number of cores. Moreover, the storage cost (for postcards) is about 6.84 MB/s; a 1 TB disk array can store all

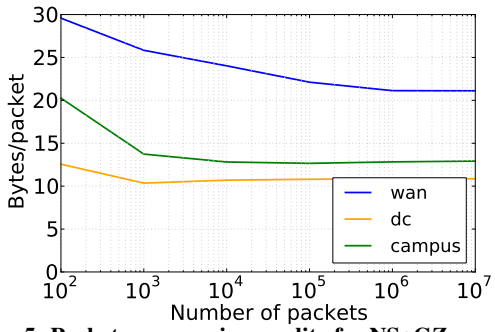


Figure 5: Packet compression quality for NS+GZ as a function of packets seen in the trace. In our traces from three operating environments, we find that NetSight quickly benefits from compression after processing a few 100s of thousands of packets.

Scenario	Enterprise	WAN	Data Center
CPU cost per packet	0.725μs	0.434μs	0.585μs

Table 2: Decompression Speeds.

postcards for an entire day. The actual duration for which postcards need to be stored depends on the scenario and the organizational needs. For example, to troubleshoot routine network crashes whose symptoms are usually instantly visible, storing a day or two worth of postcards might suffice. On the other hand, to troubleshoot security breaches, whose effects might show up much later, postcards might have to be stored for a longer period, say a week. Most of this storage cost goes into storing the first packet of a flow; as the number of packets per flow increases (e.g. in datacenter traces), the storage costs reduce further.

Duration. A key parameter for NetSight is the round length. Longer rounds present more opportunities for postcard compression, but increase the delay until the applications see matching packet histories. Smaller rounds reduce the size of the hash table used to store flow keys in NS compression, speeding up software implementations and making hardware implementations easier. Figure 5 shows packet compression performance as a function of the number of packets in a round. This graph suggests that short rounds of 1000 packets see many of the compression benefits, while long rounds of 1M postcards maximize them. On most lightly loaded 10Gb/s links, a 1M postcard round translates to about a second.

Decompression Speed. Table 2 shows NS+GZ decompression costs for one trace from each of the environments. In every case, NS+GZ decompression is significantly faster than compression. These numbers underrepresent the achievable per-postcard

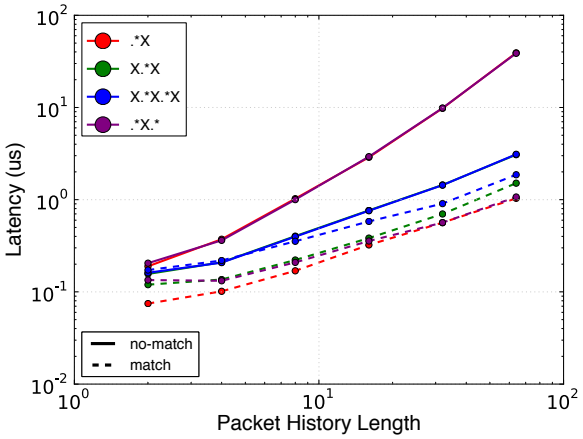


Figure 6: PHF matching latency microbenchmark for various sample PHFs and packet histories of increasing length.

latencies, because the implementation loads the entire set of first packets and timestamps into memory before iterating through the list of diff records. As with compression, a small round timer would improve cache locality and use less memory.

7.2 Packet History Assembly

At the end of the shuffle phase, each NetSight server assembles packet histories by topologically sorting received postcards, which may have arrived out-of-order. We measure the speed of our history assembly module written in C++. Topological sorting is fast – it runs in $O(p)$, where p is the number of postcards in the packet history, and typically, p will be small. For typical packet history lengths (2 to 8 hops long in each of the networks we observed) history assembly takes less than 100 nanoseconds. In other words, a single NetSight server can assemble more than 10 million packet histories per second per core.

7.3 Triggering and Query Processing

NetSight needs to match assembled packet histories against PHFs, either on a live stream of packet histories or on an archive. In this section, we measure the speed of packet history matching using both microbenchmarks and a macrobenchmark suite, looking for where matching might be slowest. The PHF match latency depends on (1) the length of the packet history, (2) the size and type of the PHF, and (3) whether the packet history matches against the PHF.

Microbenchmarks. Figure 6 shows the performance of our PHF implementation for sample PHFs of varying size on packet histories of varying length. The sample PHFs are of the type `.*X`, `.*X.*`, `X.*X`, and `X.*X.*X`, where each `X` is a postcard filter and

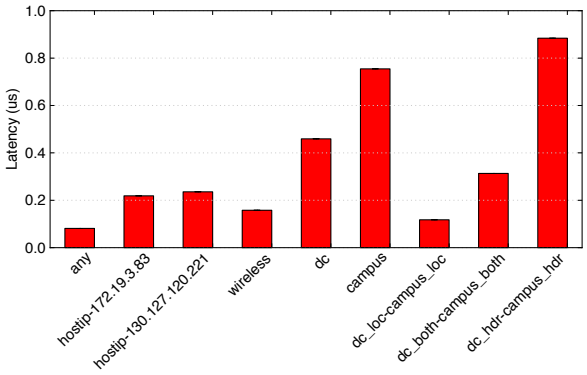


Figure 7: Representative results from the macrobenchmark suite of queries run on the Clemson trace. The most expensive queries were those with complex BPF expressions.

contains filters on packet headers (BPF), switch ID, and input ports. We match a large number of packet histories against each PHF and calculate the average latency per match. In order to avoid any data-caching effects, we read the packet histories from a 6GB file, and we ignore the I/O latency incurred while reading packet histories from the disk.

The dashed lines show the latency when the packet history matches the PHF ("match"), while the solid lines show the latency when the packet history does match the PHF ("no-match"). We see that the "match" latencies are typically smaller than the corresponding "non-match" latencies, since the code can return as soon as a match is detected. We also see that the match latency increases with the number of PFs in the PHF as well as the length of the packet history. Importantly, the region of interest is the bottom left corner – packet histories of length 2 to 8. Here, the match latency is low: a few hundred nanoseconds.

Macrobenchmarks. The UNIV trace was captured at the core router connecting two large datacenters and 150 buildings to the Internet. We reconstruct packet histories for packets in this trace using topology and subnet information. Then we run a suite of 28 benchmark PHF queries which include filters on specific hosts, locations (datacenter, campus and Internet), paths between locations, and headers. Figure 7 shows the average PHF match time (on a single CPU core) for a representative set of queries on hosts, subnets (campus), and paths (dc_hdr–campus_hdr). Most matches execute fast (<300ns/match); the most expensive ones (900ns/-match) are complex BPF queries that contain a predictate on 24 subnets.

The above results show that even an unoptimized single-threaded implementation of PHF matching can achieve high throughput. In addition, PHF matching is embarrassingly parallel: each packet history can be matched against a PHF independent of all other packet histories, enabling linear multi-core scalability. A future optimized implementation can also perform the matching directly on compressed archives of packet histories, rather than on each individual packet history.

7.4 Provisioning Scenario

At the beginning of this paper, we suggested a set of questions, each of which maps to a filter in NetSight. With performance numbers for each piece of NetSight, we can estimate the traffic rate it can handle to answer those questions.

Adding up the end-to-end processing costs in NetSight – compressing, decompressing, assembling, and filtering packets – yields a per-core throughput of 240K postcards/second. With five hops on the typical path and 1000-byte packets, a single 16-core server, available for under $2000, can handle 6.1 Gb/s of network traffic. This is approximately the average rate of the transit traffic in our campus backbone. To handle the peak, a few additional servers would suffice, and as the network grows, the administrator can add servers knowing that NetSight will scale to handle the added demand.

The key takeaway is that NetSight is able to handle the load from an entire campus backbone with 20,000 users, with a small number of servers.

8 Scaling NetSight

If we do not compress postcards before sending them over the network, we need to send them each as a min-sized packet. We can calculate the bandwidth cost as a fraction of the data traffic as:

$$\text{cost} = \frac{\text{postcard packet size}}{\text{avg packet size}} \times \text{avg number of hops.}$$

The bandwidth cost is inversely proportional to the average packet size in the network.

For example, consider our university campus backbone with 14 internal routers connected by 10Gb/s links, two Internet-facing routers, a network diameter of 5 hops, and an average packet size of 1031 bytes. If we assume postcards are minimum-sized Ethernet packets, they increase traffic by $\frac{64B}{1031B} \times 5(hops) = 31\%$.[8]

The average aggregate utilization in our university backbone is about 5.9Gb/s, for which postcard traffic adds 1.8Gb/s. Adding together the peak traffic seen at each campus router, we get 25Gb/s of packet data, which will generate 7.8Gb/s of postcard traffic,

[8]If we overcome the min-size requirement by aggregating the 40 byte postcards into larger packets before sending them, the bandwidth overhead reduces to 19%.

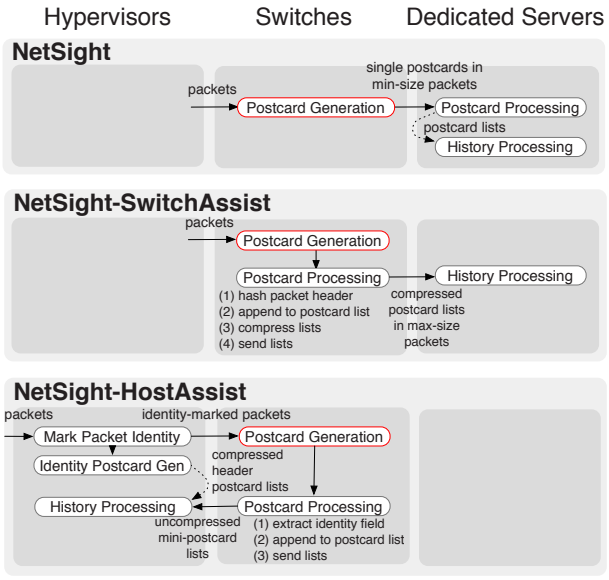


Figure 8: NetSight uses only dedicated servers, but adding switch processing (-SwitchAssist) and VM servers (-HostAssist) can reduce bandwidth costs and increase scalability. Postcard generation is common to all approaches.

yet can be handled by two NetSight servers (§7.4). If the postcards are sent in-band, this extra traffic may affect the network performance.

For a low-utilization network, especially test networks or production networks in the bring-up phase, these bandwidth costs may be acceptable for the debugging functionality NetSight provides. But for a live network with more hops, smaller packets, or higher utilization, our NetSight may consume too much network bandwidth. To scale NetSight to a large data center or an enterprise, we present two design modifications that reduce network bandwidth by moving some of the processing into the switches and end hosts, respectively.

NetSight-SwitchAssist, shown in the middle of Figure 8, uses additional logic in the switches to locally implement the Postcard Stage with compression, thus avoiding the extra network capacity needed to transport uncompressed postcards to the NetSight servers in minimum-size packets. Since switches send compressed aggregates of postcards to NetSight servers (rather than individual uncompressed postcards), the bandwidth requirement diminishes. For example, with a size of 15 bytes per compressed postcard (as shown in §7), the bandwidth requirement reduces from 31% to 7%.

NetSight-HostAssist, shown at the bottom of Figure 8, is suited for environments where end hosts can be modified. This design reduces postcard traffic by using a thin shim layer at each end host (e.g. in a software switch such as Open vSwitch [33]) to tag packets to help switches succinctly describe postcards. The shim tags each outgoing packet with a sequentially-incrementing globally-unique packet ID and locally stores the mapping between the ID and the packet header. When a switch receives a packet, it extracts the tag and generates a mini-postcard that contains only the packet ID, the flow table state and the input/output ports. This state is appended to a hash table entry keyed by the source address of this packet. Since a packet ID is valid only to a particular host, the shim can use fewer bytes (e.g. 4 bytes) to uniquely identify a packet. When enough bytes accumulate, the switch dispatches the hash entry (a list of packet IDs and state) to the source. At the end of a round, the hosts locally assemble and archive the packet history.

If on average, it takes 15 bytes per packet to store compressed headers at the VM hosts (§7), and 6 bytes per mini-postcard, the bandwidth overhead to collect postcards in the network reduces to 3%. This number contrasts with 31% overhead when postcards are collected naively. Since each end host stores packet histories for its own traffic, the mechanism scales with the number of hosts. If 3% is still unacceptable, then NetSight may be deployed for a subset of packets or a subset of switches. However, both of these options are qualitatively different; either NetSight cannot guarantee that a requested packet history will be available when ignoring some packets, or NetSight cannot guarantee that each generated packet history will represent a packet's complete path when not enabled network-wide.

To put things in perspective, while NetSight requires firmware modifications to expose existing hardware functionality, NetSight-SwitchAssist and NetSight-HostAssist require hardware modifications in the switches. If our campus network (§7.4) were to get upgraded to NetSight-SwitchAssist, one of the expensive compression steps would go away and yield a traffic processing rate of 7.3 Gb/s per server. Adding NetSight-HostAssist would yield a rate of 55 Gb/s per server, because mini-postcards require no compression or decompression. The processing costs are heavily dominated by compression, and reducing these costs seems like a worthwhile future direction to improve NetSight.

9 Related Work

Commercial tools for troubleshooting networks provide visibility through packet sampling [8, 34], configurable packet duplication [15, 18, 32], or log analysis [42]. Most lack the network-wide visibility and the packet-level state consistency provided by Net-

Sight. cPacket Networks has a commercial product that offers a central view with a grep-based interface, but it is unclear whether they support mechanisms to obtain network state that pertains to a specific packet's forwarding [10].

In the SDN space, OFRewind [45] records and plays back SDN control plane traffic; like NetSight, it also records flow table state via a proxy and logs packet traces. `ndb` [20] proposes the postcard-based approach to reconstruct the path taken by a packet. In this paper, we build upon those ideas with the packet history abstraction, the PHF API, four troubleshooting applications, and also describe and evaluate methods to scale the system. Other academic work, IP traceback, builds paths to locate denial-of-service attacks on the Internet [12, 38, 41]. Flow sampling monitors traffic and its distribution [14] or improves sampling efficiency and fairness [2, 13, 39]; NetSight has a different goal (network diagnosis) and uses different methods. Packet Obituaries [3] proposes an accountability framework to provide information about the fate of packets. Its notion of a "report" is similar to a packet history but provides only the inter-AS path information. *Each lacks a systematic framework to pose questions of these reports in a scalable way.*

Another class of related systems look for invariant violations atop a model of network behavior. These include data-plane configuration checkers like Anteater [29], Header Space Analysis [23, 22], and VeriFlow [25], as well as tools like NICE [6], which combines model checking and symbolic execution to verify network programs. These systems all *model* network behavior, but firmware and hardware bugs can creep in and break this idealized model. NetSight on the other hand, takes a more direct approach – it finds bugs that manifest themselves as errantly forwarded packets and provides direct evidence to help identify their root cause. Automatic Test Packet Generation [46] shares our overall approach, but uses a completely different method: sending test packets, as opposed to monitoring existing traffic. NetSight appears better suited for networks with rapidly changing state, because it avoids the expensive test packet set minimization step.

Virtual Network Diagnosis [44] shares surface similarities with NetSight, such as a distributed implementation and a query API; however, its focus is performance diagnosis for tenants, rather than connectivity debugging for the infrastructure owner. Gigascope [11] is a stream query processing system used to process large streams of packet data using an SQL-like query language. NetSight's query engine uses PHF, a regular-expression-like query language for fast processing of packet histories. X-Trace [16] is a tracing framework that helps in debugging general distributed systems by tracing tasks across different administrative domains. While similar in spirit, NetSight takes a different approach to address state-correlation and scalability challenges specific to tracing, storing, and querying packet histories.

10 Summary

Networks are inherently distributed, with highly concurrent data-plane and control-plane events, and they move data at aggregate rates greater than any single server can process. These factors make it challenging to pause or "single-step" a network, and none of our network diagnosis tools try to connect packet events to control events. As a result, administrators find it hard to construct a packet's perspective of its forwarding journey, despite the value for diagnosing problems.

NetSight addresses these challenges to improve network visibility in operational networks, by leveraging SDN to first gain visibility into forwarding events, and then tackling performance concerns with a scale-out system architecture, aggressive packet header compression, and carefully optimized C++ code. The surprising result is the feasibility and practicality of collecting and storing complete packet histories for all traffic on moderate-size networks.

Furthermore, NetSight demonstrates that given access to a network's complete packet histories, one can implement a number of compelling new applications. Atop the NetSight Packet History Filter (PHF) API, we implemented four applications—a network debugger, invariant monitor, packet logger, and hierarchical network profiler—none of which required more than 200 lines of code. These tools manifested their utility immediately, when a single, incompletely assembled packet history revealed a switch configuration error within minutes of our first test deployment.

Acknowledgments

We would like to thank our shepherd Ethan Katz-Bassett and the anonymous NSDI reviewers for their valuable feedback that helped significantly improve the quality of this paper. This work was funded by NSF FIA award CNS-1040190, NSF FIA award CNS-1040593-001, Stanford Open Networking Research Center (ONRC), a Hewlett Packard Fellowship, and a gift from Google.

References

[1] NetSight Source Code. `http://yuba.stanford.edu/netsight`.

[2] A. Arefin, A. Khurshid, M. Caesar, and K. Nahrstedt. Scaling Data-Plane Logging in Large Scale Networks. *MILCOM*, 2011.

[3] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing Packet Obituaries. *HotNets*, 2004.

[4] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. *IMC*, 2010.

[5] A JIT for packet filters. `http://lwn.net/Articles/437981/`.

[6] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. *NSDI*, 2012.

[7] Cisco: Configuring Local SPAN, RSPAN, and ERSPAN. `http://www.cisco.com/en/US/docs/switches/lan/catalyst6500/ios/12.2SX/configuration/guide/span.html`.

[8] B. Claise. RFC 5101: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. `http://tools.ietf.org/html/rfc5101`.

[9] R. Cox. Regular Expression Matching: the Virtual Machine Approach. `http://swtch.com/~rsc/regexp/regexp2.html`.

[10] cPacket Networks. Product overview. `http://cpacket.com/products/`.

[11] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. *SIGMOD*, 2003.

[12] D. Dean, M. Franklin, and A. Stubblefield. An Algebraic Approach to IP Traceback. *ACM Transactions on Information and System Security (TISSEC)*, 2002.

[13] N. Duffield. Fair Sampling Across Network Flow Measurements. *SIGMETRICS*, 2012.

[14] N. G. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM Transactions on Networking*, 2001.

[15] Endace Inc. `http://www.endace.com/`.

[16] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. *NSDI*, 2007.

[17] Intel Ethernet Switch FM5000/FM6000 Series. `http://www.intel.com/content/www/us/en/switch-silicon/ethernet-switch-fm5000-fm6000-series.html`.

[18] Gigamon. `http://www.gigamon.com/`.

[19] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, and N. McKeown. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, July 2008.

[20] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the Debugger for my Software-Defined Network? *HotSDN*, 2012.

[21] V. Jacobson. RFC 1144: Compressing TCP. `http://tools.ietf.org/html/rfc1144`.

[22] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. *NSDI*, 2013.

[23] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking For Networks. *NSDI*, 2012.

[24] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. *HotSDN*, 2012.

[25] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. Brighton Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. *NSDI*, 2013.

[26] M. Kobayashi, S. Seetharaman, G. Parulkar, G. Appenzeller, J. Little, J. van Reijendam, P. Weissmann, and N. McKeown. Maturing of OpenFlow and Software-Defined Networking through Deployments. *Computer Networks (Elsevier)*, 2013.

[27] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. *HotNets*, 2010.

[28] LBNL/ICSI Enterprise Tracing Project. `http://www.icir.org/enterprise-tracing/download.html`.

[29] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. *SIGCOMM*, 2011.

[30] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. *USENIX Winter Conference*, 1993.

[31] NEC IP8800 OpenFlow-enabled Switch. `http://support.necam.com/pflow/legacy/ip8800/`.

[32] Net optics: Architecting visibility into your network. `http://www.netoptics.com/`.

[33] Open vSwitch: An Open Virtual Switch. `http://openvswitch.org/`.

[34] P. Phaal. sFlow Version 5. `http://sflow.org/sflow_version_5.txt`.

[35] The POX Controller. `http://github.com/noxrepo/pox`.

[36] RipL-POX (Ripcord-Lite for POX): A simple network controller for OpenFlow-based data centers. `https://github.com/brandonheller/riplpox`.

[37] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 1992.

[38] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. *SIGCOMM*, 2000.

[39] V. Sekar, M. Reiter, W. Willinger, H. Zhang, R. Kompella, and D. Andersen. CSAMP: A System for Network-Wide Flow Monitoring. *NSDI*, 2008.

[40] S. Shenker. The Future of Networking, and the Past of Protocols. *Open Networking Summit*, 2011.

[41] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP traceback. *SIGCOMM*, 2001.

[42] Splunk: Operational Intelligence, Log Management, Application Management, Enterprise Security and Compliance. `http://splunk.com`.

[43] The CAIDA UCSD Anonymized Internet Traces 2012 – Nov 15 2012. `http://www.caida.org/data/passive/passive_2012_dataset.xml`.

[44] W. Wu, G. Wang, A. Akella, and A. Shaikh. Virtual Network Diagnosis as a Service. *SoCC*, 2013.

[45] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. *USENIX ATC*, 2011.

[46] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. *CoNEXT*, 2012.

Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks

Hongyi Zeng[†*], Shidong Zhang[§], Fei Ye[§], Vimalkumar Jeyakumar[†*]
Mickey Ju[§], Junda Liu[§], Nick McKeown[†], Amin Vahdat[§‡]
[†]*Stanford University* [§]*Google* [‡]*UCSD*

Abstract

Data center networks often have errors in the forwarding tables, causing packets to loop indefinitely, fall into black-holes or simply get dropped before they reach the correct destination. Finding forwarding errors is possible using static analysis, but none of the existing tools scale to a large data center network with thousands of switches and millions of forwarding entries. Worse still, in a large data center network the forwarding state is constantly in flux, which makes it hard to take an accurate snapshot of the state for static analysis.

We solve these problems with Libra, a new tool for verifying forwarding tables in very large networks. Libra runs fast because it can exploit the scaling properties of MapReduce. We show how Libra can take an accurate snapshot of the forwarding state 99.9% of the time, and knows when the snapshot cannot be trusted. We show results for Libra analyzing a 10,000 switch network in less than a minute, using 50 servers.

1 Introduction

Data center networks are immense. Modern data centers can employ 10,000 switches or more, each with its own forwarding table. In such a large network, failures are frequent: links go down, switches reboot, and routers may hold incorrect prefix entries. Whether routing entries are written by a distributed routing protocol (such as OSPF) or by a remote route server, the routing state is so large and complex that errors are inevitable. We have seen logs from a production data center reporting many thousands of routing changes per day, creating substantial opportunity for error.

Data centers withstand failures using the principles of scale-out and redundant design. However, the underlying assumption is that the system reacts correctly to failures. Dormant bugs in the routing system triggered by rare boundary conditions are particularly difficult to find. Common routing failures include routing loops and black-holes (where traffic to one part of the network disappears). Some errors only become visible when an otherwise benign change is made. For example, when a routing prefix is removed it can suddenly expose a mistake with a less specific prefix.

Routing errors should be caught quickly before too much traffic is lost or security is breached. We therefore need a fast and scalable approach to verify correctness of the entire forwarding state. A number of tools have been proposed for analyzing networks including HSA [10], Anteater [13], NetPlumber [9] and Veriflow [11]. These systems take a snapshot of forwarding tables, then analyze them for errors. We first tried to adopt these tools for our purposes, but ran into two problems. First, they assume the snapshot is consistent. In large networks with frequent changes to routing state, the snapshot might be inconsistent because the network state changes while the snapshot is being taken. Second, none of the tools are sufficiently fast to meet the performance requirements of modern data center networks. For example, Anteater [13] takes more than 5 minutes to check for loops in a 178-router topology.

Hence, we set out to create Libra, a fast, scalable tool to quickly detect loops, black-holes, and other reachability failures in networks with tens of thousands of switches. Libra is much faster than any previous system for verifying forwarding correctness in a large-scale network. Our benchmark goal is to verify all forwarding entries in a 10,000 switch network with millions of rules in minutes.

We make two main contributions in Libra. First, Libra capture stable and consistent snapshots across large network deployments, using the event stream from routing processes (Section 3). Second, in contrast to prior tools that deal with arbitrarily structured forwarding tables, we substantially improve scalability by assuming packet forwarding based on longest prefix matching.

[*]Hongyi Zeng and Vimalkumar Jeyakumar were interns at Google when this work was done. Hongyi Zeng is currently with Facebook.

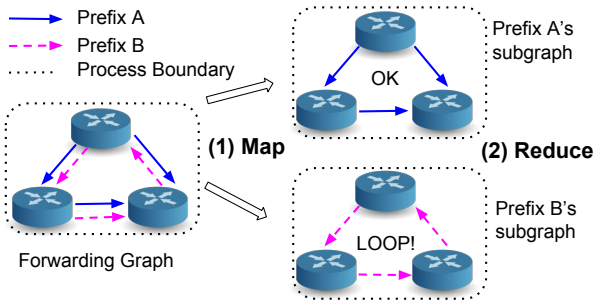


Figure 1: Libra divides the network into multiple forwarding graphs in mapping phase, and checks graph properties in reducing phase.

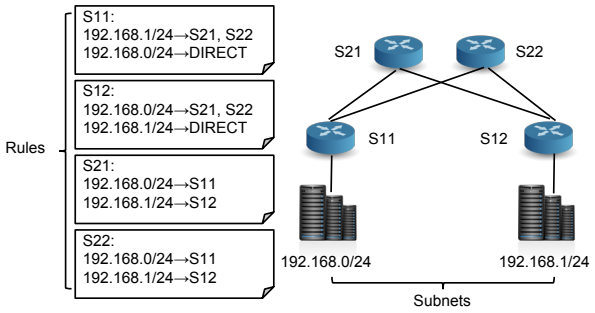


Figure 2: Small network example for describing the types of forwarding error found by Libra.

Libra uses MapReduce for verification. It starts with the full graph of switches, each with its own prefix table. As depicted in Figure 1, Libra completes verification in two phases. In the *map* phase, it breaks the graph into a number of slices, one for each prefix. The slice consists of only those forwarding rules used to route packets to the destination. In the *reduce* phase, Libra indepedently analyzes each slice, represented as a forwarding graph, *in parallel* for routing failures.

We evaluate Libra on the forwarding tables from three different networks. First, "DCN" is an emulated data center network with 2 million rules and 10,000 switches. Second, "DCN-G" is made from 100 replicas of DCN connected together; i.e., 1 million switches. Third, "INET" is a network with 300 IPv4 routers each contains the full BGP table with half a million rules. The results are encouraging. Libra takes one minute to check for loops and black-holes in DCN, 15 minutes for DCN-G and 1.5 minutes for INET.

2 Forwarding Errors

A small toy network can illustrate three common types of error found in forwarding tables. In the two-level tree network in Figure 2 two top-of-rack (ToR) switches (S11, S12) are connected to two spine switches (S21, S22). The downlinks from S11 and S12 connect to up to 254 servers on the same /24 subnet. The figure shows a "correct" set of forwarding tables. Note that our example network uses multipath routing. Packets arriving at S12 on the right and destined to subnet 192.168.0/24 on the left are load-balanced over switches S21 and S22. Our toy network has 8 rules, and 2 subnets.

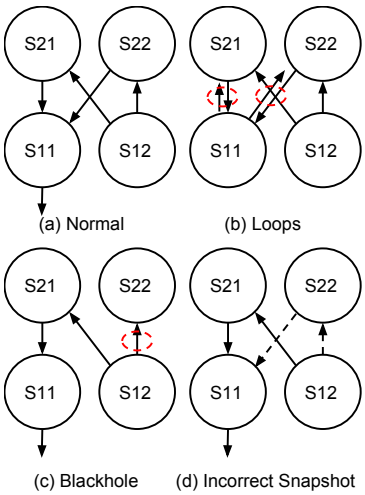


Figure 3: Forwarding graphs for 192.168.0/24 as in Figure 2, and potential abnormalities.

A *forwarding graph* is a directed graph that defines the network behavior for each subnet. It contains a list of (local_switch, remote_switch) pairs. For example, in Figure 3(a), an arrow from S12 to S21 means the packets of subnet 192.168.0/24 can be forwarded from S12 to S21. Multipath routing can be represented by a node that has more than one outgoing edge. Figure 3(b)-(d) illustrates three types of forwarding error in our simple network, depicted in forwarding graphs.

Loops: Figure 3(b) shows how an error in S11's forwarding tables causes a *loop*. Instead of forwarding 192.168.0/24 down to the servers, S11 forwards packets up, i.e., to S21 and S22. S11's forwarding table is now:

`192.168.0/24` → `S21, S22`
`192.168.1/24` → `S21, S22`

The network has two loops: S21-S11-S21 and S22-S11-S22, and packets addressed to 192.168.0/24 will never reach their destination.

Black-holes: Figure 3(c) shows what happens if S22 loses one of its forwarding entries: `192.168.0/24` → `S11`. In this case, if S12 spreads packets destined to 192.168.0/24 over both S21 and S22, packets arriving to S22 will be dropped.

Incorrect Snapshot: Figure 3(d) shows a subtle problem that can lead to *false positives* when verifying forwarding tables. Suppose the link between S11-S22 goes down. Two events take place (shown as dashed arrows

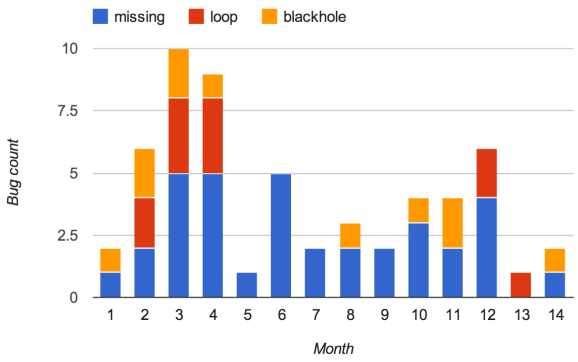


Figure 4: Routing related tickets by month and type.

in the figure): $e1$: S22 deletes `192.168.0/24` $\rightarrow$ `S11`, and $e2$: S12 stops forwarding packets to S22. Because of the asynchronous nature of routing updates, the two events could take place in either order $(e1, e2)$ or $(e2, e1)$. A snapshot may capture one event, but not the other, or might detect them happening in the reverse order.

The sequence $(e1, e2)$ creates a temporary blackhole as in Figure 3(c), whereas the desired sequence $(e2, e1)$ does not. To avoid raising an unnecessary alarm (by detecting $(e1, e2)$ even though it did not happen), or missing an error altogether (by incorrectly assuming that $(e2, e1)$ happened), Libra must detect the correct state of the network.

2.1 Real-world Failure Examples

To understand how often forwarding errors take place, we examined a log of "bug tickets" from 14 months of operation in a large Google data center. Figure 4 categorizes 35 tickets for missing forwarding entries, 11 for loops, and 11 for black-holes. On average, four issues are reported per month.

Today, forwarding errors are tracked down by hand which - given the size of the network and the number of entries - often takes many hours. And because the diagnosis is done after the error occurred, the sequence of events causing the error has usually long-since disappeared before the diagnosis starts. This makes it hard to reproduce the error.

Case 1: Detecting Loops. One type of loop is caused by prefix aggregation. Prefixes are aggregated to compact the forwarding tables: a cluster E can advertise a *single* prefix to reach all of the servers connected "below" it to the core C, which usually includes the addresses of servers that have not yet been deployed. However, packets destined to these non-deployed addresses (e.g., due to machine maintenance) can get stuck in loops. This is because C believes these packets are destined to E, while E lacks the forwarding rules to digest these packets due to the incomplete deployment, instead, E's default rules lead packets back to C.

This failure does not cause a service to fail (because the service will use other servers instead), but it does degrade the network causing unnecessary congestion. In the past, these errors were ignored because of the prohibitive cost of performing a full cluster check. Libra can finish checking in less a minute, and identify and report the specific switch and prefix entry that are at risk.

Case 2: Discovering Black-holes. In one incident, traffic was interrupted to hundreds of servers. Initial investigation showed that some prefixes had high packet loss rate, but packets seemed to be discarded randomly. It took several days to finally uncover the root cause: A subset of routing information was lost during BGP updates between domains, likely due to a bug in the routing software, leading to black-holes.

Libra will detect missing forwarding entries quickly, reducing the outage time. Libra's stable snapshots also allow it to disambiguate temporary states during updates from long-term back-holes.

Case 3: Identifying Inconsistencies. Network control runs across several instances, which may fail from time to time. When a secondary becomes the primary, it results in a flurry of changes to the forwarding tables. The changes may temporarily or permanently conflict with the previous forwarding state, particularly if the changeover itself fails before completing. The network can be left in an inconsistent state, leading to packet loss, black-holes and loops.

2.2 Lessons Learned

Simple things go wrong: Routing errors occur even in networks using relatively simple IP forwarding. They also occur due to firmware upgrades, controller failure and software bugs. It is essential to check the forwarding state *independently*, outside the control software.

Multiple moving parts: The network consists of multiple interacting subsystems. For example, in case 1 above, Intra-DC routing is handled locally, but routing is a global property. This can create loops that are hard to detect locally within a subsystem. There are also multiple network controllers. Inconsistent state makes it hard for the control plane to detect failures on its own.

Scale matters: Large data center networks use multipath routing, which means there are many forwarding paths to check. As the number of switches, N, grows the number of paths and prefix tables grow, and the complexity of checking all routes grows with N^2. It is essential for a static checker to scale linearly with the network.

3 Stable Snapshots

It is not easy to take an accurate snapshot of the forwarding state of a large, constantly changing network. But if Libra runs its static checks on a snapshot of the state that never actually occurred, it will raise false alarms and miss real errors. We therefore need to capture - and check - a snapshot of the global forwarding state that actually existed at one instant in time. We call these *stable snapshots*.[1]

When is the state stable? A large network is usually controlled by multiple *routing processes*,[2] each responsible for one or more switches. Each process sends timestamped updates, which we call *routing events*, to add, modify and delete forwarding entries in the switches it is responsible for. Libra monitors the stream of routing events to learn the global network state.

Finding the stable state of a *single* switch is easy: each table is only written by one routing process using a single clock, and all events are processed in order. Hence, Libra can reconstruct a stable state simply by replaying events in timestamp order.

By contrast, it is not obvious how to take a *globally* stable snapshot of the state when different routing processes update their switches using different, unsynchronized clocks. Because the clocks are different, and events may be delayed in the network, simply replaying the events in timestamp order can result in a state that did not actually occur in practice, leading to false positives or missed errors (Section 2).

However, even if we can not precisely synchronize clocks, we can *bound* the difference between any pair of clocks with high confidence using NTP [15]. And we can bound how out-of-date an event packet is, by prioritizing event packets in the network. Thus, every timestamp t can be treated as lying in an interval $(t-\varepsilon, t+\varepsilon)$, where ε bounds the uncertainty of when the event took place.[3] The interval represents the notion that network state changes atomically at some unknown time instant within the interval.

Figure 5 shows an example of finding a stable snapshot instant. It is easy to see that if no routing events are recorded during a 2ε period we can be confident that no routing changes actually took place. Therefore, the snapshot of the current state is stable (i.e., accurate).[4]

The order of any two past events from *different* processes is irrelevant to the current state, since they are applied to different tables without interfering with each other (recall that each table is controlled by only one process). So Libra only needs to replay all events in timestamp order (to ensure events for the *same* table are played in order) to accurately reconstruct the current state.

This observation suggests a simple way to create a stable snapshot by simply waiting for a quiet 2ε period with no routing update events.

Feasibility: The scheme only works if there are frequent windows of size 2ε in which no routing events take place. Luckily, we found that these quiet periods happen frequently: we analyzed a day of logs from all routing processes in a large Google data center with a few thousand switches. Figure 6 shows the CDF of the inter-arrival times for the 28,445 routing events reported by the routing processes during the day. The first thing to notice is the burstiness — over 95% of events occur within 400ms of another event, which means there are long periods when the state is stable. Table 1 shows the fraction of time the network is stable, for different values of ε. As expected, larger ε leads to fewer stable states and smaller percentage of stable time. For example, when ε=100ms, only 2,137 out of all 28,445 states are stable. However,

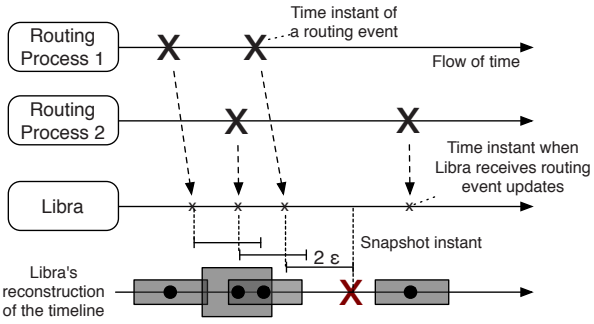


Figure 5: Libra's reconstruction of the timeline of routing events, taking into account bounded timestamp uncertainty ε. Libra waits for twice the uncertainty to ensure there are no outstanding events, which is sufficient to deduce that routing has stabilized.

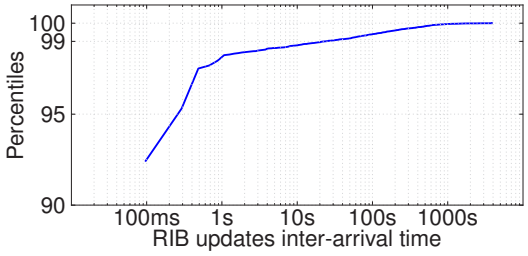


Figure 6: CDF of inter-arrival times of routing events from a large production data center. Routing events are very bursty: over 95% of events happen within 400ms of another event.

[1] Note that a stable snapshot is not the same as a *consistent* snapshot [3], which is only one *possible state* of a distributed system that might not actually have occurred in practice.

[2] Libra only considers processes that can directly modify tables. While multiple high-level protocols can co-exist (e.g., OSPF and BGP), there is usually one common low-level table manipulation API.

[3] The positive and negative uncertainties can be different, but here we assume they are the same for simplicity.

[4] A formal proof can be found in [14, § 3.3].

ε/ms	# of stable states	time in stable state/%
0	28,445	100.00
1	16,957	99.97
100	2,137	99.90
1,000	456	99.75
10,000	298	99.60

Table 1: As the uncertainty in routing event timestamps (ε) increases, the number of stable states decreases. However, since routing events are bursty, the state is stable most of the time.

because the event stream is so bursty, the unstable states are extremely short-lived, occupying *in total* only 0.1% ($\sim$1.5min) of the entire day. Put another way, for 99.9% of the time, snapshots are stable and the static analysis result is trustworthy.

Taking stable snapshots: The stable snapshot instant provides a reference point to reconstruct the global state. Libra's stable snapshot process works as follows:

1) Take an initial snapshot S_0 as the combination of all switches' forwarding tables. At this stage, each table can be recorded at a slightly different time.

2) Subscribe to timestamped event streams from all routing processes, and apply each event e_i, in the order of their timestamps, to update the state from S_{i-1} to S_i.

3) After applying e_j, if no event is received for 2ε time, declare the current snapshot S_j stable. In other words, S_0 and all past events e_i form a stable state that actually existed at this time instant.

4 Divide and Conquer

After Libra has taken a stable snapshot of the forwarding state, it sets out to statically check its correctness. Given our goal of checking networks with over 10,000 switches and millions of forwarding rules, we will need to break down the task into smaller, parallel computations. There are two natural ways to consider partitioning the problem:

Partition based on switches: Each server could hold the forwarding state for a cluster of switches, partitioning the network into a number of clusters. We found this approach does not scale well because checking a forwarding rule means checking the rules in many (or all) partitions - the computation is quickly bogged down by communication between servers. Also, it is hard to balance the computation among servers because some switches have very different numbers of forwarding rules (e.g. spine and leaf switches).

Partition based on subnets: Each server could hold the forwarding state to reach a set of subnets. The server computes the forwarding graph to reach each subnet, then checks the graph for abnormalities. The difficulty with this approach is that each server must hold the entire set of forwarding tables in memory, and any update to the forwarding rules affects all servers.

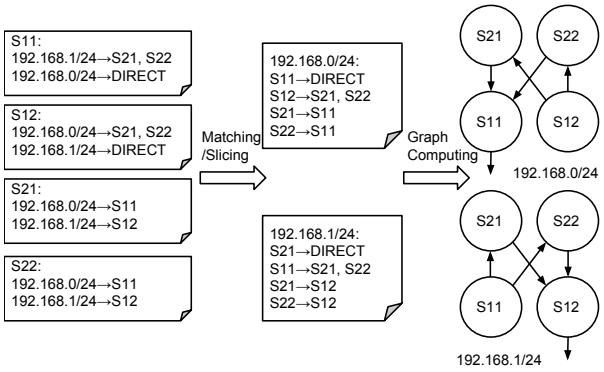


Figure 7: Steps to check the routing correctness in Figure 2.

Libra partitions the network based on subnets, for reasons that will become clear. We observe that the route checker's task can be divided into two steps. First, Libra *associates* forwarding rules with subnets, by finding the set of forwarding rules relevant to a subnet (i.e., they are associated if the subnet is included in the rule's prefix). Second, Libra builds a forwarding graph to reach each subnet, by assembling all forwarding rules for the subnet. Both steps are embarrassingly parallel: matching is done per (subnet, forwarding rule) pair; and each subnet's forwarding graph can be analyzed independently.

Libra therefore proceeds in three steps using N servers:

Step 1 - Matching: Each server is initialized with the *entire* list of subnets, and each server is assigned $1/N$ of all forwarding rules. The server considers each forwarding rule in turn to see if it belongs to the forwarding graph to a subnet (i.e. the forwarding rule is a prefix of the subnet).[5] If there is a match, the server outputs the (subnet, rule) pair. Note that a rule may match more than one subnet.

Step 2 - Slicing: The (subnet, rule) pairs are grouped by subnet. We call each group a *slice*, because it contains all the rules and switches related to this subnet.

Step 3 - Graph Computing: The slices are distributed to N servers. Each server constructs a forwarding graph based on the rules contained in the slice. Standard graph algorithms are used to detect network abnormalities, such as loops and black-holes.

Figure 7 shows the steps to check the network in Figure 2. After the slicing stage, the forwarding rules are organized into two slices, corresponding to the two subnets 192.168.0/24 and 192.168.1/24. The forwarding graph for each slice is calculated and checked in parallel.

[5]Otherwise, a subnet will be fragmented by a more specific rule, leading to a complex forwarding graph. See the last paragraph in Section 9 for detailed discussion.

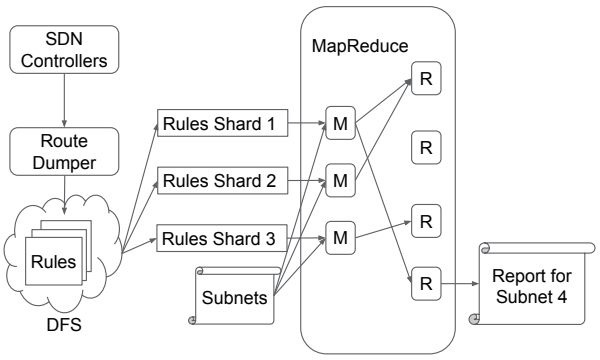


Figure 8: Libra workflow.

If a routing error occurs and the second rule in S11 becomes `192.168.0/24` $\rightarrow$ `S21, S22`, the loop will show up in the forwarding graph for 192.168.0/24. S11 will point back to S21 and S22, which will be caught in graph loop detection algorithm.

Our three-step process is easily mapped to MapReduce, which we describe in the next section.

5 Libra

Libra consists of two main components: a *route dumper* and a MapReduce-based *route checker*. Figure 8 shows Libra's workflow.

The route dumper takes stable snapshots from switches or controllers, and stores them in a distributed file system. Next, the snapshot is processed by a MapReduce-based checker.

A quick review of MapReduce: MapReduce [5] divides computation into two phases: *mapping* and *reducing*. In the mapping phase, the input is partitioned into small "shards". Each of them is processed by a mapper in parallel. The mapper reads in the shard line by line and outputs a list of `<key, value>` pairs. After the mapping phase, the MapReduce system *shuffles* outputs from different mappers by sorting by the key. After shuffling, each reducer receives a `<key, values>` pair, where `values=[value1, value2, ...]` is a list of all values corresponding to the `key`. The reducer processes this list and outputs the final result. The MapReduce system also handles checkpointing and failure recovery.

In Libra, the set of forwarding rules is partitioned into small shards and delivered to mappers. Each mapper also takes a full set of subnets to check, which by default contains all subnets in the cluster, but alternatively can be subsets selected by user. Mappers generate intermediate keys and values, which are shuffled by MapReduce. The reducers compile the values that belong to the same subnet and generate final reports.

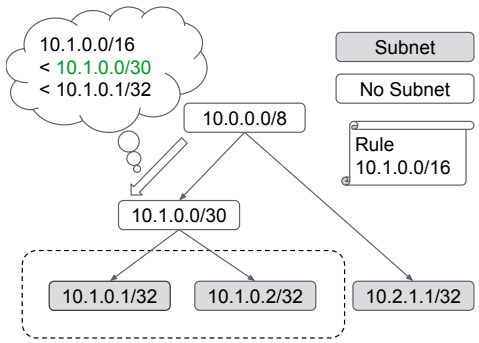


Figure 9: Find all matching subnets in the trie. `10.1.0.0/30` (*X*) is the smallest matching trie node bigger than the rule `10.1.0.0/16` (*A*). Hence, its children with subnets `10.1.0.1/32` and `10.1.0.2/32` match the rule.

5.1 Mapper

Mappers are responsible for slicing networks by subnet. Each mapper reads one forwarding rule at a time. If a subnet matches the rule, the mapper outputs the subnet prefix as the intermediate key, along with the value `<rule_mask_len, local_switch, remote_switches, priority>`. The following is an example (`local_switch`, `remote_switches`, `priority` is omitted):

```
Subnets:   192.168.1.1/32
           192.168.1.2/32
Rules:     192.168.1.0/28
           192.168.0.0/16
Outputs:   <192.168.1.1/32, 28>
           <192.168.1.1/32, 16>
           <192.168.1.2/32, 16>
```

Since each mapper only sees a portion of the forwarding rules, there may be a longer and more specific—but unseen—matching prefix for the subnet in the same forwarding table. We *defer* finding the longest matching to the reducers, which see all matching rules.

Mappers are first initialized with a full list of subnets, which are stored in an in-memory binary trie for fast prefix matching. After initialization, each mapper takes a shard of the routing table, and matches the rules against the subnet trie. This process is different from the conventional longest prefix matching: First, in conventional packet matching, rules are placed in a trie and packets are matched one by one. In Libra, we build the trie with *subnets*. Second, the goal is different. In conventional packet matching, one looks for the longest matching rule. Here, mappers simply output *all* matching subnets in the trie. Here, matching has the same meaning—the subnet's prefix must fully fall within the rule's prefix.

We use a trie to efficiently find "all matching prefixes," by searching for the *smallest matching trie node* (called node *X*) that is *bigger or equal to* the rule prefix (called

node A). Here, "small" and "big" refer to the lexicographical order (not address space size), where for each bit in an IP address, *wildcard* $< 0 < 1$. X may or may not contain a subnet. If X exists, we enumerate all its non-empty decedents (including X itself). Otherwise, we declare that there exist no matching subnets in the trie. Figure 9 shows an example. `10.1.0.0/30` (X) is the smallest matching trie node bigger than the rule `10.1.0.0/16` (A). Hence, its children with subnets `10.1.0.1/32` and `10.1.0.2/32` match the rule.

Proof: We briefly prove why this algorithm is correct. In an uncompressed trie, each bit in the IP address is represented by one level, and so the algorithm is correct by definition: if there exist matching subnets in the trie, A must exist in the trie and its descendants contain all matching prefixes, which means $A = X$.

In a compressed trie, nearby nodes may be combined. A may or may not exist in the trie. If it exists, the problem reduces to the uncompressed trie scenario. If A does not exist in the trie, X (if it exists) contains all matching subnets in its descendants. This is because:

a) Any node Y smaller than X does not match A. Because there is no node bigger than A and smaller than X (otherwise X is not the smallest matching node), $Y < X$ also means $Y < A$. As a result, Y cannot fall within A's range. This is because for Y to fall within A, all A's non-wildcard bits should appear in Y, which implies $Y \geq A$.

b) Any node Y bigger than the biggest descendants of X does not match A. Otherwise, X and Y must have a common ancestor Z, where Z matches A because both X and Y match A, and $Z < X$ because Z is the ancestor of X (a node is always smaller than its descendants). This contradicts the assumption that X is smallest matching node of A.

Time complexity: We can break down the time consumed by the mapping phase into two parts. The time to construct the subnet trie is $O(T)$, where T is the number of subnets, because inserting an IP prefix into a trie takes constant time ($\leq$ length of IP address). If we consider a single thread, it takes $O(R)$ time to match R rules against the trie. So the total time complexity is $O(R+T)$. If N mappers share the same trie, we can reduce the time to $O(R+\frac{T}{N})$. Here, we assume $R \gg T$. If $T \gg R$, one may want to construct a trie with rules rather than subnets (as in conventional longest-prefix-matching).

5.2 Reducer

The outputs from the mapping phase are shuffled by intermediate keys, which are the subnets. When shuffling finishes, a reducer will receive a subnet, along with an unordered set of values, each containing `rule_mask_len`, `local_switch`, `remote_switches`, and `priority`. The reducer first selects the highest priority rule per `local_switch`: For the same `local_switch`, the rule with higher `priority` is selected; if two rules have the same priority, the one with larger `mask_len` is chosen. The reducer then constructs a directed forwarding graph using the selected rules. Once the graph is constructed, the reducer uses graph library to verify the properties of the graph, for example, to check if the graph is loop-free.

Time complexity: In most networks we have seen, a subnet matches at most 2-4 rules in the routing table. Hence, selecting the highest priority rule and constructing the graph takes $O(E)$ time, where E is the number of physical links in the network. However, the total runtime depends on the verification task, as we will discuss in Section 6.

5.3 Incremental Updates

Until now, we have assumed Libra checks the forwarding correctness from scratch each time it runs. Libra also supports incremental updates of subnets and forwarding rules, allowing it to be used as an independent "correctness checking service" similar to NetPlumber [9] and Veriflow [11]. In this way, Libra could be used to check forwarding rules quickly, *before* they are added to the forwarding tables in the switches. Here, a in-memory, "streaming" MapReduce runtime (such as [4]) is needed to speed up the event processing.

Subnet updates. Each time we add a subnet for verification, we need to rerun the whole MapReduce pipeline. The mappers takes $O(\frac{R}{N})$ time to find the relevant rules. And a single reducer takes $O(E)$ time to construct the directed graph slice for the new subnet. If one has several subnets to add, it is faster to run them in a batch, which takes $O(T+\frac{R}{N})$ instead of $O(\frac{RT}{N})$ to map.

Removing subnets is trivial. All results related to the subnets are simply discarded.

Forwarding rule updates. Figure 10 shows the workflow to add new forwarding rules. To support incremental updates of rules, reducers need to store the forwarding graph for each slice it is responsible for. The reducer could keep the graph in memory or disk—the trade-off is a larger memory footprint.[6] If the graphs are in disk, a fixed number of idle reducer processes live in the memory and fetch graphs upon request. Similarly, the mappers need to keep the subnet trie.

To add a rule, a mapper is spawned just as it sees another line of input (Step 1). Matching subnets from the trie are shuffled to multiple reducers (Step 2). Each reducer reads the previous slice graph (Step 3), and recalculates it with the new rule (Step 4).

Deleting a rule is similar. The mapper tags the rule as "to be deleted" and pass it to reducers for updating the

[6] At any time instance, only a small fraction of graphs will be updated, and so keeping all states in-memory can be quite inefficient.

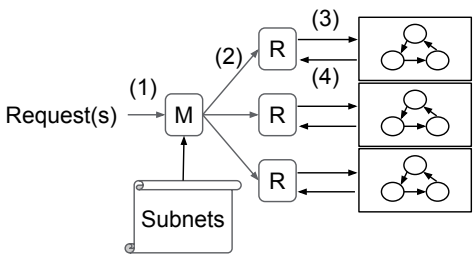


Figure 10: Incremental rule updates in Libra. Mappers dispatch matching <subnet, rule> pair to reducers, indexed by subnet. Reducers update the forwarding graph and recompute graph properties.

slice graph. However, in the graph's adjacency list, the reducer not only needs to store the highest priority rule, but also *all* matching rules. This is because if a highest priority rule is deleted, the reducer must use the second highest priority rule to update the graph.

Besides updating graphs, in certain cases, graph properties can also be checked incrementally, since the update only affects a small part of graph. For example, in loop-detection, adding an edge only requires a Depth-First-Search (DFS) starting from the new edge's destination node, which normally will not traverse the entire graph.

Unlike NetPlumber and Veriflow, Libra does not need to explicitly remember the dependency between rules. This is because the dependency is already encoded in the matching and shuffling phases.

5.4 Route Dumper

The route dumper records each rule using five fields: `<switch, ip, mask_len, nexthops, priority>`. `switch` is the unique ID of the switch; `ip` and `mask_len` is the prefix. `nexthops` is a list of port names because of multipath. `priority` is an integer field serving as a tie-breaker in longest prefix matching. By storing the egress ports in `nexthops`, Libra encodes the topology information in the forwarding table.

Although the forwarding table format is mostly straightforward, two cases need special handling:

Ingress port dependent rules. Some forwarding rules depend on particular ingress ports. For example, a rule may only be in effect for packets entering the switch from port `xe1/1`. In reducers we want to construct a simple directed graph that can be represented by an adjacency list. Passing this ingress port dependency to the route checker will complicate the reducer design, since the next hop in the graph depends not only on the current hop, but also the *previous hop*.

We use the notion of *logical switches* to solve this problem. First, if a switch has rules that depend on the ingress port, we split the switch into multiple logical switches. Each logical switch is given a new name and contains the rules depending on one ingress port, so that the port is "owned" by the new logical switch. We copy rules from the original switch to the logical switch. Second, we update the rules in upstream switches to forward to the logical switch.

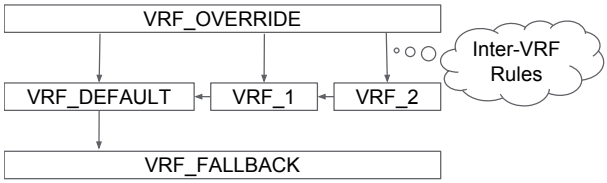


Figure 11: Virtual Routing and Forwarding (VRFs) are multiple tables within the same physical switch. The tables have dependency (inter-VRF rules) between them.

Multiple tables. Modern switches can have multiple forwarding tables that are chained together by arbitrary matching rules, usually called "Virtual Routing and Forwarding" (VRF). Figure 11 depicts an example VRF set up: incoming packets are matched against VRF_OVERRIDE. If no rule is matched, they enter VRF_1 to VRF_16 according to some "triggering" rules. If all matching fails, the packet enters VRF_DEFAULT.

The route dumper maps multiple tables in a physical switch into multiple logical switches, each containing one forwarding table. Each logical switch connects to other logical switches directly. The rules chaining these VRFs are added as lowest priority rules in the logical switch's table. Hence, if no rule is matched, the packet will continue to the next logical switch in the chain.

6 Use cases

In Libra, the directed graph constructed by the reducer contains *all* data plane information for a particular subnet. In this graph, each vertex corresponds to a forwarding table the subnet matched, and each edge represents a possible link the packet can traverse. This graph also encodes multipath information. Therefore, routing correctness directly corresponds to graph properties.

Reachability: A reachability check ensures the subnet can be reached from any switch in the network. This property can be verified by doing a (reverse) DFS from the subnet switch, and checking if the resulting vertex set contains all switches in the network. The verification takes $O(V+E)$ time where V is the number of switches and E the number of links.

Loop detection: A loop in the graph is equivalent to at least one *strongly connected component* in the directed graph. Two vertices `s1` and `s2` belong to a strongly connected component, if there is a path from `s1` to `s2` and a path from `s2` to `s1`. We find strongly connected components using Tarjan's Algorithm [21] whose time com-

plexity is $O(V+E)$.

Black-holes: A switch is a black-hole for a subnet if the switch does not have a matching route entry for the subnet. Some black-holes are legitimate: if the switch is the last hop for the subnet, or there is an explicit drop rule. Implicit drop rules need to be checked if that is by design. Black-holes map to vertices with zero out-degree, which can therefore be enumerated in $O(V)$ time.

Waypoint routing: Network operators may require traffic destined to certain subnets to go through a "waypoint," such as a firewall or a middlebox. Such behavior can be verified in the forwarding graph by checking if the waypoint exists on all the forwarding paths. Specifically, one can remove the waypoint and the associated links, and verify that no edge switches appear any more in a DFS originated from the subnet's first hop switch, with the runtime complexity of $O(V+E)$.

7 Implementation

We have implemented Libra for checking the correctness of Software-Defined Network (SDN) clusters. Each cluster is divided into several domains where each domain is controlled by a controller. Controllers exchange routing information and build the routing tables for each switch.

Our Libra prototype has two software components. The route dumper, implemented in Python, connects to each controller and downloads routing events, forwarding tables and VRF configurations in Protocol Buffers [17] format in parallel. It also consults the topology database to identify the peer of each switch link. Once the routing information is downloaded, we preprocess the data as described in Section 5.4 and store it in a distributed file system.

The route checker is implemented in C++ as a MapReduce application in about 500 lines of code. We use a Trie library for storing subnets, and use Boost Graph Library [1] for all graph computation. The same binary can run at different levels of parallelism—on a single machine with multiple processes, or on a cluster with multiple machines, simply by changing command line flags.

Although Libra's design supports incremental updates, our current prototype only does batch processing. We use micro-benchmarks to evaluate the specific costs for incremental processing in Section 8.5, on a simplified prototype with one mapper and one reducer.

8 Evaluation

To evaluate Libra's performance, we first measure start-to-finish runtime on a single machine with multi-threading, as well as on multiple machines in a cluster. We also demonstrate Libra's linear scalability as well as its incremental update capability.

Data set	Switches	Rules	Subnets
DCN	11,260	2,657,422	11,136
DCN-G	1,126,001	265,742,626	1,113,600
INET	316	151,649,486	482,966

Table 2: Data sets used for evaluating Libra.

8.1 Data Sets

We use three data sets to evaluate the performance of Libra. The detailed statistics are shown in Table 2.

DCN: DCN is an SDN testbed used to evaluate the scalability of the controller software. Switches are emulated by OpenFlow agents running on commodity machines and connected together through a virtual network fabric. The network is partitioned among controllers, which exchange routing information to compute the forwarding state for switches in their partition. DCN contains about 10 thousand switches and 2.6 million IPv4 forwarding rules. VRF is used throughout the network.

DCN-G: To stress test Libra, we replicate DCN 100 times by shifting the address space in DCN such that each DCN-part has a unique IP address space. A single top-level switch interconnects all the DCN pieces together. DCN-G has 1 million switches and 265 million forwarding rules.

INET: INET is a synthetic wide area backbone network. First, we use the Sprint network topology discovered by RocketFuel project [20], which contains roughly 300 routers. Then, we create an interface for each prefix found in a full BGP table from Route Views [18] ($\sim$500k entries as of July 2013), and spread them randomly and uniformly to each router as "local prefixes." Finally, we compute forwarding tables using shortest path routing.

8.2 Single Machine Performance

We start our evaluation of Libra by running loop detection locally on a desktop with Intel 6-core CPU and 32GB memory. Table 3 summarizes the results. We have learned several aspects of Libra from single machine evaluation:

I/O bottlenecks: Standard MapReduce is disk-based: Inputs are piped into the system from disks, which can create I/O bottlenecks. For example, on INET, reading from disk take 15 times longer than graph computation. On DCN, the I/O time is much shorter due to the smaller number of forwarding rules. In fact, in both cases, the I/O is the bottleneck and the CPU is not fully utilized. The runtime remains the same with or without mapping. Hence, the mapping phase is omitted in Table 3.

Memory consumption: In standard MapReduce, intermediate results are flushed to disk after the mapping phase before shuffling, which is very slow on a single machine. We avoid this by keeping all intermediate states

Threads	1	2	4	6	8
Read/s	13.7				
Shuffle/s	7.4				
Reduce/s	46.3	25.8	15.6	12.1	11.1
Speedup	1.00	1.79	2.96	3.82	4.17

a) DCN with 2000 subnets

Threads	1	2	4	6	8
Read/s	170				
Shuffle/s	3.8				
Reduce/s	11.3	5.9	3.2	2.7	2.1
Speedup	1.00	1.91	3.53	4.18	5.38

b) INET with 10,000 subnets

Table 3: Runtime of loop detection on DCN and INET data sets on single machine. The number of subnets is reduced compared to Table 2 so that all intermediate states can fit in the memory. Read and shuffle phases are single-threaded due to the framework limitation.

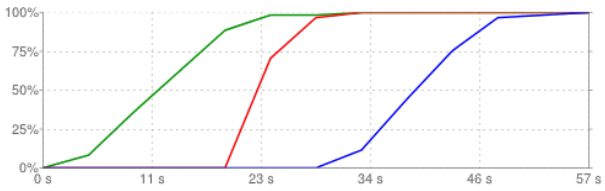


Figure 12: Example progress percentage (in Bytes) of Libra on DCN. The three curves represent (from left to right) Mapping, Shuffling, and Reducing phases, which are partially overlapping. The whole process ends in 57 seconds.

in-memory. However, it limits the number of subnets that can be verified at a time—intermediate results are all matching (subnet, rule) pairs. On a single machine, we have to limit the number of subnets to 2000 in DCN and 10,000 in INET to avoid running out of memory.

Graph size dominates reducing phase: The reducing phase on DCN is significantly slower than on INET, despite INET having 75 times more forwarding rules. With a single thread, Libra can only process 43.2 subnets per second on DCN, compared with 885.0 subnets per second on INET (20.5 times faster). Note that DCN has 35.6 times more nodes. This explains the faster running time on INET, since the time to detect loops grows linearly with the number of edges and nodes in the graph.

Multi-thread: Libra is multi-threaded, but the multi-thread speedup is not linear. For example, on DCN, using 8 threads only resulted in a 4.17 speedup. This effect is likely due to inefficiencies in the threading implementation in the underlying MapReduce framework, although theoretically, all reducer threads should run in parallel without state sharing.

	DCN	DCN-G	INET
Machines	50	20,000	50
Map Input/Byte	844M	52.41G	12.04G
Shuffle Input/Byte	1.61G	16.95T	5.72G
Reduce Input/Byte	15.65G	132T	15.71G
Map Time/s	31	258	76.8
Shuffle Time/s	32	768	76.2
Reduce Time/s	25	672	16
Total Time/s	**57**	**906**	**93**

Table 4: Running time summary of the three data sets. Shuffle input is compressed, while map and reduce inputs are uncompressed. DCN-G results are extrapolated from processing 1% of subnets with 200 machines as a single job.

8.3 Cluster

We use Libra to check for loops against our three data sets on a computing cluster. Table 4 summarizes the results. Libra spends 57 seconds on DCN, 15 minutes on DCN-G, and 93 seconds on INET. To avoid overloading the cluster, the DCN-G result is extrapolated from the runtime of 1% of DCN-G subnets with 200 machines. We assume 100 such jobs running in parallel—each job processes 1% of subnets against all rules. All the jobs are *independent* of each other.

We make the following observations from our cluster-based evaluation.

Runtime in different phases: In all data sets, the sum of the runtime in the phases is larger than the start-to-end runtime. This is because the phases can overlap each other. There is no dependency between different mapping shards. A shard that finishes the mapping phase can enter the shuffling phase without waiting for other shards. However, there is a global barrier between mapping and reducing phases since MapReduce requires a reducer to receive all intermediate values before starting. Hence, the sum of runtime of mapping and reducing phases roughly equals the total runtime. Table 4 shows the overlapping progress (in bytes) of all three phases in an analysis of DCN.

Shared-cluster overhead: These numbers are a lower bound of what Libra can achieve for two reasons: First, the cluster is shared with other processes and lacks performance isolation. In all experiments, Libra uses 8 threads. However, the CPU utilization is between 100% and 350% on 12-core machines, whereas it can achieve 800% on a dedicated machine. Second, the machines start processing at different times—each machine may need different times for initialization. Hence, all the machines are not running at full-speed from the start.

Parallelism amortizes I/O overhead: Through detailed counters, we found that unlike in the single machine case (where I/O is the bottleneck), the mapping and reducing

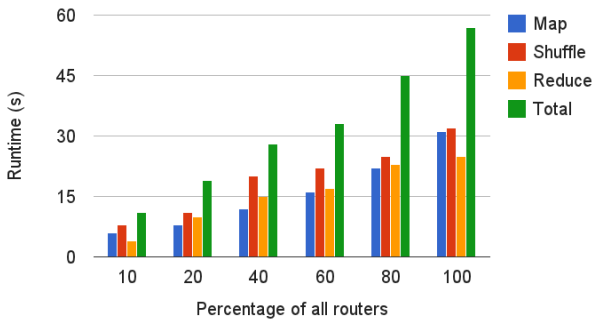


Figure 13: Libra runtime increases linearly with network size.

time dominates the total runtime. We have seen 75%–80% of time spent on mapping/reducing. This is because the aggregated I/O bandwidth of all machines in a cluster is much higher than a single machine. The I/O is faster than the computation throughput, which means threads will not starve.

8.4 Linear scalability

Figure 13 shows how Libra scales with the size of the network. We change the number of devices in the DCN network, effectively changing both the size of the forwarding graph and the number of rules. We do not change the number of subnets. Our experiments run the loop detection on 50 machines, as in the previous section. The figure shows that the Libra runtime scales linearly with the number of rules. The reduce phase grows more erratically than the mapping time, because it is affected by both nodes and edges in the network, while mapping only depends on the number of rules.

Libra's runtime is not necessarily inversely proportional to the number of machines used. The linear scalability only applies when mapping and reducing time dominate. In fact, we observe that more machines can take *longer* to finish a job, because the overhead of the MapReduce system can slow down Libra. For example, if we have more mapping/reducing shards, we need to spend an additional overhead on disk shuffling. We omit the detailed discussion as it depends on the specifics of the underlying MapReduce framework.

8.5 Incremental Updates

Libra can update forwarding graphs incrementally as we add and delete rules in the network, as shown in Section 5.3. To understand its performance, we can breakdown Libra's incremental computation into two steps: (1) time spent in prefix matching (map phase) to find which subnets are affected, and (2) time to do an incremental DFS starting from the node whose routing entries have changed (reduce phase). We also report the total heap memory allocated.

	Map (μs)	Reduce (ms)	Memory (MB)
DCN	0.133	0.62	12
DCN-G	0.156	1.76	412
INET	0.158	<0.01	7

Table 5: Breakdown of runtime for incremental loop checks. The unit for map phase is microsecond and the unit for reduce phase is millisecond.

We measured the time for each of the components as follows: (1) for prefix matching, we randomly select rules and find out all matching subnets using the algorithm described in Section 5.1, and (2) for incremental DFS, we started a new DFS from randomly chosen nodes in the graph. Both results are averaged across 1000 tests. The results are shown in Table 5.

First, we verified that no matter how large the subnet trie is, prefix matching takes almost constant time: DCN-G's subnet trie is 100 times larger than DCN-G's but takes almost the same time. Second, the results also show that the runtime for incremental DFS is likely to be dominated by I/O rather than compute, because the size of the forwarding graph does not exceed the size of the physical network. Even the largest dataset, DCN-G, has only about a million nodes and 10 million edges, which fits into 412MBytes of memory. This millisecond runtime is comparable to results reported in [9] and [11], but now on much bigger networks.

9 Limitations of Libra

Libra is designed for static headers: Libra is faster and more scalable than existing tools because it solves a narrower problem; it assumes packets are only forwarded based on IP prefixes, and that headers are not modified along the way. Unlike, say HSA, Libra cannot process a graph that forwards on an arbitrary mix of headers, since it is not obvious how to carry matching information from mappers to reducers, or how to partition the problem.

As with other static checkers, Libra cannot handle non-deterministic network behavior or dynamic forwarding state (e.g., NAT). It requires a complete, static snapshot of forwarding state to verify correctness. Moreover, Libra cannot tell *why* a forwarding state is incorrect or how it will evolve as it does not interpret control logic.

Libra is designed to slice the network by IP subnet: If headers are transformed in a deterministic way (e.g., static NAT and IP tunnels), Libra can be extended by combining results from multiple forwarding graphs at the end. For example, 192.168.0/24 in the Intra-DC network may be translated to 10.0.0/24 in the Inter-DC network. Libra can construct forwarding graphs for both 192.168.0/24 and 10.0.0/24. When analyzing the two

subgraphs we can add an edge to connect them.

Forwarding graph too big for a single server: Libra scales linearly with both subnets and rules. However, a single reducer still computes the entire forwarding graph, which might still be too large for a single server. Since the reduce speed depends on the size of the graph, we could use distributed graph libraries [16] in the reduce phase to accelerate Libra.

Subnets must be contained by a forwarding rule: In order to break the network into one forwarding graph per subnet, Libra examines all the forwarding rules to decide which rules *contain* the subnet. This is a practical assumption because, in most networks, the rule is a prefix that *aggregates* many subnets. However, if the rule has a longer, more specific prefix (e.g., it is for routing to a specific end-host or router console) than the subnet's, the forwarding graph would be complicated since the rule, represented as an edge in the graph, does not apply to all addresses of the subnet. In this case, one can use Veriflow [11]'s notion of equivalence classes to acquire subnets directly from the rules themselves. This technique may serve as an alternative way to find all matching (subnet, rule) pairs. We leave this for future work.

10 Related Work

Static data plane checkers: Xie et. al introduced algorithms to analyze reachability in IP networks [22]. Anteater [13] makes them practical by converting the checking problem into a Boolean satisfiability problem and solving it with SAT solvers. Header space analysis [10] tackles general protocol-independent static checking using a geometric model and functional simulation. Recently, NetPlumber [9] and Veriflow [11] show that, for small networks (compared to the ones we consider here) static checking can be done in milliseconds by tracking the dependency between rules. Specifically, Veriflow slices the network into *equivalence classes* and builds a *forwarding graph* for each class, in a similar fashion to Libra.

However, with the exception of NetPlumber, all of these tools and algorithms assume centralized computing. NetPlumber introduces a "rule clustering" technique for scalabilty, observing that rule dependencies can be separated into several relatively *independent* clusters. Each cluster is assigned to a process so that rule updates can be handled individually. However, the benefits of parallelism diminish when the number of workers exceeds the number of natural clusters in the ruleset. In contrast, Libra scales linearly with both rules and subnets. Specifically, even two rules have dependency, Libra can still place them into different map shards, and allow reducers to resolve the conflicts.

Other network troubleshooting techniques: Existing network troubleshooting tools focus on a variety of network components. Specifically, the explicitly layered design of SDN facilitates systematic troubleshooting [8]. Efforts in formal language foundations [6] and model-checking control programs [2] reduce the probability of buggy control planes. This effort has been recently extended to the embedded software on switches [12]. However, based on our experience, multiple simultaneous writers in a dynamic environment make developing a bug-free control plane extremely difficult.

Active testing tools [23] reveal the inconsistency between the forwarding table and the actual forwarding state by sending out specially designed probes. They can discover runtime properties such as congestion, packet loss, or faulty hardware, which cannot be detected by static checking tools. Libra is orthogonal to these tools since we focus on forwarding table *correctness*.

Researchers have proposed systems to extract abnormalities from event histories. STS [19] extracts "minimal causal sequences" from control plane event history to explain a particular crash or other abnormalities. NDB [7] compiles packet histories and reasons about data plane correctness. These methods avoid taking a stable snapshot from the network.

11 Conclusion

Today's networks require way too much human intervention to keep them working. As networks get larger and larger there is huge interest in automating the control, error-reporting, troubleshooting and debugging. Until now, there has been no way to automatically verify all the forwarding behavior in a network with tens of thousands of switches. Libra is fast because it focuses on checking the IP-only fabric commonly used in data centers. Libra is scalable because it can be implemented using MapReduce allowing it to harness large numbers of servers. In our experiments, Libra can meet the benchmark goal we set out to achieve: it can verify the correctness of a 10,000-node network in 1 minute using 50 servers. In future, we expect tools like Libra to check the correctness of even larger networks in real-time.

Modern large networks have gone far beyond what human operators can debug with their wisdom and intuition. Our experience shows that it also goes beyond what single machine can comfortably handle. We hope that Libra is just the beginning of bringing distributed computing into the network verification world.

References

[1] Boost Graph Library. `http://www.boost.org/libs/graph`.

[2] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. *NSDI*, 2012.

[3] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM ToCS*, 1985.

[4] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. *NSDI*, 2010.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[6] N. Foster, A. Guha, M. Reitblatt, A. Story, M. Freedman, N. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison. Languages for Software-Defined Networks. *IEEE Communications Magazine*, 2013.

[7] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. Where is the Debugger for my Software-Defined Network? *HotSDN*, 2012.

[8] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley, K. Zarifis, and P. Kazemian. Leveraging SDN layering to systematically troubleshoot networks. *HotSDN*, 2013.

[9] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking using Header Space Analysis. *NSDI*, 2013.

[10] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. *NSDI*, 2012.

[11] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. *NSDI*, 2013.

[12] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for OpenFlow Switch Interoperability Testing. *CoNEXT*, 2012.

[13] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. *SIGCOMM*, 2011.

[14] K. Marzullo and G. Neiger. Detection of global state predicates. *Springer*, 1992.

[15] D. L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 1991.

[16] The Parallel Boost Graph Library. `http://osl.iu.edu/research/pbgl/`.

[17] Protocol Buffers. `https://code.google.com/p/protobuf/`.

[18] Route Views. `http://www.routeviews.org/`.

[19] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker. How Did We Get Into This Mess? Isolating Fault-Inducing Inputs to SDN Control Software. *Technical Report UCB/EECS-2013-8*, 2013.

[20] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with rocketfuel. *IEEE/ACM TON*, 2004.

[21] R. Tarjan. Depth-first search and linear graph algorithms. *12th Annual Symposium on Switching and Automata Theory*, 1971.

[22] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. *INFOCOM*, 2005.

[23] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. *CoNEXT*, 2012.

Software Dataplane Verification

Mihai Dobrescu and Katerina Argyraki
EPFL, Switzerland

Abstract

Software dataplanes are emerging as an alternative to traditional hardware switches and routers, promising programmability and short time to market. These advantages are set against the risk of disrupting the network with bugs, unpredictable performance, or security vulnerabilities. We explore the feasibility of verifying software dataplanes to ensure smooth network operation. For general programs, verifiability and performance are competing goals; we argue that software dataplanes are different—we can write them in a way that enables verification *and* preserves performance. We present a verification tool that takes as input a software dataplane, written in a way that meets a given set of conditions, and (dis)proves that the dataplane satisfies crash-freedom, bounded-execution, and filtering properties. We evaluate our tool on stateless and simple stateful Click pipelines; we perform complete and sound verification of these pipelines within tens of minutes, whereas a state-of-the-art general-purpose tool fails to complete the same task within several hours.

1 Introduction

Software dataplanes are emerging from both research [17,26,27,37] and industry [2,3] backgrounds as a more flexible alternative to traditional hardware switches and routers. They promise to cut network provisioning costs by half, by enabling dynamic allocation of packet-processing tasks to network devices [42]; or to turn the Internet into an evolvable architecture, by enabling continuous functionality update of devices located at strategic network points [41].

Flexibility, however, typically comes at the cost of reliability. A system of non-trivial size that is subject to frequent updates is typically plagued by behavior and performance bugs, as well as security vulnerabilities. It makes sense then that network operators are skeptical about the vision of software dataplanes that are continuously reprogrammed in response to user and operator needs—as they were skeptical a decade ago toward active networking. The question is, has anything changed? Have software verification techniques matured enough to enable us to reason about the behavior and performance of software dataplanes? Or must we accept that frequently reprogrammed software dataplanes will always be less reliable than their static hardware counterparts?

The subject of this work is a verification tool that takes as input the executable binary of a software dataplane and proves that it does (or does not) satisfy a target property; if the target property is not satisfied, the tool should provide counter-examples, i.e., packet sequences that cause the property to be violated. Developers of packet-processing apps could use such a tool to produce software with guarantees, e.g., that never seg-faults or kernel-panics, no matter what traffic it receives. Network operators could use the tool to verify that a new packet-processing app they are considering for deployment will not destabilize their network, e.g., it will not introduce more than some known fixed amount of per-packet latency. One might even envision markets for packet-processing apps—similar to today's smartphone/tablet app markets—where network operators would shop for new code to "drop" into their network devices. The operators of such markets would need a verification tool to certify that their apps will not disrupt their customers' networks.

For general programs, verifiability and performance are competing goals. Proving properties of real programs (unlike searching for bugs) remains an elusive goal for the systems community, at least for programs that consist of more than a few hundred lines of code and are written in a low-level language like C++. A high-level language like Haskell can guarantee certain properties (like the impossibility of buffer overflow) by construction, but typically at the cost of performance.

For software dataplanes, it does not have to be this way: we will argue that we can write them in a way that enables verification and preserves performance. The key question then is: what defines a "software dataplane" and how much more restricted is it than a "general program"? how much do we need to restrict our dataplane programming model so that we can reconcile verifiability with performance?

There are different ways to approach this question: one could start from a restricted, easily verifiable model and broaden it as much as possible without losing verifiability; or, one could start from a popular, but not verifiable model and restrict it as little as necessary to achieve verifiability. We chose the latter in an effort to be practical. We present in this paper the result of working iteratively on two tasks: designing a verification tool for software dataplanes, while trying to identify a minimal set of conditions that a software dataplane must meet in order to be verifiable.

We fundamentally rely on the assumption that software dataplanes follow a pipeline structure, i.e., they are composed of distinct packet-processing elements (e.g., an IP lookup element, an element that performs Network Address Translation or NAT) that are organized in a directed graph and do not share mutable state. Intuitively, the fact that there are no state interactions between elements (other than one passing a packet to another) makes it feasible to reason about each element in isolation, as opposed to having to reason about the entire pipeline as a whole. Software dataplanes that are created with Click [33] typically conform to this structure, and these arguably constitute the majority of research prototypes. We also know of at least one industry prototype that uses Click [1], while the vision of a "composable" dataplane put forward by Intel earlier this year [3] strongly implies a pipeline structure as well.

We aim to prove properties that, in the case of hardware dataplanes, are either taken for granted or can be proved using practical techniques [28–30, 38, 43]: *crash-freedom*, which means that no packet sequence can cause the dataplane to stop executing; *bounded-execution*, which means that no packet sequence can cause the execution of more than a known, reasonable number of instructions; or *filtering* properties, e.g., "any packet with source IP A and destination IP B will be dropped by the pipeline."

In this paper, we describe a verification tool that proves such properties for stateless pipelines (e.g., an IP router or static firewall) and two simple stateful pipelines (a NAT box and a traffic monitor). Certain proofs assume arbitrary configuration[1], while others assume a specific one. For instance, we prove crash-freedom or bounded-execution assuming arbitrary configuration, and such proofs are useful independently of the frequency of configuration changes. In contrast, proving that a pipeline will drop a packet with given headers makes sense only given a specific configuration, and such proofs are useful when configuration changes relatively slowly.

We evaluate our tool by proving crash-freedom and bounded-execution for different Click pipelines. Our proofs complete within tens of minutes, whereas a state-of-the-art general-purpose tool fails to complete the same task within hours. Keeping verification time within minutes is necessary and sufficient given our goals: We envision our tool being used by developers, for instance to ensure that a new piece of packet-processing code cannot seg-fault, or by network operators, for instance to ensure that a given configuration change will not result in undesirable network behavior. In both cases, having to wait for hours would be impractical; waiting for tens of minutes is non-negligible, but on par with the experience of waiting for compilation to complete or configuration to be downloaded to network devices.

Even though we focus on conceptually simple pipelines, performing complete and sound verification on them required overcoming significant challenges (dealing with path explosion, loops, and large data structures). Our contribution is to address these challenges by applying existing verification ideas (symbolic execution [10, 21] and compositionality [4, 20, 22]) and combining them with certain domain specifics of packet-processing software (pipeline structure, bounded loops over packet contents, pre-allocated data structures that expose a key/value store interface). We share common ground with many verification tools, especially the ones that use compositional symbolic execution [4, 22], but those were designed for different goals (increase line coverage or find bugs), so they do not solve our problem.

The rest of the paper is organized as follows: After providing the necessary background (§2), we describe our system (§3) and the properties that it can prove (§4). Then we present our evaluation (§5), discuss limitations (§6) and related work (§7), and conclude (§8).

2 Setup

In this section, we provide background on symbolic execution (§2.1), summarize our approach (§2.2), and describe our basic assumption about the structure of software dataplanes (§2.3).

2.1 Background

Symbolic Execution.

A program can be viewed as an "execution tree," where each node corresponds to a program state, and each edge is associated with a basic block. Running the program for a given input leads to the execution of a sequence of instructions that corresponds to a path through the execution tree, from the root to a leaf. For example, the program E_1 in Fig. 1 may execute two instruction sequences: one for input $in < 0$ and the other for input $in \geq 0$; hence, its execution tree (shown to the right of the program) consists of two edges, one for each "input class" and instruction sequence.

Symbolic execution [10, 21] is a practical way of generating execution trees. During normal execution of a program, each variable is assigned a concrete value, and only a single path of the tree is executed. During symbolic execution, a variable may be symbolic, i.e., assigned a set of values that is specified by an associated constraint. For example, a symbolic integer x with associated constraint $x > 2 \wedge x < 5$ is the set of concrete values $x = \{3, 4\}$. A symbolic-execution engine can

[1] By "configuration" we mean all state that the control plane writes into the dataplane, e.g., the contents of forwarding or filtering tables.

take a program, make the program's input symbolic, and execute all the paths that are feasible given this input.

Consider the program E_2 in Fig. 1 and assume that the input in can take any integer value. To symbolically execute this program, we start at the root of the tree and execute all the feasible paths. As we go down each path, we collect two pieces of information: the "path constraint" specifies which values of in lead to this path, and the "symbolic state" maps each variable to its current value on this path. For example, at the end of path e_4, the path constraint is $C = (in \geq 0 \wedge in < 10)$, and the symbolic state is $S = \{out = 10\}$; at the end of path e_5, the path constraint is $C = (in \geq 10)$, and the symbolic state is $S = \{out = in\}$.

Construction of Proofs.

If we can execute all the feasible paths of a program and verify that none of them violates a target property, that constitutes proof that the entire program satisfies this property. By constructing proofs in this manner, we can also automatically determine all the problematic inputs that prevent us from completing the proof.

However, proof by execution can be rarely used in practice, because of path explosion [9]: The sheer number of feasible paths in a real program (even one that consists of a few hundred lines of code) is typically so large that it is impossible to execute all of them in useful time. This is because the number of paths generally grows exponentially in the number of branching points, and real software has a branching point every few instructions. For instance, when Klee [10] symbolically executes UNIX coreutils like nice or cat, it achieves more than 70% line coverage, but executes less than 1% of the feasible paths [35]. This is fine when the goal is high line coverage or discovery of interesting paths (e.g., to uncover bugs), but not when the goal is to reason about all feasible paths (i.e., to prove properties).

Researchers have been proposing smarter ways to address path explosion [4,20,22,35], but constructing complete and sound proofs for real programs that consist of more than a few hundred lines of code still takes a lot of manual effort [32].

2.2 Our Approach

We observe that symbolic execution is a good fit for packet-processing pipelines, because their special structure can help sidestep path explosion. In a typical pipeline, two elements (stages) never concurrently hold read or write permissions to the same mutable state, regardless of whether that state is a packet being processed or some other data structure. This level of isolation can help significantly with path explosion.

Our approach is to first analyze each pipeline element in isolation, then compose the results to prove properties about the entire pipeline. This reduces by an exponential factor the amount of work that needs to be done to prove something about the pipeline: If each element has n branches and roughly 2^n paths, a pipeline of m such elements has roughly $2^{m \cdot n}$ paths. Analyzing each element in isolation—as opposed to the entire pipeline in one piece—cuts the number of paths that need to be explored roughly from $2^{m \cdot n}$ to $m \cdot 2^n$. In the worst case, the per-element analyses yield that every single pipeline path warrants further analysis—so we end up having to consider all the paths anyway. In practice, we expect that most pipeline paths are irrelevant to the target property, and we only need to consider a small fraction.

Our verifier relies on S2E [13], an automated path explorer with pluggable path analyzers: the explorer uses symbolic execution to drive the target system down multiple execution paths, while the analyzers measure and/or check properties of each such path. We chose S2E for two reasons: First, it performs what is called "in-vivo" (as opposed to "in-vitro") program analysis, i.e., analyzes code that runs within a real (not modeled) software stack; this enables us to analyze a software pipeline without having to model the underlying system—libraries, kernel, drivers, etc. Second, it can directly analyze binaries (as opposed to source code); this enables us to analyze proprietary packet-processing elements, for which we do not have access to the source code. We use S2E as a building block, to symbolically execute pieces of packet-processing code and obtain, for each piece, a set of path constraints and symbolic states.

2.3 Starting Point: Pipeline Structure

We focus on packet-processing pipelines that consist of packet-processing elements, where each element may access three types of state (Table 1):

Packet state is owned by exactly one element at any point in time. It can be read or written only by its owner; the current owner (and nobody else) may atomically transfer ownership to another element. Packet state is used for communicating packet content and metadata between elements. For each newly arrived packet, there is typically an element that reads it from the network, creates a packet object, and transfers object ownership to the next element in the pipeline. Once an element has transferred ownership of a packet, it cannot read or write it any more.

Private state is owned by one element and never changes ownership. It can be read or written only by its owner, and it persists across the processing of multiple packets. A typical example is a map in a NAT element, or a flow table in a traffic-monitoring element.

Static state can be read by any element but not written by any element. This state is immutable as far as the

	Written by	Read by	Transferable ownership
Packet state	owner	owner	yes
Private state	owner	owner	no
Static state	–	any	–

Table 1: Types of packet-processing state.

pipeline is concerned. A typical example is an IP forwarding table.

This structure is not accidental: it is a natural fit for any platform that must perform high-performance streaming. The alternative would be to allow multiple stages of the pipeline to share read/write access to the same data, which would necessarily require synchronization and the unavoidable contention and complexity that comes with it.

3 System

In this section, we describe our system: first how it leverages the pipeline structure to sidestep inter-element path explosion (§3.1); second, how it leverages other aspects of packet processing to sidestep intra-element path explosion resulting from loops (§3.2), large data structures (§3.3), and mutable state (§3.4).

As we describe each technique used by our system, we also state any extra conditions (on top of pipeline structure) that this technique requires from the target software in order to work well. If a software dataplane does not meet these conditions, our tool may not be able to complete a proof for this dataplane.

We will illustrate our system through Fig. 1, which shows a pipeline consisting of two elements. We will use the term *segment* to refer to an instruction sequence through a single element, and the term *path* to refer to an instruction sequence through the entire pipeline. The input in corresponds to a newly received packet, and we assume that this may contain anything, i.e., we make in symbolic and unconstrained.

For illustration purposes, our examples simplify two aspects of our system: first, our example input in is an integer, whereas in reality the input packet object is an array of bytes; second, our example code snippets consist of pseudo-code, whereas in reality S2E takes as input X86 code.

3.1 Pipeline Decomposition

Verification consists of two main steps: step 1 searches inside each element, in isolation, for code that may violate the target property, while step 2 determines which of these potential violations are feasible once we assemble the elements into a pipeline. More specifically, we cut each pipeline path into element-level segments (Fig. 1).

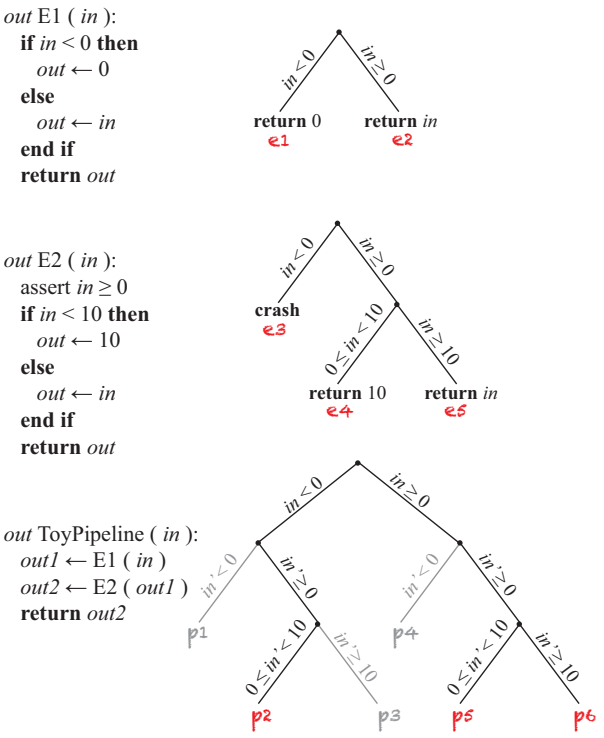


Figure 1: A toy pipeline that consists of two elements.

In step 1, we obtain, for each segment, a logical expression that specifies how this segment transforms state; this allows us to identify all the "suspect segments" that may cause the target property to be violated. In step 2, we determine which of the suspect segments are feasible and indeed cause the target property to be violated, once we assemble segments into paths.

In step 1, we analyze each element in isolation: First, we symbolically execute the element assuming unconstrained symbolic input. Next, we conservatively tag as "suspect" all the segments that may cause the target property to be violated. For example, in Fig. 1, if the target property is crash-freedom, segment e_3 is tagged as suspect, because, if executed, it leads to a crash.

If we stopped at step 1, our verification would catch all property violations, but could yield false positives: If this step does not yield any suspect segments for any element, then we have proved that the pipeline satisfies the target property. For instance, if none of the elements ever crashes for any input, we have proved that the pipeline never crashes. However, a suspect segment does not necessarily mean that the pipeline violates the target property, because a segment that is feasible in the context of an individual element may become infeasible in the context of the full pipeline. For example, in Fig. 1, if we consider element E_2 alone, segment e_3 leads to a crash; however, in a pipeline where E_2 always follows E_1, segment e_3 becomes infeasible, and the pipeline never

crashes. In program-analysis terminology, in step 1, we over-approximate, i.e., we execute some segments that would never be executed within the pipeline that we are aiming to verify.

Step 2 discards suspect segments that are infeasible in the context of the pipeline: First, we construct each potential path p_i that includes at least one suspect segment; each p_i is a sequence of segments e_j. Next, we compose the path constraint and symbolic state for p_i based on the constraints and symbolic state of its constituent segments (that we have already obtained in step 1). Finally, for every p_i, we determine whether it is feasible (based on its constraints) and whether it violates the target property (based on its symbolic state). Note that the last step does not require actually executing p_i, only composing the logical expressions of its constituent segments.

For example, here is how we prove that the pipeline in Fig. 1 does not crash:

Step 1:

1. We symbolically execute E_1 assuming input in can take any integer value. We collect the following constraints and symbolic state for its segments e_1 and e_2.
   - $C_1(in) = (in < 0)$, $S_1(in) = \{out = 0\}$.
   - $C_2(in) = (in \geq 0)$, $S_2(in) = \{out = in\}$.
2. We symbolically execute E_2 assuming input in can take any integer value. We collect the following constraints and symbolic state for its segments e_3, e_4, and e_5:
   - $C_3(in) = (in < 0)$, $S_3(in) = \{crash\}$.
   - $C_4(in) = (in \geq 0 \wedge in < 10)$, $S_4(in) = \{out = 10\}$.
   - $C_5(in) = (in \geq 10)$, $S_5(in) = \{out = in\}$.
3. We tag segment e_3 as suspect.

Step 2:

1. The paths that include the suspect segment are p_1 (i.e., sequence $< e_1, e_3 >$) and p_4 (i.e., sequence $< e_2, e_3 >$).
2. We compute p_1's path constraint as $C_1^*(in) =$

   $C_1(in) \wedge C_3(\ S_1(in)\ [out]\ ) =$
   $C_1(in) \wedge C_3(\ 0\ ) = (in < 0) \wedge (0 < 0) = False.$
3. We compute p_4's path constraint as $C_4^*(in) =$

   $C_2(in) \wedge C_3(\ S_2(in)\ [out]\ ) =$
   $C_2(in) \wedge C_3(\ in\ ) = (in \geq 0) \wedge (in < 0) = False.$
4. Both path p_1's and path p_4's constraints always evaluate to false, hence p_1 and p_4 are infeasible, i.e., there are no feasible paths that include suspect segments, hence the platform never crashes.

Pipeline decomposition enables us to prove properties about the pipeline without having to consider every single pipeline path; but it still requires us to consider every single element segment. This is not straightforward for elements that involve loops, large data structures, and/or mutable private state. We will next discuss how we address each of these scenarios.

3.2 Loops

In general, loops can be a challenge for program verification, especially when the number of loop iterations depends on the input. For example, a loop of t iterations, where t is a 64-bit unsigned integer input, can yield 2^{64} execution paths.

In contrast to general programs, a software dataplane typically will not contain input-dependent loops with such a large number of maximum iterations. A worst-case realistic example is a packet-processing element that loops over the bytes of a packet for encryption or compression; in this case, the number of loop iterations is bounded by the maximum packet size, typically 1500.

Still, loops can create an impractical number of segments within an element. Consider an element that implements the processing of IP options: for each received packet, it loops over the options stored in the packet's IP header and performs the processing required by each specified option type. If the processing of one option yields up to 2^n segments, then the processing of t options yields up to $2^{t \cdot n}$ segments. For example, in the IP-options element that comes with the Click distribution, the processing of 3 options yields millions of segments that—we estimated—would take months to symbolically execute.

To address this, we reuse the idea of decomposition, this time applying it not to the entire pipeline, but to each loop: If a loop has t iterations, we view it as a "mini-pipeline" that consists of t "mini-elements," each one corresponding to one iteration of the loop. We have described how, if we have a pipeline of m elements, we symbolically execute each element in isolation, then compose the results to reason about the entire pipeline. Similarly, if we have a loop of t mini-elements (iterations), we symbolically execute each mini-element in isolation, then compose the results to reason about the entire loop. Unlike a pipeline that consists of different element types, a loop of t iterations consists of the same mini-element type, repeated t times; hence, for each loop, we only need to symbolically execute one mini-element.

This brings us to our first extra condition on packet-processing code: To use decomposition as we do, the only mutable state shared across components must be the packet object itself. For instance, to decompose a pipeline into individual elements, we rely on the fact that the only mutable state shared across elements is the packet object. Similarly, to decompose a loop into individual iterations, the only mutable state shared across iterations must be part of the packet object.

For example, consider again an IP-options element: Such an element typically includes a next variable, which points to the IP-header location that stores the next option to be processed; each iteration of the main loop starts by reading this variable and ends by incrementing it. In a conventional element, next would be a local variable. In our verification-optimized element, next is part of the packet metadata, hence part of packet. And since, in step 1, we make packet symbolic and unconstrained, next is also symbolic and unconstrained, allowing us to reason about the behavior of one iteration of the main loop, assuming that iteration may start reading from *anywhere* in the IP header.

Condition 1 *Any mutable state shared across loop iterations is part of the packet metadata.*

To make a packet-processing element satisfy this condition, a developer needs to identify any variables that are read and written across loop iterations and make these variables part of the packet metadata. For the Click IP-options element, this process required changing 26 lines (12%) of the code and took less than an hour. Alternatively, this can be done automatically by a compiler (that would force developers to explicitly declare mutable state shared across iterations of a loop). Either way, this condition does not restrict the functionality that a packet-processing element can implement; it only forces the developer to create—either manually or with compiler help—an explicit interface between loop iterations.

3.3 Data Structures

Symbolic-execution engines lack the semantics to reason about data structures in a scalable manner. For instance, symbolically executing an element that uses a packet's destination IP address to index an array with a thousand entries will cause a symbolic-execution engine to essentially branch into a thousand different segments—independently from the array content or the logic of the code that uses the returned value. So, if we naïvely feed an element with a forwarding or filtering table of more than a few hundred entries to a symbolic-execution engine, step 1 of our verification process will not complete in useful time.

```
value          =  read ( key )
                  write ( key, value )
{True, False}  =  test ( key )
                  expire ( key, value )
```

Figure 2: An interface for dataplane data structures.

To address this, when we reason about an element, we abstract away any data-structure access; this allows us to symbolically execute the element and identify suspect segments, without requiring the symbolic-execution engine to handle any data structures. To reason about the data structures themselves, we rely on other means, e.g., manual or static analysis; this restricts us to using only data structures that are manually or statically verifiable, but we have evidence that these are typically sufficient for packet-processing functionality.

This brings us to our second extra condition on packet-processing code: To reason about different components of the same executable separately, there must exist a well-defined interface between them. For instance, to reason about each pipeline element separately and compose the results, we rely on the existence of a well-defined interface between each pair of elements, which specifies all the state that can be exchanged between them (the packet object). Similarly, to reason about a data structure separately from the element that uses it and compose the results, the data structure must expose a well-defined interface to the element.

We need an interface that abstracts a data structure as a key/value store that supports at least read, write, membership test, and expiration. The first three operations are straightforward; the last one—expiration—allows an element to indicate that a {key, value} pair will not be accessed by the element any more, hence is ready to be removed and processed by the higher layers. For example, suppose an element maintains a data structure with per-flow packet counters; when a flow completes (e.g., because a FIN packet from that flow is observed), the element can use the expiration operation to signal this completion to the control-plane process that manages traffic statistics.

Condition 2 *Elements use data structures that expose a key/value-store interface like the one in Fig. 2.*

Moreover, we need data structures that expose the above interface *and* can be verified in useful time. When we say that a data structure is "verified," we mean that the implementation of the interface exposed by the data structure is proved to satisfy crash-freedom, bounded-execution, and correctness. The latter depends on the particular semantics of the data structure, e.g., a hash-table should satisfy the following property: a "write (key, value)" followed by a "read (key)" should return

"value." If an element uses only data structures for which these properties hold, then, when we reason about the element, we can abstract away all the data-structure implementations and consider only the rest of the element code plus the data-structure interfaces.

Condition 3 *Elements use data structures that are implemented on top of verifiable building blocks, e.g., pre-allocated arrays.*

As evidence that such data structures exist, we implemented a hash table and a longest-prefix-match table that satisfy crash-freedom and bounded-execution. They both consist of chains of pre-allocated arrays. Our hash table is a sequence of K such arrays; when adding the k-th key/value pair that hashes to the same index, if $k \leq K$, the new pair is stored in the k-th array, otherwise it cannot be added (the write operation returns False). For the longest-prefix-match table, we use the idea of "flattening" of all entries to $/24$ prefixes [25].

We chose arrays as the main building block, because they combine two desirable properties: (a) They enable line-rate access to packet-processing state, because of their $O(1)$ lookup time. (b) They are easy to verify, because of the simplicity of their semantics. For example, a write to an array (that is within the array bounds) is guaranteed not to cause a crash and not to cause the execution of more than a known number of instructions that depends on the particular CPU architecture. In contrast, a write to a dynamically-growing data structure, e.g., a linked list or a radix trie, may result in a variable number of memory allocations, deallocations and accesses, which can fail in unpredictable ways.

Conditions 2 and 3 introduce two kinds of overhead:

First, existing elements may need to be rewritten to satisfy them; this involves replacing existing data structures with ones that satisfy the two conditions and changing any line of code that accesses a data structure. We have not yet applied our approach widely enough to have statistically meaningful results on the corresponding amount of effort. In one case (Click IP lookup element), we had to change about 130 lines (20%) of the code, which took a few hours. In another case (Click NAT element), we had to write the element from scratch (because most of the NAT code is about accessing data structures), which took a couple of days. To reduce this overhead, part of our work is to create a library of data structures that satisfy the two conditions.

Second, implementing sophisticated data structures on top of pre-allocated arrays typically requires more memory than conventional implementations. For instance, the original Click NAT element stores per-connection state in a hash table, implemented as an array of dynamically growing linked lists (when adding the k-th key/value pair that hashes to the same index, the new pair is stored as the k-th item of a linked list). In contrast, our NAT element uses the hash-table implementation outlined above, with $K = 3$ pre-allocated arrays (this value makes the probability of dropping a connection negligible). Hence, our NAT element may use up to 3 times more memory to store the same amount of state. In our opinion, sacrificing memory for verifiability is worth considering given the relative costs of memory and the human support for dealing with network problems.

3.4 Mutable Private State

Mutable private state is hard to reason about because it may depend on a sequence of observed packets (as opposed to the currently observed packet alone). For instance, if an element maintains connection state or traffic statistics, then its private state is a function of all traffic observed since the element was initialized. Hence, it is not enough to reason about the segments of the element that can result from all possible contents of the current packet; we need to reason about the segments that can result from all possible contents of all possible packet sequences that can be observed by the element. The challenge is that symbolic-execution engines (and verification tools in general) are not yet at the point where they can handle symbolic inputs of arbitrary length in a scalable manner.

We can currently verify two kinds of elements that maintain mutable state: a NAT element (maintains per-connection state and rewrites packet headers accordingly) and a traffic monitor (collects per-flow statistics). We believe that our approach can be generalized to other elements, but we do not expect to be able to perform complete and sound verification of an element that performs arbitrary state manipulation—claiming that would be close to claiming that we could verify arbitrary software.

Our approach is to break verification step 1 (§3.1) into two sub-steps: the first one searches for "suspect" values of the private state that would cause the target property to be violated, while the second one determines which of these potential violations are feasible given the logic of the element. In the first sub-step, we assume that the private state can take *any* value allowed by its type (i.e., we over-approximate). In the second sub-step, we take into account the fact that private state cannot, in reality, take any value, but is restricted by the particular type of state manipulation performed by the given element.

So far, we have not needed to exercise the second sub-step in practice: in the two stateful elements that we have experimented with, the first sub-step did not reveal any suspect states, hence the second one was not exercised. We describe both sub-steps through a manufactured example in our technical report [16].

4 Target Properties

In this section, we describe the three target properties that our current prototype can (dis)prove. These properties are expressed imperatively using the S2E analyzer interface [13].

Crash-freedom.

We say that a pipeline is *crash-free* when it is guaranteed not to execute any instruction that would cause it to terminate abnormally, e.g., an assertion with a false argument or a division by zero. The definition of "abnormal termination" depends on the environment where the pipeline runs: in the case of user-mode Click, it is the receipt of a signal (e.g., SIGSEGV, SIGABRT, SIGFPE) that is not handled by the Click process and causes the process to terminate; in the case of kernel-mode Click, it is a call to the kernel's panic method. As stated in §2.2, our verifier is built on top of an in-vivo path explorer, which can detect any of these conditions.

We prove crash-freedom for a pipeline given an arbitrary input packet and arbitrary configuration state. If a pipeline does not include any instruction that may cause abnormal termination, proving crash-freedom is trivial. On the other hand, if a pipeline does include an instruction that may cause abnormal termination, that does not necessarily mean that this instruction may be executed. So, proving crash-freedom is equivalent to proving that any such instruction will never be executed, and proving lack of crash-freedom is equivalent to providing a specific packet and specific state that causes such an instruction to be executed.

Bounded-execution.

We say that a pipeline satisfies *bounded-execution* when it is guaranteed to execute no more than I_{max} instructions per packet. This ensures that no packet is ever caught in an infinite loop. It can also be used to produce a "latency envelope," i.e., argue that once a packet enters the pipeline, it will exit within a bounded amount of time. To translate instruction sequences into latency bounds, we need to map each instruction to the minimum and maximum number of cycles that it can take to complete, which can be typically obtained from the CPU and/or chip manual.

We prove bounded-execution for a pipeline given an arbitrary input packet and arbitrary configuration state. We find the longest path of a pipeline as follows: In step 1 of the verification process, when we symbolically execute an element, we also record the length (number of instructions) of each of its segments. In step 2, we search for the longest feasible path by considering different segment combinations. We use a simple search heuristic that first checks if the path that consists of the longest segment of each element is feasible (if yes, we are done), then checks if any path that involves either the first or second longest segment of each element is feasible, and so on. In the worst case, we have to check all possible segment combinations; in practice, we find the longest feasible path after considering only a few combinations.

Filtering.

Given a pipeline with specific configuration state, can we guarantee that a packet that enters the pipeline with source IP A and destination IP B will be dropped?

We leverage existing work that answers this type of question for hardware dataplanes [28–30, 38, 43]. This work abstracts each network device as a function that maps an input packet header to an output port, and then it composes different device functions to reason about the entire network; the mapping function of each device is determined by the contents of its forwarding table. In contrast, we abstract each packet-processing element as a function that maps an input packet header to an output port, and then we compose different element functions to reason about the entire pipeline; the mapping function of each element is automatically derived by symbolically executing the element's code given an arbitrary input packet.

The main difference lies in the derivation of the mapping function of each packet-processing element (that we do by symbolically executing the element in isolation). This is useful in cases where an element, e.g., includes a line of code that drops all packets with source IP A, even though the device's forwarding table indicates otherwise. Composing element functions to reason about a pipeline is equivalent to composing device functions to reason about a network, and we can reuse the algorithms proposed by the above work.

5 Evaluation

We tested our system on pipelines created with Click. In each tested pipeline, packets are generated by a "generator" element and dropped by a "sink" element; what we verify is the packet-processing code between generator and sink. We answer the following questions: Can we perform complete and sound verification of software dataplanes (§5.1)? How does verification time increase with pipeline length (§5.2)? Can we use our tool to uncover bugs, useful performance characteristics, or unintended dataplane behavior (§5.3)?

5.1 Feasibility

We verified pipelines that consist of various combinations of the elements in Table 2. The table indicates the origin of each element (whether it is an original Click element, one that we modified, or one that we wrote from

Element	New LoC (% of total)	Loops	Data Structs	Mutable State
Click:				
Classifier				
CheckIPhdr				
EthEncap				
EthDecap				
DecTTL				
DropBcast				
Click+:				
IPoptions	26 (12%)	X		
IPlookup	130 (20%)		X	
Ours:				
NAT	870		X	X
TrafficMonitor	650		X	X

Table 2: Verified packet-processing elements. "Click" indicates an unmodified element from Click distribution 2.0.1; "Click+" indicates an element from the same distribution that we modified modestly; "ours" indicates an element that we wrote from scratch. "New LoC" is the number of lines of code that we modified or introduced in each element. "X"s indicate which technique(s) we applied to each element.

scratch). Our modifications consisted of loop rewriting and replacing data structures with our verifiable ones. The table also indicates the number of new lines of code (LoC) and which of our techniques were needed to complete step 1 of the verification process for each element.

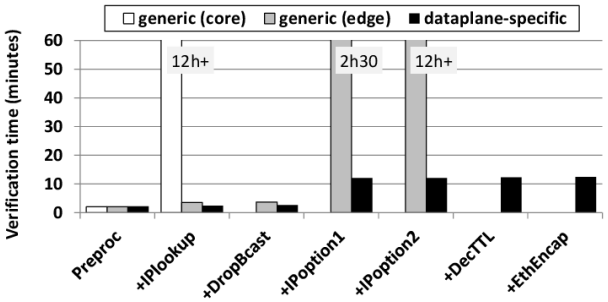
For each pipeline, we proved crash-freedom and bounded-execution. More generally, for each pipeline, we were able to answer questions of the following kind: can line X in element Y be executed with arguments Z in the context of this pipeline? if yes, what is a packet that would cause this line to be executed with these arguments?

5.2 Scalability

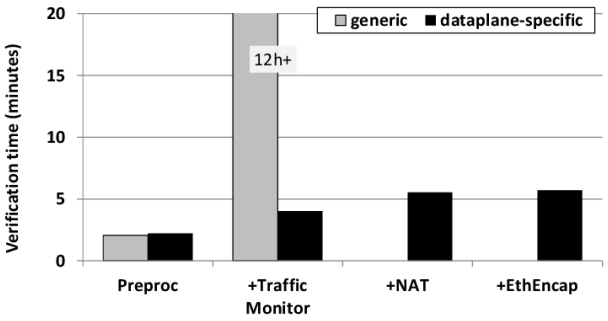
We now examine how verification time increases with pipeline length. Given the intended uses of our tool, it should not take more than a few tens of minutes to prove a target property per pipeline. We first look at meaningful pipelines (that it makes sense to actually deploy), then at microbenchmarks that illustrate different aspects of our system. To show the benefit of our domain-specific techniques, we use as a baseline vanilla S2E—a state-of-the-art, publicly available verification framework for general software. We refer to our tool as "dataplane-specific verification" and to S2E as "generic verification." We feed the same code to the two systems.

Meaningful Pipelines.

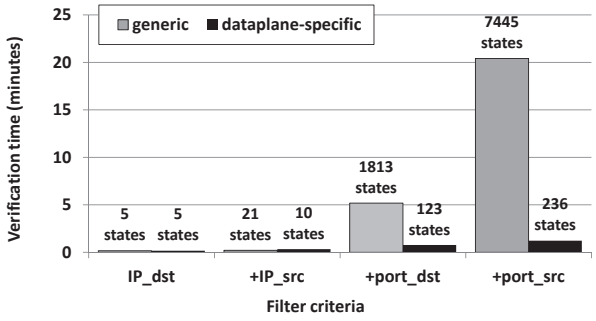
We consider three meaningful pipelines: (a) *edge router* implements a standard IP router (the first 8 elements in Table 2) with a small forwarding table (10 entries); (b) *core router* is similar but has a large forwarding table (100,000 entries); (c) *network gateway* implements NAT and per-flow statistics collection. Each of them presents an extra verification challenge: the first one in-



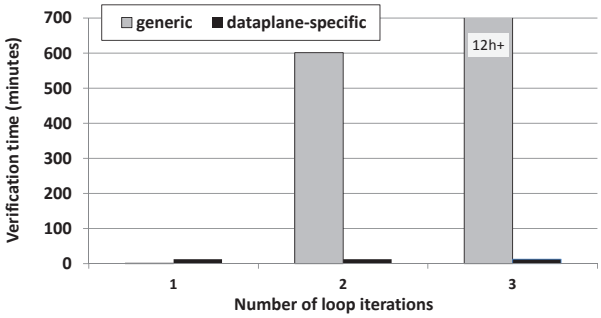
(a) IP router. For the dataplane-specific tool, the results are the same for the edge and core pipelines.



(b) Network gateway



(c) Pipeline microbenchmark



(d) Loop microbenchmark

Figure 3: Verification time as a function of pipeline length. "preproc" consists of the first 3 elements in Table 2.

cludes a loop (in IPoptions), the second one a large data structure (in IPlookup), and the third one mutable private state (in NAT and TrafficMonitor).

Fig. 3(a) shows how verification time increases as we add more elements to the IP-router pipelines: Dataplane-specific verification completes in less than 20 minutes. Most of this time is spent on IP options, because this element has significantly more branching points than the rest. Generic verification of the edge router exceeds 12 hours (at which point we abort it) the moment we allow packets to carry 2 IP options (so we do not show any data point for it beyond "+IPoption2"). Generic verification of the core router exceeds 12 hours the moment we add the IP lookup element to the pipeline (so we do not show any data point for it beyond "+IPlookup"). The difference between the two tools comes from our special treatment of loops and large data structures.

Fig. 3(b) shows the same information for the network-gateway pipeline: Dataplane-specific verification completes in less than 6 minutes, whereas generic verification exceeds 12 hours the moment we add either the TrafficMonitor or the NAT element. The difference comes from the fact that we abstract away data-structure implementations.

Compositionality Microbenchmarks.

We consider two synthetic pipelines to illustrate the benefit of pipeline and loop decomposition. The first one consists of a sequence of simple filtering elements, each of which reads a different part of the input packet's IP header to make a filtering decision. The second pipeline implements a simplified version of the IP options processing loop, i.e., in each iteration, it reads some portion of the IP header, updates it, and advances a next variable that indicates where the next read should start.

Fig. 3(c) shows how verification time increases as we add more filtering elements to the first pipeline: generic verification time increases significantly faster than dataplane-specific verification time. This is because the former executes all feasible segments of each element in isolation, whereas the latter executes all feasible paths of the pipeline. In this scenario, generic verification does complete in useful time, because this pipeline involves few elements, without loops, that access minimal state. Still, it takes an order of magnitude more time than dataplane-specific verification because of the exponential increase in the number of paths. To make this clear, we note, on top of each bar, the number of verification states that each tool generates and processes.

Fig. 3(d) shows how verification time increases as we add more iterations to the loop of the second pipeline: dataplane-specific verification time remains constant, whereas generic verification time increases exponentially. This is because the former executes all feasible segments of one loop iteration, whereas the latter executes all feasible paths of the entire loop. Dataplane-specific verification is slower than generic verification only in the special case where we have a loop with a single iteration. That is because it symbolically executes one loop iteration, assuming that iteration may start reading from *anywhere* in the IP header; this pays off as soon as we add a second loop iteration, but it is unnecessary in the special case of a loop with a single iteration.

5.3 Usefulness

We said that our tool can help developers debug their code, and network operators better understand the performance and behavior of their dataplanes; we now look at a few specific examples.

Bugs in Click Elements.

We found the following while trying to prove crash-freedom and bounded-execution for various Click pipelines:

Bug #1: Any pipeline that includes the Click IP fragmenter element will enter an infinite loop, if it tries to fragment a packet with IP options. This is because the for loop that processes IP options in the fragmenter does not have an increment (the programmer forgot to add one).[2]

Bug #2: Any pipeline that does not include an IP options element but includes the Click IP fragmenter element will enter an infinite loop, if it tries to fragment a packet that carries a zero-length IP option. This is because the current option length determines where the next iteration of the loop will start reading, so, a zero-length option causes the loop to get stuck.[3] The Click IP options element discards any packet with a zero-length option, so including it in the pipeline prevents the bug from being exercised.

Bug #3: Any pipeline that includes the Click NAT element[4] will hit a failed assertion[5], if it receives a packet with source IP address/port tuple $T_s = T$ and destination tuple $T_d = T$, where T is the public IP address/port of the NAT box.

All three bugs constitute security vulnerabilities: they enable any end-host to disable the pipeline by sending a specially crafted packet.

How hard would it be to find these bugs manually? The first one is probably not that hard: a loop missing its increment stands out visually, plus any serious testing of the fragmenter element would involve a packet with IP options. The other two bugs, however, manifest in scenarios that a developer is unlikely to test, but an attacker can easily exploit: fragmentation of an illegal packet

[2] elements/ip/ipfragmenter.cc, line 64
[3] elements/ip/ipfragmenter.cc, line 69
[4] elements/tcpudp/iprewriter.cc
[5] include/click/heap.hh, line 149

Bug	Pipeline	Time	# Paths
#1	Edge router with 1 IP option + Click IP fragmenter	3 min	432
#2	Edge router with 1 IP option + Click IP fragmenter	47 min	8423
#2	Edge router without options + Click IP fragmenter	5 sec	26
#3	Network gateway with Click NAT	5 sec	10

Table 3: Time spent and number of paths composed in verification step 2, when the pipeline contains buggy elements.

while processing of IP options is turned off (which does happen, in practice, for performance or security reasons); and processing of an IP header that would be meaningless in a legitimate packet.

In §2.2, we said that we expected verification step 2 to compose the constraints only for a small fraction of the pipeline paths. Table 3 reports, for a given bug and pipeline, the amount of time spent and the number of paths composed in this step. Consider bug #2: When verifying a pipeline that includes the Click IP fragmenter element, step 1 determines that this element has a suspect segment. If the pipeline does not support IP options, step 2 determines that the suspect segment is feasible in this pipeline (hence the pipeline does not satisfy bounded-execution); this requires finding *one* feasible path that contains the suspect segment, and we succeed after composing the constraints for 26 paths, which takes 5 seconds. If the pipeline supports one IP option, step 2 determines that the suspect element is not feasible in this pipeline; this requires showing that *all* the paths that contain the suspect segment are not feasible, and we succeed after composing the constraints for 8423 paths, which takes 47 minutes. These numbers are consistent with our expectation that, in practice, verification step 2 completes in useful time.

Longest paths in IP router.

We used our tool to construct adversarial—from a performance point of view—workloads for a pipeline implementing a standard IP router. Recent research showed that such a router is capable of multi-Gbps lines rates [17], but this result was obtained using workloads of well-formed packets, not meant to exercise the pipeline's exception paths. Instead, we obtained the pipeline's 10 (it could have been any number) longest paths, as well as the packets that cause them to be executed.

It is not surprising that the longest paths are executed in response to problematic packets that trigger further packet examination and logging; what may be surprising is that these paths execute 2.5 times as many instructions than the most common path. Moreover, these extra instructions are CPU-heavy, i.e., they include memory accesses and system calls for logging; an attacker may cause significant performance degradation by sending a sequence of packets that are specially crafted to exercise these particular paths. This is useful information to a developer, because it reveals to him paths that may require his attention. It is also useful to a network operator, because it reveals to her the performance limits of a pipeline and the workloads that trigger them—allowing her to decide whether it is suitable for her network.

Unintended behavior.

Certain implementations of the Loose Source Record Route (LSRR) IP option may enable illegal traffic to bypass a firewall [23]: An IP router that supports the LSRR option may replace the source IP address of an incoming packet with its own IP address. In this case, any filtering based on the source IP address of the packet that happens *after* the processing of IP options is ineffective. This has been exploited to bypass firewalls, eventually causing network operators to disable LSRR and router manufacturers to change their LSRR implementations.

Our tool would have uncovered this vulnerability. To verify that, we created a pipeline that includes an IP options element followed by a firewall, and we tried to prove that it satisfies the following property: "any packet whose source IP address is blacklisted by the firewall will be dropped." The tool responded that the property is not satisfied, and it provided an example packet that causes it to be violated: a packet with a blacklisted source IP address that carries the LSRR option.

6 Limitations

The key enabler and at the same time limitation of our work is that we focus on software dataplanes that follow a pipeline structure and also satisfy three other conditions (§3). The pipeline structure is a natural fit for dataplanes; most research prototypes are already written this way, and we know of at least one industry prototype as well. Favoring an already popular programming model is, in our opinion, a modest price to pay for verifiability. The other three conditions introduce overheads: existing code may need to be changed to satisfy them (but the resulting code is, in our opinion, easier to read and maintain); compared to their more dynamic counterparts, data structures that satisfy Conditions 2 and 3 typically require more memory (but trading off memory for verifiability is, in our opinion, worth considering).

We currently handle only two specific, simple forms of mutable private state. As stated earlier, we do not expect to be able to completely remove this limitation, but we do expect to expand the range of state-manipulation patterns that we can formally reason about.

Our approach is applicable to packet-processing platforms where each packet is handled by a single processing core and different cores never need to synchronize. We focused on such platforms, because there is compelling evidence that they lead to better performance (by minimizing the number of compulsory cache misses) [17]. We would need new results in order to verify platforms where different cores contend for access to the same data structures.

7 Related Work

Our work is feasible because of advances in program analysis tools for C/C++ code, from Verisoft [19] to modern model checkers [34,40] and tools based on symbolic execution [10,11,13,21]. These tools target general code, so they cannot typically construct complete and sound proofs (which is what we want). Instead, they try to increase line coverage or identify buggy paths *without* having to reason about all the paths of the analyzed program (whereas we want to reason about *all* feasible paths of the analyzed pipeline). There exist tools that prove properties of real programs, but, to the best of our knowledge, they are tailored to specific domains other than dataplanes; notable examples are Astrée [8], SLAM [5, 6], and Terminator [14].

Compositionality has been leveraged before to address path explosion, in compositional dynamic test generation [20] and follow-on work [4,22]. The particular tools evaluated in these proposals use "top-down" composition: when symbolically executing a program, encountering a function triggers the construction of a summary (logical representation) of that function in the context of its caller. This makes sense, because—to the best of our understanding—the goal of this line of work is to maximize line coverage with as little work as possible (ideally, hit each program statement exactly once). We use "bottom-up" composition: we first compute context-free summaries of all elements, then we compose them as necessary to reason about the entire pipeline.

Verification techniques have been used before to debug or verify networked systems (but not dataplanes): Musuvathi and Engler adapted the CMC model checker to test the Linux TCP implementation for interoperability with the TCP specification [39]. Bishop et al. contributed a formal specification of TCP/IP and the sockets API, and they tested existing implementations for conformance to their specification [7]. Killian et al. contributed new algorithms for finding liveness bugs in systems like Pastry and Chord [31]. NICE finds bugs in OpenFlow applications [12]. SOFT tests OpenFlow switches for interoperability with reference implementations [36]. Guha et al. contributed "the first machine-verified [Software Defined Networking] controller" [24].

Ennals et al. contributed a new language for packet-processing applications [18]. The goal of that language was to simplify the "compilation of high-level programs to the distributed memory architectures of modern Network Processors." The proposed language ensured that no two threads referenced the same packet, which is akin to our requirement that no two pipeline elements have access to the same packet.

Finally, an earlier version of this work was presented in a workshop paper [15]. That paper reported the feasibility of proving crash-freedom and bounded-execution only for stateless pipelines; it did not include a scalability analysis (§5.2) or report on bugs found using our approach (§5.3).

8 Conclusions

We presented a verification tool that takes as input a software dataplane and proves that it does (or does not) satisfy properties like crash-freedom, bounded-execution, and filtering. Proving such properties for general software faces fundamental challenges, unsolvable with existing tools; we sidestepped them by applying existing ideas (symbolic execution and compositionality), and combining them with domain specifics of packet-processing code (most importantly, that it is structured as a pipeline of elements that do not exchange mutable state outside the packet itself and its metadata). We evaluated our tool on stateless and two simple stateful Click dataplanes; we were able to perform complete and sound verification of these pipelines within tens of minutes, whereas a state-of-the-art general-purpose tool failed to complete the same task within several hours.

Acknowledgments. We are grateful for the help offered by Stefan Bucur, George Candea, Vitaly Chipounov, Johannes Kinder, Vova Kuznetsov, Christian Maciocco, Dimitris Melissovas, David Ott, Iris Safaka, Simon Schubert, and Cristian Zamfir, as well as our shepherd, Brighten Godfrey, and the anonymous reviewers. This work is supported by an Intel grant and a Swiss National Science Foundation grant.

References

[1] Meraki. http://meraki.cisco.com.

[2] Vyatta Hardware Appliances. http://www.vyatta.com/solutions/physical/appliances.

[3] Intel RFP Announcement: SDN Extensions for Programmable Data Services, 2012.

[4] S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *Proc. of the ACM EuroSys Conference*, 2006.

[6] T. Ball and S. K. Rajamani. SLAM: Debugging System Software via Static Analysis. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 2002.

[7] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets. In *Proc. of the ACM SIGCOMM Conference*, 2005.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. MinÃl', D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[9] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[10] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[11] C. Cadar and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM Conference on Computer Communication Security (CCS)*, 2006.

[12] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, 30(1), 2012.

[14] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[15] M. Dobrescu and K. Argyraki. Toward Verifiable Software Dataplanes. In *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2013.

[16] M. Dobrescu and K. Argyraki. Software Dataplane Verification. Technical Report EPFL-REPORT-197121, EPFL, Switzerland, 2014.

[17] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[18] R. Ennals, R. Sharp, and A. Mycroft. Linear Types for Packet Processing. In *European Symposium on Programming*, 2004.

[19] P. Godefroid. Model Checking for Programming Languages Using Verisoft. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 1997.

[20] P. Godefroid. Compositional Dynamic Test Generation. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 2007.

[21] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[22] P. Godefroid, A. Nori, S. Rajamani, and S. D. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proc. of the ACM Symposium on the Princinples of Programming Languages (POPL)*, 2010.

[23] F. Gont, R. Atkinson, and C. Pignataro. Recommendations on Filtering of IPv4 packets Containing IPv4 Options. http://tools.ietf.org/html/draft-ietf-opsec-ip-options-filtering-05#section-4.3.

[24] A. Guha, M. Reitblatt, and N. Foster. Machine-Verified Network Controllers. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[25] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *Proc. of the IEEE INFOCOM Conference*, 1998.

[26] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proc. of the ACM SIGCOMM Conference*, 2010.

[27] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[28] P. Kazemian, M. Chang, H. Zeng, S. Whyte, G. Varghese, and N. McKeown. Real Time Network Policy Checking using Header Space Analysis. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[29] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[30] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[31] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[32] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

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

[34] D. Kroening, E. Clarke, and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In *Proc. of the Design Automation Conference (DAC)*, 2003.

[35] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient State Merging in Symbolic Execution. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[36] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT Way for OpenFlow Switch Interoperability Testing. In *Proc. of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2012.

[37] Y. Ma, S. Banerjee, S. Lu, and C. Estan. Leveraging Parallelism for Multi-dimensional Packet Classification on Software Routers. In *Proc. of the ACM SIGMETRICS Conference*, 2010.

[38] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *Proc. of the ACM SIGCOMM Conference*, 2011.

[39] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[40] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[41] B. Raghavan, T. Koponen, A. Ghodsi, M. Casado, S. Ratnasamy, and S. Shenker. Software Defined Internet Architecture. In *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets)*, 2012.

[42] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[43] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On Static reachability Analysis of IP Networks. In *Proc. of the IEEE INFOCOM Conference*, 2005.

NetCheck: Network Diagnoses from Blackbox Traces

Yanyan Zhuang[†‡*], Eleni Gessiou[†*], Steven Portzer[◁], Fraida Fund[†],
Monzur Muhammad[†], Ivan Beschastnikh[‡], Justin Cappos[†]
[†]*NYU Poly,* [‡]*University of British Columbia,* [◁]*University of Washington*

Abstract

This paper introduces NetCheck, a tool designed to diagnose network problems in large and complex applications. NetCheck relies on blackbox tracing mechanisms, such as `strace`, to automatically collect sequences of network system call invocations generated by the application hosts. NetCheck performs its diagnosis by (1) totally ordering the distributed set of input traces, and by (2) utilizing a network model to identify points in the totally ordered execution where the traces deviated from expected network semantics.

Our evaluation demonstrates that NetCheck is able to diagnose failures in popular and complex applications without relying on any application- or network-specific information. For instance, NetCheck correctly identified the existence of NAT devices, simultaneous network disconnection/reconnection, and platform portability issues. In a more targeted evaluation, NetCheck correctly detects over 95% of the network problems we found from bug trackers of projects like Python, Apache, and Ruby. When applied to traces of faults reproduced in a live network, NetCheck identified the primary cause of the fault in 90% of the cases. Additionally, NetCheck is efficient and can process a GB-long trace in about 2 minutes.

1 Introduction

Application failures due to network issues are some of the most difficult to diagnose and debug. This is because the failure might be due to in-network state or state maintained by a remote end-host, both of which are invisible to an application host. For instance, data might be dropped due to MTU issues [26], NAT devices and firewalls introduce problems due to address changes and connection blocking [11], default IPv6 options can cause IPv4 applications to fail [8], and default buffer size settings can cause UDP datagrams to be dropped or truncated [49].

Such application failures are challenging for developers and administrators to understand and fix. Hence, numerous fault diagnosis tools have been developed [3, 13, 17, 37, 23, 19]. However, few of these tools are applicable to large applications whose source code is not available. Without source code, administrators often resort to probing tools such as `ping` and `traceroute`, which can help to diagnose reachability, but cannot diagnose application-level issues.

This paper presents NetCheck. In contrast with most prior approaches, NetCheck does not require application- or network-specific knowledge to perform its diagnoses, and no modification to the application or the infrastructure is necessary. NetCheck treats an application as a blackbox and requires only a set of system call (syscall) invocation traces from the relevant end-hosts. These traces can be easily collected at runtime with standard blackbox tracing tools, such as `strace`. To perform its diagnosis, NetCheck derives a global ordering of the input syscalls by simulating the syscalls against a network model. The model is also used to identify those syscalls that deviate from expected network semantics. These deviations are then mapped to a diagnosis using a set of heuristics.

NetCheck diagnosis output is intended for application developers and network administrators. NetCheck outputs high-level diagnosis information, such as "an MTU issue on a flow is the likely cause of loss," which may be useful to network administrators. NetCheck also outputs detailed low-level information about the sequence of system calls that triggered the high-level diagnosis. This information can help developers locate the underlying issue in the application code.

This work makes the following three contributions:

- **Accurate diagnosis of network issues from plausible global orderings.** Because of complex network semantics, it is not always possible to globally order an input set of host traces without a global

*The two authors are co-primary authors.

clock. NetCheck approximates the true ordering by generating a plausible ordering of the input traces. We show that for 46 of the bugs reproduced from public bug-trackers, this strategy correctly detected and diagnosed over 90% of the bugs. Additionally, NetCheck found and diagnosed a new bug in VirtualBox [49].

- **Modeling expected network behavior to identify unexpected behavior.** By using a model of an idealized network environment NetCheck is capable of diagnosing issues even in applications that execute in complex environments. We demonstrate that this approach is effective at detecting many real-world problems, including failures reported in bug trackers of projects like Python and Apache, and problems in everyday applications such as Pidgin, Skype and VirtualBox.

- **Efficient algorithm for finding plausible global orderings.** We present a heuristic trace-ordering algorithm that utilizes valuable information inherent in network API semantics. We prove that our algorithm has a best-case linear running time and demonstrate that NetCheck needs less than 1 second to process most of the traces studied in this paper (Section 6.4). Even on large traces, such as a 1 GB trace collected from Skype, NetCheck completes in less than two minutes.

The following section provides an overview of NetCheck. Section 3 describes the challenges and corresponding contributions of this work. Details of NetCheck's design and implementation are given in Sections 4 and 5, respectively. In Section 6, we evaluate the accuracy, effectiveness, and efficiency of NetCheck. Section 7 outlines limitations of NetCheck and our future work. Related work is discussed in Section 8 and we conclude with Section 9.

2 NetCheck Overview

To use NetCheck, a user needs to first gather a set of *host traces* for an application using a tool like `strace`, `dtrace`, `ktrace`, or `truss`. The user invokes NetCheck with a configuration file that lists the host trace files to analyze and the IP addresses of the hosts. A host trace, as in Figure 1, is a sequence of *syscall invocations* at a single host. A syscall invocation is a 4-tuple that includes (1) a string, such as `socket` denoting the name of the syscall, (2) the arguments passed to the invoked syscall, (3) the returned value, and optionally, (4) an error number (errno) that is returned upon a syscall failure.

For example, the first line of the host A trace in Figure 1 is `socket(...)=4`, which is a `socket` syscall invocation with a return value of 4. For certain syscalls the value is returned through an argument pointer[1]. Figure 1 shows an example of this: `recv` call on host A passes a buffer to a location in memory where the kernel writes a 5-byte string indicated by one of the logged arguments ("Hola!"). For clarity we omit some arguments and errno from syscall invocations in this paper.

```
     Host A trace:                      Host B trace:
A1. socket(...)             = 4    B1. socket(...)             = 3
A2. bind(4, ...)            = 0    B2. connect(3, ...)         = 0
A3. listen(4, 1)            = 0    B3. send(3, "Hello", ...) = 5
A4. accept(4, ...)          = 6
A5. recv(6, "Hola!", ...) = 5
```

Figure 1: An example input trace detailing a TCP connection between two hosts. Many system call arguments are omitted for readability. Returned values (including buffer contents) are underlined. Data sent by host B ("Hello") has been modified in-transit before being received by host A ("Hola!").

```
A1. socket(...)          = 4
B1. socket(...)          = 3
A2. bind(4, ...)         = 0
A3. listen(4, 1)         = 0
B2. connect(3,...)       = 0
A4. accept(4, ...)       = 6
B3. send(3,"Hello",...) = 5
A5. recv(6,"Hola!",...) = 5
```

Figure 2: A valid global ordering of syscall invocations from the two host traces in Figure 1.

The example traces in Figure 1 indicate an error with the network. Host B sends a 5-byte string "Hello" to A, but A receives "Hola!", a different 5-byte string. Used independently, the two host traces are insufficient to identify this issue — the corresponding `send` and `recv` calls both returned successfully. To detect the problem, a developer must manually reason about both the order in which the calls occurred (their serialization) and the underlying behavior of the calls (their semantics). For the traces in Figure 1, the logical serialization of the two host traces reveals a semantic problem: what was received is different from what was sent. NetCheck automatically detects this and other issues by serializing the traces, simulating the calls, and then observing their impact on network and host state.

To detect and diagnose network problems such as the issue in Figure 1, NetCheck uses a **global ordering** that it automatically reconstructs from the input set of black-box host traces. Figure 2 shows one global ordering for the two input traces in Figure 1. A valid global ordering must preserve the local orders of host traces, and conform to the network API semantics. For example, the local ordering at host A in Figure 1 requires that `bind` occur after `socket` has returned successfully. And, `connect` at host B cannot be ordered before `listen` at host A, as such an ordering violates the network API se-

[1] NetCheck expects the host traces to include such return values, which are provided by most common tools.

```
Host A trace:                Host B trace:
A1. send("hello") = 5        B1. send("hi") = 2
A2. recv("hi")    = 2        B2. recv()     = -1, EWOULDBLOCK
```

(a) Two input host traces. All operations are performed on a single connected TCP socket.

```
Valid ordering 1:            Valid ordering 2:
B1. send("hi")    = 2        A1. send("hello")= 5
B2. recv()      = -1, EWOULDBLOCK  B1. send("hi")   = 2
A1. send("hello")= 5         A2. recv("hi")   = 2
A2. recv("hi")    = 2        B2. recv()     = -1, EWOULDBLOCK
```

(b) Two valid orderings of (a): (left) `recv` returned `EWOULDBLOCK` because the data has not been sent yet. (right) `recv` returned `EWOULDBLOCK` because the content is still in the network.

```
A1. send("hello") = 5
A2. recv("hi")    = 2
B1. send("hi")    = 2
B2. recv()        = -1, EWOULDBLOCK
```

(c) An invalid ordering of (a): data is received before being sent.

Figure 3: An example illustrating the ambiguity of reconstructing a valid order from two host traces considered by NetCheck. For the two host traces in (a), there are two possible valid orderings in (b). An invalid ordering, such as (c), will never be produced by NetCheck.

mantics.

NetCheck reconstructs the global ordering without relying on globally synchronized clocks or logical clocks [18, 31]. Both approaches require modification of the existing systems, and incur performance overhead and complexity[2]. Instead, NetCheck uses a **heuristic algorithm** and a **network model** to simulate and check if a particular ordering of syscall invocations is feasible. As a result, NetCheck has a higher level of transparency and usability.

Next, we overview the challenges that NetCheck faces in diagnosing network issues in complex applications, and then describe the contributions of our work.

3 Challenges and Contributions

Challenge 1. Accuracy: ambiguity in order reconstruction. Reconstructing a global order of traces collected from edge hosts without a globally synchronized clock is sometimes impossible. For example, Figure 3(b) lists two valid orderings of the traces in Figure 3(a). In the ordering shown on the left the `recv` call on B failed because it occurred before the `send("hello")` call on A. In the ordering shown on the right the `send("hello")` call on A occurred before the `recv` call on B, but network delay prevented B from receiving the message when `recv` was invoked. The ordering in Figure 3(c) is invalid since data must be sent before it is received. However, even if invalid orderings are eliminated, from the traces in Figure 3(a) it is impossible to tell if the `send` call on A occurred before or after the `recv` call on B. How can NetCheck diagnose issues without being able to reconstruct what actually happened?

Challenge 2. Network complexity: diagnosing issues in real networks. The host traces that we consider are blackbox traces: they omit information regarding the physical network or the environment in which the traces were collected. How can NetCheck diagnose network issues without this crucial information?

Challenge 3. Efficiency: exploring an exponential space of possible orderings. The space of the potential sequences is exponential in the length of the host traces and the number of hosts. Exhaustive exploration of this space to find an ordering is intractable even at small trace lengths (e.g., 30 – 100 syscalls). Real-world applications, such as a Pidgin client, make over 100K syscall invocations in a single execution. Given this huge space of possible orderings, how can NetCheck efficiently find problems in user applications?

NetCheck handles each of the above three challenges as follows:

Contribution 1. Deriving a plausible global ordering as a proxy for the ground truth. NetCheck approximates the true ordering by generating a plausible ordering of the input traces that preserves the host-local orderings of syscalls (Section 4.1). For this, NetCheck assumes that syscalls are atomic: a syscall runs to completion before the next syscall in the trace. Our evaluation shows that for 46 of the bugs reproduced from public bug-trackers, NetCheck correctly detects and diagnoses more than 90% of the problems (Sections 6.1 and 6.2). Additionally, NetCheck fails to find a plausible ordering in only 5% of the input traces that we studied in our evaluation.

Contribution 2. Modeling expected simple network behavior to identify unexpected behavior. NetCheck tackles the complexity of an application's execution environment by modeling an idealized network (Section 4.2). We rely on the fact that, from the network edge, network behavior can be described with a simple model due to the end-to-end principle. Our network model is based on Deutsch's Fallacies [15, 16, 39], and encodes misconceptions commonly held by developers, such as: the network is reliable, latency is zero, hosts communicate over a direct link, etc. NetCheck detects and diagnoses network problems and application failures by finding deviations from this ideal model of the network (Section 4.3). Our evaluation (Section 6) demonstrates that this approach is effective at detecting many real-world problems, including failures reported in bug trackers of projects like Python and Apache, and problems in everyday applications such as Pidgin, Skype and VirtualBox.

Contribution 3. A best-case linear time algorithm to find a plausible global ordering. We present a

[2]Over 90% of syscall invocations we observed completed in less than 0.1 ms. Widespread and practical clock-synchronization techniques do not provide a sufficiently fine timing granularity to unambiguously order traces from multiple hosts.

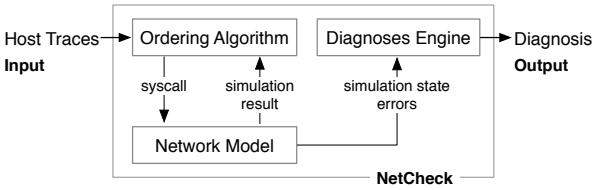


Figure 4: Overview of NetCheck.

Algorithm 1 NetCheck pseudo code.

```
1:  function NETCHECK(trace_0, ..., trace_n-1)
2:    // trace_i is a list of syscall invocations at host i
3:    netModel = new POSIXNetworkModel()
4:    (orderError, permReject) = (False, False)
5:    try:
6:      // Call Algorithm 2 for trace ordering
7:      OrderingAlg(trace_0,...,trace_n-1, netModel)
8:    catch OrderError:
9:      orderError = True
10:   catch PermanentReject:
11:     permReject = True
12:   diagnosis = DiagnosesEngine(netModel.state,
                                  orderError, permReject)
13:   Output diagnosis
```

Algorithm 2 Trace ordering algorithm pseudo code.

```
1:  function ORDERINGALG(trace_0, ..., trace_n-1, netModel)
2:    // trace_i is a list of syscall invocations at host i
3:    while (trace_0, ..., trace_n-1) not empty do
4:      // Record topmost calls
5:      topCalls = top(trace_0, ..., trace_n-1)
6:      // Assign each call a priority (see Table 1)
7:      topCalls.sort()
8:      // Process each call in topCalls or raise OrderError
9:      while True do
10:       if topCalls empty then // No calls can be processed
11:         raise OrderError
12:       // Highest priority call remaining
13:       call_i = topCalls.dequeue()
14:       outcome = netModel.simulate(call_i)
15:       if outcome == ACCEPT then // Completed this call
16:         ordered_trace.push(trace_i.pop())
17:         break // Break to outer while
18:       else if outcome == REJECT then
19:         continue // Continue in inner while
20:       else // outcome == PERMANENT_REJECT
21:         raise PermanentReject
22:     end while
23:   end while
```

heuristic trace-ordering algorithm that utilizes valuable information inherent in network API semantics. The best case running time of our algorithm is linear in the length of the input host traces. In the worst case, our algorithm is asymptotic with the length of the input host traces times the number of traces. Our evaluation demonstrates these bounds and illustrates that NetCheck needs less than 1 second to process most of the traces studied in this paper (Section 6.4). Even on large traces, such as a 1 GB trace collected from Skype, NetCheck completes in less than two minutes.

Next, we explain NetCheck's design in further detail.

4 NetCheck Design

First, NetCheck orders the syscalls with a heuristic algorithm and a network model. Following this, NetCheck uses a diagnoses engine to compile any detected deviations from the network model into a diagnosis. These steps are overviewed in Figure 4 and Algorithm 1. The next three sections describe each of these steps in further detail.

4.1 Ordering host traces

Algorithm 2 lists the pseudocode of the trace-ordering algorithm in NetCheck. The algorithm traverses the set of all input traces in local-host order, and at each iteration considers the calls that are at the top of each of the host traces (maintained in a priority queue `topCalls`, defined on line 5). In each iteration of the outer while loop (line 3), one of the calls from `topCalls` is processed. If this is not possible because no call at the top of the trace can be executed, an `OrderError` is raised. This completes execution of the ordering algorithm (returning to Algorithm 1) and thus no further calls are processed.

To find a call in `topCalls` to process, the algorithm performs two steps. First, the algorithm sorts `topCalls` (line 7) according to syscall priorities in Table 1. These priorities are derived from the dependency graph in Figure 5 (discussed below). Second, the algorithm simulates the highest priority call using the network model (see Section 4.2). This simulation can result in one of three outcomes: (1) accept the call (line 16) and continue with the outer while loop iteration, (2) reject the call (line 19) and then try another call in priority order from `topCalls`, or (3) a `PermanentReject` exception is raised (line 21) — the syscall can never be processed by the model, return back to the NetCheck algorithm (Algorithm 1 line 10).

Prioritizing syscalls. The key to the efficiency of the order reconstruction algorithm is to simulate syscalls in an order that is derived from the POSIX syscalls dependency graph in Figure 5. This graph was created by examining the POSIX specification for each system call and looking at which calls can modify the state used by other calls. This graph can be used to derive a priority value for each syscall (for simplicity we use integer priority values): if syscall `x` may-depend-on `y`, then `x` has a higher priority value and should be simulated before `y` (Table 1). For example, according to Figure 5, `connect` should be simulated before `listen`. This scheme is justified because processing a syscall `y` in the network model could affect `x` and make it impossible to simulate `x` without undoing `y`. This helps NetCheck to avoid significant

Priority value	Syscalls
0	socket, bind, getsockname, getsockopt, setsockopt
1	poll, select, getpeername
2	accept, recv, recvfrom, recvmsg, read
3	connect, send, sendto, sendmsg, write, writev, sendfile
4	close, shutdown, listen

Table 1: Simulation priority of common syscalls: the lower the priority value, the higher the priority.

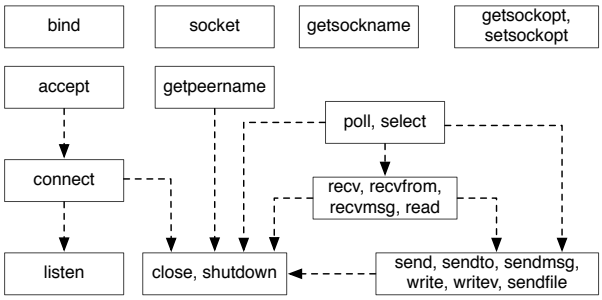


Figure 5: Dependency graph of system calls in the POSIX networking API. Edges represent the *may-depend-on* relation.

backtracking in the case where the return value of `x` requires that `y` has not yet occurred.

Syscall prioritization enables the ordering algorithm to permanently process a syscall after trying (in the worst case) the top syscall on each trace. The inner while loop (line 9) iterates through the top-most syscalls on each trace and removes them from a priority queue (thus considering each syscall at most once per inner loop execution). If this syscall is accepted (lines 15–17), then `pop` removes it from trace_i (line 16) which permanently consumes the syscall (there is no way to later undo this call). If this call cannot currently be processed and is rejected (lines 18–19), then it will be placed again in `topCalls` (line 5) after a syscall in the current priority queue (and thus on the top of a trace) is consumed. Therefore, in the worst case, Algorithm 2 will never backtrack beyond the current rejected call. This makes the trace-ordering algorithm in Figure 2 *efficient* — its best-case running time is linear in the length of the input host traces if no backtracking occurs, and its worst-case running time is the length of the input host traces multiplied by the number of host traces i.e., the number of hosts (see Section 6.4).

4.2 Model-based syscall simulation

The network model component of NetCheck simulates syscalls (line 14 in Algorithm 2) to determine if a given syscall can be added as the next syscall in the global order. The network model treats the network and the application that generated the traces as a blackbox and requires no application-specific information.

To simulate a syscall, the model uses the current network and host states tracked by the model, and network semantics defined by the POSIX API. The network model state includes information related to the observed connections/protocols (e.g., pending or established TCP connections), buffer lengths and their contents, datagrams sent/lost, etc. Simulating a syscall with a model results in one of three determinations: accept the call, reject the call, or permanently reject the call.

```
A2. bind(3, ...)        = 0
A3. listen(3, 1)        = 0
B2. connect(3, ...)     = 0
```

(a) One valid ordering: all syscalls returned successfully.

```
A2. bind(3, ...)        = 0
B2. connect(3, ...)     = -1, ECONNREFUSED
A3. listen(3, 1)        = 0
```

(b) A second valid ordering: `connect` returned `ECONNREFUSED`.

Figure 6: An example that demonstrates how return values of syscalls can guide trace ordering.

A key technique used by the model to determine if a syscall can be accepted is to use the syscall's logged return and errno values. As an example, Figure 6 shows two possible orderings for `bind` and `listen` calls at host A, and a `connect` call at host B. For `connect` to return 0 (success), it is necessary for `listen` to have already occurred. So, a return value of 0 by `connect` indicates that (a) is a valid ordering. However, had `connect` at host B returned -1 (and a connection refused errno), then (b) would have been the valid ordering.

The current state of the model determines if the model can accept a syscall invocation with a specific return value. In certain cases the model can accept a syscall with a range of possible return values. For example, if the model's receive buffer for a connection has n bytes, then the return value of the `read` syscall (number of bytes read) may be a value between 0 and n.

We now explain how NetCheck produces one of the three outcomes through Figure 7, which illustrates how NetCheck processes the log in Figure 1. We use `A.x` to denote a syscall `x` at host `A`. When a call is accepted, the vertex of this syscall and all of its incoming edges are removed in the next step in the figure; the removed call is added to the final output ordering in Figure 7(l). If a call is rejected, the dashed-arrow *may-depend-on* relation edge between two syscalls is converted into a solid-arrow *depends-on* relation edge. The network model produces one of three outcomes:

Accept the syscall: the simulation of the syscall is successful, and the return value and errno match the logged values. In Figure 7(a), the syscalls in priority queue `topCalls` (defined on line 5 of Algorithm 2) are `A.socket` and `B.socket`, shaded in yellow. The priority of the two calls are the same, so either of them could be simulated. In this case, `A.socket` is simulated, accepted, and then removed from the trace. In Figure 7(a),

this corresponds to removing the vertex and all its incoming edges to generate Figure 7(b). Similarly, `B.socket` and `A.bind` are accepted and removed from the traces in Figure 7(b) and (c), indicated by the bold circle around each syscall.

Reject the syscall: the network model cannot simulate the syscall because the model is not in a state that can accept the syscall with its logged return value and errno. This indicates that the syscall should be simulated at a later point. In Figure 7(d), the calls `A.listen` and `B.connect` are dependent according to Figure 5. In Table 1, `connect` has a higher priority than `listen`, so `B.connect` is simulated first. (This is necessary because of situations like Figure 6(b) where `connect` fails because `listen` was not yet called.) However, the network model rejects this syscall (indicated by a bold and red circle) because for B to successfully connect to A, A must be actively listening. In Figure 7(e), the directed edge from `B.connect` to `A.listen` indicates that `B.connect` should be ordered after `A.listen`. As `A.listen` is the next call with the highest priority in topCalls, it gets simulated and accepted — its vertex and all incoming edges are removed in Figure 7(f).

We describe all of the cases in which our NetCheck prototype rejects a syscall on our wiki[3].

Permanently reject the syscall: there are errors during the simulation of the syscall and the call can never be correctly simulated at a future point. In Figure 7(j) and (k), the network model first accepts `B.send` and then attempts to simulate `A.recv`. This triggers an error since the content in the receiving buffer of `A`, "`Hola!`", is different from the content in the send buffer. No additional system calls will allow `A.recv` to correctly complete in the future.

4.3 Fault diagnoses engine

When NetCheck finished processing the trace (Algorithm 1), either through consuming all actions, finding an order error, or permanently rejecting an action, the state of the model contains valuable information. The diagnoses engine analyzes the model simulation state and any simulation errors to derive a diagnosis. The diagnoses engine makes the simulation results more meaningful to an administrator who might be tasked with resolving the issue.

The diagnoses engine infers a diagnosis based on a set of rules. If the simulation state matches a rule, then the corresponding diagnosis is emitted. Table 2 summarizes the rules; our technical report contains example output for each of the rules [56]. Although the diagnosis rules are heuristics, Section 6 shows that these are effective at detecting problems in a wide range of applications.

Figure 8 lists an example of NetCheck output for the `multibyte` unit test from Table 4. In this test, the server incorrectly uses byte size to calculate the content-length of an HTTP header. This gives the wrong value for HTTP responses with multi-byte characters and the client fails to get the entire content that it requested. This fault was listed on the Ruby bug tracker (test case 2.16 in [56]).

The output in Figure 8 consists of three parts, each of which can be optionally omitted. Part (1) lists non-fatal errors uncovered through simulation by the network model in the form of a snippet of the valid ordering derived with *OrderingAlg* and any model deviations at the syscall level. This information is useful to application developers to understand the series of actions leading to the fault. In the figure, the model detected that there is still data remaining in the buffer at Browser even though both `close` on Browser and `shutdown` on Server returned successfully. Part (2) presents statistics for the observed connections, which may be useful for network administrators to perform performance debugging or see loss / MTU issues. Finally, part (3) present a high-level diagnosis summary generated by the diagnoses engine, which is of interest to all users of NetCheck. Part (3) of Figure 8 shows that the network connection with outstanding data has been shut down by the Browser. This is due to an application-level miscommunication between the Browser and Server.

In larger applications, small network errors can accumulate to cause an application failure. For example, an MTU problem that can only be detected after considering a loss pattern across multiple transmission attempts. Packet loss is not unexpected as the network may drop packets, but diagnoses engine correctly diagnoses this as an MTU issue by considering the pattern of loss over the entire trace set (Section 6.2).

5 Implementation

NetCheck consists of 5.1K lines of Python code and is released freely under an MIT license [33]. The implementation supports the widely used POSIX network API, including support for common flags and optional arguments. This includes all of the syscalls that operate on file descriptors, and optional flags observed in traces of popular applications. It took 2 person-months to implement the network model.

NetCheck currently supports traces generated by `strace` on Linux. We are developing parsers for traces generated by other syscall tracing tools, such as `truss` on Solaris and `dtrace` on BSD and Mac OSX, to process these traces in a uniform manner [35].

[3] https://netcheck.poly.edu/projects/project/wiki/network_model

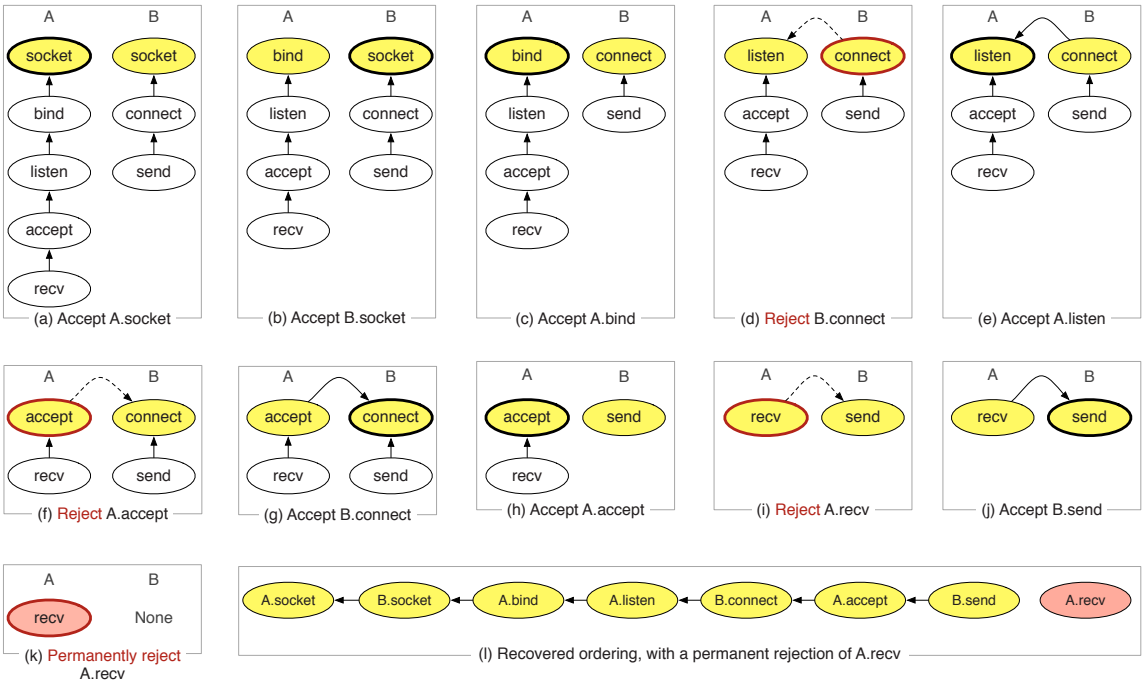


Figure 7: A step-by-step demonstration of how the ordering algorithm (Algorithm 1) processes the traces in Figure 1. Vertices are syscall invocations and solid edges represent dependency — capturing both the local ordering constraint and dependencies between remote syscall invocations. Shaded yellow vertices in each step represent the syscall invocations in the `topCalls` list in Algorithm 2. The syscall invocation simulated by the model at each step of the algorithm is circled in bold. A dashed edge denotes the may-depend-on relation from Figure 5. Each model simulation step either accepts or rejects a syscall. If accepted, the vertex of the syscall and all its incoming edges are removed in the next step, and the call is placed in the final output ordering (l). Steps (d), (f), and (i) show steps in which the syscall invocations were rejected (converting the may-depend-on relation edge into a depends-on relation edge). In step (k), the `A.recv` is permanently rejected because the traces in Figure 1 contain a bug: what was received is different from what was sent. The final output ordering in (l) orders all of the calls, except for the permanently rejected `A.recv`.

```
(1) Verifying Traces
------------------------------------------------------
Serve: write(6, "<html>\n<head>\n<title> ... ") = 343
Browser: read(3, "<html>\n<head>\n<title> ... ") = 302
Browser: close(3) = 0
=> NONEMPTY_BUFFER: Socket closed with data in buffer.
Server: read(6, ...) = -1, ECONNRESET (Connection reset)
=> UNEXPECTED_FAILURE: Recv failed unexpectedly.
Server: shutdown (6) = 0
=> ENOTCONN: [Application Error] Attempted to shutdown
             a socket that is not connected.
------------------------------------------------------
(2) TCP Connection Statistics
------------------------------------------------------
Connection from Browser (128.238.38.67:40830) to Server
(128.238.38.71:3000)
* Data sent to accepting Server: 114 bytes sent,
  114 bytes received, 0 bytes lost
* Data sent to connected Browser: 517 bytes sent,
  476 bytes received, 41 bytes lost (7.93%)
------------------------------------------------------
(3) Possible Problems Detected
------------------------------------------------------
* Browser has 1 TCP connection to 128.238.38.71:3000
  with data in the buffer
* Connection to Server has been reset by Browser
* Server attempted to shutdown an unconnected socket
* Data loss is most likely due to application behavior
```

Figure 8: NetCheck's output for the `multibyte` unit test from Table 4.

Rule	Description
1. Unaccepted Connections	If a TCP connection has unaccepted (pending) connections, this is an indicator that the connecting host may be behind a middlebox.
2. Ignored Accepts	No matching connect corresponding for an accept (middlebox indicator).
3. Connect Failure	Connect fails for reasons other than (1) or (2), indicating that a middlebox (e.g., NAT) is filtering network connections.
4. Connection Refused	Connection is refused to an address that is being listened on (middlebox indicator).
5. Nonblocking Connect Failure	A nonblocking connect never connects (middlebox indicator).
6. No Relevant Traffic	A host has outgoing traffic, but not to a relevant address, then the host is likely connecting through a proxy.
7. Datagram Loss	A significant (user-defined) fraction of datagrams are lost.
8. MTU	Datagrams larger than a certain size are dropped by the network.
9. Non-transitive Connectivity	*A* can communicate with *B*, *B* can communicate with *C*, but *A* cannot communicate with *C*.

Table 2: NetCheck post-processing diagnoses. Example of rule (1): when a client is behind a NAT, (i) the client uses a private IP, (ii) the peer socket address in server's accept is not the client's IP.

6 Evaluation

We evaluated NetCheck in four ways. First, we examined network issues in popular applications reported on public bug trackers (Section 6.1). NetCheck diagnosed known bugs across multiple projects with a rate of 95.7%, demonstrating its **accuracy**. Second, we replicated failures on a real network, the WITest testbed [51], and used NetCheck to diagnose the issues (Section 6.2). This result indicates that we can diagnose real network issues as **deviations from a simple model of expected network behavior**. Third, we used NetCheck to diagnose the root cause of faults in widely-used applications, such as FTP, Pidgin, Skype and VirtualBox (Section 6.3). Finally, we evaluated NetCheck's performance across all of the test cases and applications detailed in our evaluation and proved that the algorithm has a best-case linear running time (Section 6.4). This demonstrates that NetCheck is **efficient**. A detailed description of all the bug traces in this section, and the corresponding bug reports are provided on our wiki [33] and in our technical report [56].

6.1 Diagnosing Bugs Reported in Bug Trackers

As noted in Section 3, the first challenge for NetCheck is to reconstruct a global ordering of traces. To evaluate NetCheck's **accuracy**, we collected bugs from public bug trackers of 30 popular projects. We targeted networked-related bugs that are small, reproducible, and have a known cause. We did not intentionally select bugs based on how NetCheck works. For each bug report we reproduced the issue from the report description by writing code to cause the observed behavior. This generated a total of 71 traces. Table 3 lists the results of running NetCheck on these traces. The traces are grouped into 24 categories based on the exhibited bug or behavior. Note that not all traces produce a bug. Some traces generate different behaviors on different OSes and can potentially lead to portability problems. For example, the `loopback_address` category captures the case when a socket exhibited different behaviors bound to the local interface (0.0.0.0 or 127.0.0.1). We briefly review four bugs from this evaluation.

1. MySQL provides a server-side option to only accept TCP connections from the loopback interface. If a user connects to the local IP address of the host that is hosting the MySQL server configured with this option, the connection is refused. This option provides extra security, but it is also difficult to debug. NetCheck correctly diagnoses the root cause as a socket bound to the loopback interface attempting to connect to a non-loopback address — `bind_local_interface` in Table 3.

2. When a Skype call's quality degrades, the user often terminates and restarts the call or restarts Skype. This may cause a known issue: when a TCP/UDP socket is waiting on `recv`/`recvfrom`, a `close` call made on the socket from a different thread will keep the socket blocking *indefinitely*. This bug has also been reported on GCC and Ruby bug trackers [10, 45]. We reproduced this bug, and NetCheck successfully diagnosed it (`block_tcp/udp_close_socket` in Table 3).

Bug category (number of traces)	Bugs detected & correctly diagnosed / # Bugs
bind_local_interface (2)	2 / 2
block_udp_close_socket (2)	1 / 1
block_tcp_close_socket (2)	1 / 1
broadcast_flag (2)	2 / 2
buffer_full (1)	1 / 1
invalid_port (3)	3 / 3
loopback_address (7)	0 / 0
multicast_issue (3)	3 / 3
multiple_bind (1)	1 / 1
nonblock_connect (13)	8 / 9
nonblock_flag_inheritance (2)	1 / 1
oob_data (5)	5 / 5
readline (1)	0 / 0
recvtimeo (1)	1 / 1
setsockopt_misc (3)	2 / 2
shutdown_reset (1)	0 / 1
sigstop_signal (3)	0 / 0
so_linger (5)	2 / 2
so_reuseaddr (2)	2 / 2
tcp_nodelay (3)	0 / 0
tcp_set_buf_size_vm (4)	4 / 4
udp_large_datagram_vm (2)	2 / 2
udp_set_buf_size_vm (2)	2 / 2
vary_udp_datagram (1)	1 / 1
Total Number of Traces: 71	**44 / 46 (95.7%)**

Table 3: Evaluating NetCheck on reported network bugs.

3. Different interpretations of network APIs have resulted in OS portability issues. For example, implementations of `accept` vary in whether file status flags, such as `O_NONBLOCK` and `O_ASYNC`, are inherited from a listening socket [1]. This has caused faults in a variety of applications, including Python's socket implementation [2, 34]. This issue is successfully diagnosed by NetCheck (`nonblock_flag_inheritance` in Table 3).

4. Variations in socket API implementations can also have security implications. On Windows, an application can exploit the `SO_REUSEADDR` socket option to deny access to, or impersonate, services listening on the same local address. Over a dozen major software projects [47, 43, 41, 50, 42, 36] include platform-specific code to mitigate security risks associated with `SO_REUSEADDR`. This issue is successfully diagnosed by NetCheck (`so_reuseaddr` in Table 3).

Overall, NetCheck correctly detected and diagnosed **95.7%** of the 46 reported bugs we considered. This indicates that NetCheck is **accurate**. Our technical report [56] further details the test cases in Table 3.

6.2 Diagnosing Injected Bugs in a Testbed

Deployed applications run in complex networking environments. To evaluate whether NetCheck can diagnose issues in real networks, we conducted an experiment on the WITest testbed [51] — a networking environment

Bug	Total number of diagnoses	Bug detected	Incorrect diagnoses
bind6	1	✓	0
bind6-2	1	✓	0
clientpermission	2	✓	0
closethread	3	✗	0
conn0	1	✓	0
firewall	2	✓	1
gethostbyaddr	1	✗	0
httpprox1	1	✓	0
httpprox2	1	✓	0
keepalive	1	✓	0
local	3	✓	0
max1	3	✓	0
maxconn	4	✓	0
maxconn2	6	✓	0
mtu	2	✓	0
multibyte	2	✓	0
noread	1	✓	0
permission	1	✓	0
portfwd	1	✓	0
special	1	✓	0
Total:	**38**	**18/20 (90%)**	**1/38 (3%)**

Table 4: NetCheck's classification of controlled network bugs. The total number of diagnoses are all the issues detected by NetCheck. The ✓/✗ indicate the success/failure of NetCheck in diagnosing the root cause of the bug. Incorrect diagnoses are false positives.

for studying wireless connectivity. We used a typical setup for this testbed, running a client-side application that issued requests to a WEBrick [48] HTTP server. For the experiment, an administrator replicated and injected network-related bugs from the WEBrick tracker into the testbed. We then gathered trace data from both hosts with `strace`. Table 4 lists the results from this evaluation. We now review two categories of bugs that NetCheck successfully diagnosed: IPv6 compatibility, and issues related to middleboxes.

IPv6 compatibility. With pervasive IPv6 deployment, applications are beginning to set IPv6-only socket options by default. However, IPv6 lacks backward compatibility and can break many legacy IPv4 applications [8]. For example, the IPv6-only option breaks the networking functionality in Java and triggers a "network unreachable" exception [21]. Other applications including Eclipse, VNC, Google Go, etc., will generate similar exceptions that are too generic to diagnose the root cause. NetCheck's model expects that IPv6 does not preempt IPv4 in an idealized environment, making it possible to detect this incompatibility by tracking inconsistent addressing schemes (`bind6`/`bind6-2` bugs in Table 4). As a result, NetCheck generates a much more informative diagnosis of this issue [56].

Middleboxes. Middleboxes, such as firewalls and NATs, introduce features that impact most network applications. NetCheck can be used to detect and diagnose the effects of middleboxes on an application. For example, in an FTP session, a `PASV` command on the client-side allows the server to define an IP/port that the client can use to connect to the server and receive data. The negotiated port is usually a high numbered port on the server that is typically blocked by firewalls on the server-side. NetCheck can detect and diagnose this kind of failure (`firewall` bug in Table 4).

Note that for many of the injected bugs, NetCheck also provided accurate supplementary information (included in the total diagnoses count in Table 4), such as packet loss information and buffer state. For example, if a connection is closed with data in the receive buffer, then NetCheck warns that the connection might have been closed prematurely by the receiver.

Our evaluation of injected bugs in a controlled network environment (Table 4) shows that despite the complexity of a real network, NetCheck's idealized network model is **effective**, diagnosing **90%** of the injected bugs with a false positive rate of 3%. The injected bugs used in this study are detailed in [56].

6.3 Bugs in Popular Applications

We also evaluated NetCheck on traces generated by large, widely used applications — an FTP client, Pidgin, Skype and VirtualBox. The issues detailed in this section were uncovered through normal, practical use of the applications on university and home networks. NetCheck produced useful diagnosis for problems across all of these applications (Table 5).

While using an **FTP client** to interact with an FTP server, we noticed that certain commands, such as `cd` and `pwd`, executed successfully, while others, like `ls` or `get`, would not be processed. When one of the latter command was issued, the FTP client was locked up until the connection timed-out. We used NetCheck to diagnose the issue by applying it to traces from the server and the client. NetCheck's diagnoses are summarized in Table 5. The problem is that commands like `ls` and `get` are followed by a `PORT` command from the client to inform the server about the IP/port that the client is listening on to receive the data (default behavior in most FTP clients). Since the client was behind a NAT, the destination IP address in the `PORT` command was a local address. Therefore, all connection attempts from the server failed to reach the client. NetCheck correctly identified that the problem is that the client was behind a NAT. The diagnoses engine in NetCheck did this by detecting a difference between the IP of the client's original connection to the server and the IP specified in the `PORT` command.

Pidgin is a commonly used chat client application. Pidgin clients communicate with each other via an XMPP server. During a group meeting, one of the users (user A) was repeatedly dropped from the group conversation; another user (user B) could not log in with the Pidgin client. Traces were gathered from all the Pidgin clients and the XMPP server (4 hosts in total), and NetCheck was used to diagnose the network issues.

Application: Issue	NetCheck Diagnoses	Trace Size
FTP: Could issue only some of the commands.	• Client is behind NAT. • 42% data loss.	Client: 245 KB Server: 497 KB
Pidgin: Loss of connection; file transfer and login failure.	• The IP address that getsockname returns is different from the one the socket is bound to. • Message being received that has not been sent.	Client1: 49 MB Client2: 67 MB Client3: 81 MB Server: 93 MB
Skype: Poor call quality and messages lost.	• Data loss due to delay. • A different thread attempts to close sockets. • Client is behind NAT.	Client1: 831 MB Client2: 1.6 GB
VirtualBox: Silent drop of large UDP datagrams.	• Virtualization misbehavior. • Guest OS: MTU. • Host OS: UDP buffer size mismatch.	2.5 MB

Table 5: NetCheck diagnosis of faults in popular applications.

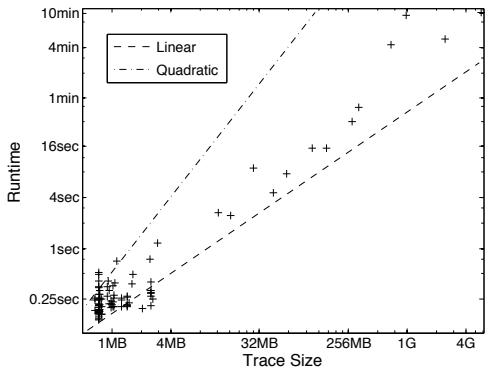


Figure 9: Runtime performance overhead of NetCheck. Data includes all traces in Sections 6.1–6.3.

NetCheck detected two important issues:

(1) Invocations of the `getsockname` syscall by user A's Pidgin client repeatedly returned an IP address that was different from the address that the socket was bound to. User A was multi-homed and was connected to both an ethernet and a wireless network. The wireless connection was poor, causing the default IP address of user A to change as she disconnected and reconnected to the wireless network. In this case, the network model simulated a `connect` syscall call on a socket bound to one IP to an endpoint that expected a different IP address. The `connect` generated a permanent rejection (line 21 in Algorithm 2). NetCheck therefore diagnosed the problem as a mismatch between the intended IP address and the IP address actually used.

(2) User B's Pidgin client could send data to the XMPP server, but all of the server's responses were lost. User B was behind a firewall that filtered packets with high-numbered source port values, including port 5222 on the XMPP server. By examining the model state towards the end of the trace, specifically considering the pattern of packet loss, diagnoses engine observed the selective filtering and diagnosed the problem as a middlebox issue.

Skype is a widely used VoIP service and instant messaging client. During a Skype session we noticed poor call quality and that messages were frequently dropped. Traces from two instances of Skype were gathered and evaluated with NetCheck. NetCheck detected that both clients were behind a NAT and that network delay caused severe data loss. NetCheck also revealed that when call quality degrades, Skype attempts to close its sockets from a separate thread. However, this does not terminate the blocked socket operation, but instead hangs the thread that is blocked on this operation. This is a known issue for some operating systems — a `close` call on a blocking socket from a different thread will keep the socket blocking indefinitely on `recv` or `recvfrom` [10, 45] (also `block_tcp/udp_close_socket` in Table 3). NetCheck diagnoses this issue correctly. The diagnoses engine also outputs a potential solution for Skype developers as part of the diagnoses: invoking `shutdown` on the blocking socket immediately unblocks `recv/recvfrom`.

VirtualBox is a popular tool for running virtualized operating system instances. We found and diagnosed a new bug in VirtualBox using Netcheck – applications running in a Linux VirtualBox instance on a Windows host OS would discard UDP datagrams of size over 8 KB when sent over an interface with VirtualBox's NAT virtual adapter [49]. When run on application traces gathered from the Linux instance (the *guest OS*), NetCheck correctly diagnoses the UDP datagram loss as a MTU issue. This is because from the standpoint of the guest VM, UDP datagrams over a certain size are discarded.

However, the root cause for this bug is as follows. The default receive buffer size on Windows is 8 KB. When the receive buffer is not full, Windows sockets can hold at least one more datagram even if the total datagram size exceeds the buffer size. When VirtualBox queries the socket for the amount of received data, Windows returns either the total size of datagrams in the buffer, or the buffer size, whichever is smaller. When a datagram larger than 8 KB is placed in the receive buffer, VirtualBox believes that the available datagram is only 8 KB and allocates an 8 KB application buffer. VirtualBox then silently drops the large datagram. To understand the usefulness of NetCheck for diagnosing this bug in VirtualBox, we collected traces of syscalls made by VirtualBox to the Windows *host OS* (`udp_set_buf_size_vm` in Table 3) to reproduce this issue. The network model of NetCheck correctly indicates the issue as being a UDP buffer size mismatch interfering with datagram delivery. Therefore, on the host OS, NetCheck also produces an accurate diagnosis of the root cause.

Our evaluation shows that NetCheck is effective at diagnosing faults in large applications in practical use.

6.4 Performance

Dynamically recording syscall invocations of complex applications can produce huge traces. Therefore, effi-

ciency is a key challenge to designing a diagnosis tool that relies on logged syscall information. Figure 9 shows NetCheck's running time for traces of varying lengths. The figure plots the data for all the traces mentioned in this paper. Note that both the x and the y axes of Figure 9 have a logarithmic scale (i.e., a quadratic function is a straight line). The figure illustrates that NetCheck's performance across the various input traces lies between a quadratic and a linear function. NetCheck completes in less than 1 second on most traces, and even on 1 GB long traces, NetCheck completes in less that two minutes[4]. These measurements demonstrate that NetCheck is **efficient** for practical use.

Algorithm complexity. We now consider the complexity of the trace-ordering algorithm (Algorithm 2) — the key algorithm in NetCheck. Let n be the number of hosts, and l be the sum of the lengths of all input host traces. The inner while loop in Algorithm 2 must accept one syscall (if it rejects all syscalls, then the algorithm terminates). In the best case, this loop accepts a syscall on the first iteration, and in the worst case it must run n times to accept a syscall. The outer while loop iterates until all syscalls are accepted, or a total of l times. Therefore, Algorithm 2 makes l simulation calls in the best case, and $n*l$ simulation calls in the worst case. Supposing that the model simulates a syscall in constant time, the worst-case running time of Algorithm 2 is $O(nl)$ and its best-case running time is $O(l)$. Typically, the number of logged syscalls is much larger than the number of hosts, so the runtime will trend towards $O(l)$.

Tracing overhead. To evaluate the overhead of `strace`, we micro-benchmarked the unit tests in Section 6.1, both with and without `strace`, 1K times. The overhead of `strace` across these runs, measured in elapsed time, had a median of 41 ms (0.79%), which is negligible. The standard deviation was 60 sec, due to the varying I/O behavior of the programs. Furthermore, in our experience, the overhead of `strace` on larger applications, such as Pidgin and Skype, was not perceptible.

6.5 Ordering Heuristic Efficacy

Algorithm 2 in Section 4.1 is a heuristic. One potential problem is if this algorithm terminates early: line 11 terminates the trace-ordering algorithm when no calls in `topCalls` can be accepted. Such a termination can occur before NetCheck detects an issue in the application traces. To evaluate the frequency of this early termination one of the authors manually inspected all of the traces collected in Sections 6.1 and 6.2, i.e., bug trackers evaluation (71 traces) and the testbed evaluation (20 traces). On just 2 of these traces (of 91), or 2.2%, did NetCheck not find any bugs and terminated without fully processing all the syscalls. This indicates that the ordering heuristic in NetCheck is effective at reconstructing plausible orderings that can then be used for diagnosis.

[4]This suggests that NetCheck can derive multiple plausible orderings without a significant performance penalty and use these to diagnose the issue. However, this would make NetCheck unnecessarily more complex and it already achieves high accuracy (Section 6.1).

7 Limitations

IPC blindspots. NetCheck cannot detect faults that do not impact an application's syscall trace. This limitations impacts two situations. First, if an application uses non-socket IPC mechanism, then NetCheck will not see the resulting network traffic. For example, the `gethostbyaddr` error in Section 6.2 is due to an issue in how DNS requests are handled. Since DNS requests are handled in part by a non-native program `avahi`, the application's `strace` information does not include the relevant calls. However, NetCheck can be extended to parse `strace` call data and arguments to handle application-specific situations, for instance, to better understand DNS resolution errors reported by `avahi`.

Network blindspots. NetCheck observation of and reasoning about the network is limited to what is captured in system call traces. For example, NetCheck does not know the state of the OS network buffers, the network topology, etc. However, NetCheck's reliance on traces allows it to process previously generated traces, which is useful for reproducing and diagnosing bug reports.

Dynamic analysis. By relying on observed behavior, NetCheck can be considered as a dynamic analysis technique. As such, it cannot diagnose latent application behaviors that are possible, but have not been observed. However, dynamic analysis allows NetCheck to diagnose application issues that arise in deployment, such as those due to in-network state.

Improving the diagnoses engine. The diagnoses engine in NetCheck may be further improved with machine learning [3, 52]. For example, a supervised machine learning approach can be used to derive a signature from application traces or network packet traces, which can then be labeled according to previously observed patterns of correct and incorrect behavior [13].

Network model completeness. The network model in NetCheck simulates the behavior of network syscalls in an idealized network. To correctly perform this simulation, the model must be faithful and complete. Currently, the network model implements all syscalls and optional flags observed in traces of popular applications (Section 6). We continue to refine and improve this model as we encounter important new behaviors.

Multithreading. Currently, NetCheck cannot model systems with multiple threads that access shared resources (e.g., use the same socket descriptor). Improving multithreading support is part of our future work.

8 Related Work

Blackbox diagnosis. Aguilera et al. introduced an important blackbox approach to debugging distributed systems [4]. In this approach, observations of a distributed executions are used to infer causality, dependencies, and other characteristics of the system. This approach was relaxed in later work to produce more informative and application-specific results in Magpie [5] and X-Trace [19]. This prior work focuses on tracking request flows through a distributed system. BorderPatrol [27] is another approach that traces requests among binary modules. In contrast, NetCheck is a blackbox approach to diagnosing network issues in applications.

Khadke et al. [24] introduced a performance debugging approach that relies on system call tracing. Unlike this prior work, NetCheck does not assume synchronized clocks and reconstructs a plausible global ordering.

Ordering events in a distributed setting. The happens-before relation logically orders events in a distributed system [28]. This relation can be realized with vector time, which produces a partial ordering of events in the system [18, 31]. Vector time requires non-trivial instrumentation of the application. NetCheck reconstructs a plausible order of the captured syscalls through heuristics, without modifying the application.

Globally synchronized clocks can order events across hosts. However, over 90% of syscall invocations we observed completed in less than 0.1 ms. Achieving synchronization at a granularity that is sufficient to order syscalls at hosts on a LAN is expensive and difficult.

Log mining. Prior work that uses dynamically captured logs of a program's execution is extensive and includes work on detecting anomalies [12, 22, 52, 30], linking logs and source code [55], identifying performance bugs [40, 44], and generating models to support system understanding [7, 6]. In contrast to this work, NetCheck's focus is on diagnosing network issues from logs of syscalls, though prior work on log mining can be used to expand the scope of NetCheck.

Debugging distributed systems. Techniques for debugging distributed systems are relevant to NetCheck's context of diagnosing network issues in applications. Many tools exist for run-time checking of distributed systems. These tools monitor a system's execution and check for specific property violations [37, 20, 29, 54, 14]. NetCheck is a more light-weight approach to diagnose issue observed through the syscall interface. This makes NetCheck broadly applicable, but it also limits the kinds of issues that NetCheck can uncover (see Section 7).

Specification and runtime verification. Substantial work has been done in validating API and protocol behaviors, e.g., finding faults in Linux TCP implementation [32], SSH2 and RCP [46], BGP configuration [17], and identifying network vulnerabilities [38]. Rigorously specifying protocols and APIs for testing and trace validation has also been described in [9]. These techniques are effective at finding bugs in an API or a protocol, but are not effective when the environment and networking semantic are also contributing factors. NetCheck can diagnose issues even if the input traces are valid API actions. Further, the simplicity of the NetCheck approach is one of its key advantages over prior work.

Application-specific fault detection. Pip [37] and Coctail [53] are distributed frameworks that enable developers to construct application-specific models, which have proven effective at finding detailed application flaws. However, to utilize these methods, a knowledge of the nature of the failures needs to be acquired, and the specific system properties must be specified. NetCheck diagnoses application failures without application-specific models. Khanna [25] identifies the source of failures using a rule base of allowed state transition paths. However, it requires specialized human-generated rules for each application.

9 Conclusion

This work proposes NetCheck, a tool for fault detection and diagnosis in networked applications. NetCheck is a blackbox technique that performs its diagnosis on an input set of traces of syscall invocations from multiple application hosts. NetCheck derives a plausible global ordering as a proxy for the ground truth, and uses a model of expected and simple network behavior to identify and diagnose unexpected behavior.

Our evaluation demonstrates that NetCheck is accurate and efficient. It correctly diagnosed over 95% of faults from traces that reproduce faults reported on bug trackers of 30 popular open-source projects. When applied to injected faults in a testbed, NetCheck identified the main cause in 90% of the cases. Furthermore, we used NetCheck to diagnose issues in large applications, such as Skype and VirtualBox, and in VirtualBox NetCheck found a new bug. We proved that NetCheck's algorithm derives a plausible global ordering in best-case linear running time and that it is efficient in practice.

Our experience with NetCheck demonstrates that it is possible to have an application-agnostic tool that provides practical and accurate fault diagnosis. NetCheck is freely available for download [33].

Acknowledgements

We thank our shepherd Dejan Kostic, Ulrike Stege for discussing ordering algorithms with us, and our reviewers for their invaluable feedback. This work was supported in part by the National Science Foundation through Awards 1223588 and 1205415, NSF Graduate Research Fellowship Award 1104522, the NYU WIRELESS research center and the Center for Advanced Technology in Telecommunications (CATT).

References

[1] accept. Accessed 2/27/2014, http://pubs.opengroup.org/onlinepubs/009695399/functions/accept.html.

[2] add SOCK_NONBLOCK and SOCK_CLOEXEC to socket module. Accessed 2/27/2014, http://bugs.python.org/issue7523.

[3] AGGARWAL, B., BHAGWAN, R., DAS, T., ESWARAN, S., PADMANABHAN, V. N., AND VOELKER, G. M. Netprints: diagnosing home network misconfigurations using shared knowledge. In *NSDI* (2009).

[4] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[5] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *OSDI* (2004).

[6] BESCHASTNIKH, I., BRUN, Y., ERNST, M. D., AND KRISHNAMURTHY, A. Inferring Models of Networked Systems from Logs of their Behavior with CSight. In *ICSE* (2014).

[7] BESCHASTNIKH, I., BRUN, Y., SCHNEIDER, S., SLOAN, M., AND ERNST, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *FSE* (2011).

[8] Biggest mistake for IPv6: It's not backwards compatible, developers admit. Accessed 2/27/2014, http://www.networkworld.com/news/2009/032509-ipv6-mistake.html.

[9] BISHOP, S., FAIRBAIRN, M., NORRISH, M., SEWELL, P., SMITH, M., AND WANSBROUGH, K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *SIGCOMM* (2005).

[10] Cannot interrupt blocking I/O calls with close(). Accessed 2/27/2014, http://gcc.gnu.org/bugzilla/show_bug.cgi?id=15430.

[11] CHEN, K.-T., HUANG, C.-Y., HUANG, P., AND LEI, C.-L. Quantifying skype user satisfaction. In *ACM SIGCOMM Computer Communication Review* (2006), vol. 36, ACM, pp. 399–410.

[12] CHEN, M., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In *DSN* (2002).

[13] COHEN, I., GOLDSZMIDT, M., KELLY, T., SYMONS, J., AND CHASE, J. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *OSDI* (2004).

[14] DAO, D., ALBRECHT, J., KILLIAN, C., AND VAHDAT, A. Live debugging of distributed systems. In *Compiler Construction* (2009), Springer, pp. 94–108.

[15] Fallacies of distributed computing. Accessed 2/27/2014, http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing.

[16] Deutsch's fallacies, 10 years after. Accessed 2/27/2014, http://java.sys-con.com/node/38665.

[17] FEAMSTER, N., AND BALAKRISHNAN, H. Detecting BGP configuration faults with static analysis. In *NSDI* (2005).

[18] FIDGE, C. J. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference* (1988).

[19] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-Trace: A pervasive network tracing framework. In *NSDI* (2007).

[20] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[21] net.ipv6.bindv6only=1 breaks some buggy programs. Accessed 2/27/2014, http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=560238#54.

[22] JIANG, G., CHEN, H., UNGUREANU, C., AND YOSHIHIRA, K. Multi-resolution abnormal trace detection using varied-length N-grams and automata. In *International Conference on Automatic Computing* (2005).

[23] KANDULA, S., MAHAJAN, R., VERKAIK, P., AGARWAL, S., PADHYE, J., AND BAHL, P. Detailed diagnosis in enterprise networks. *ACM SIGCOMM Computer Communication Review 39*, 4 (2009), 243–254.

[24] KHADKE, N., KASICK, M. P., KAVULYA, S. P., TAN, J., AND NARASIMHAN, P. Transparent system call based performance debugging for cloud computing. In *Workshop on Managing Systems Automatically and Dynamically (MAD)* (2012).

[25] KHANNA, G., CHENG, M., VARADHARAJAN, P., BAGCHI, S., CORREIA, M., AND VERÍSSIMO, P. Automated rule-based diagnosis through a distributed monitor system. *IEEE Transactions on Dependable and Secure Computing 4*, 4 (2007).

[26] KNOWLES, S. IESG advice from experience with path MTU discovery. RFC, Internet Engineering Task Force, 1993.

[27] KOSKINEN, E., AND JANNOTTI, J. Borderpatrol: isolating events for black-box tracing. In *Eurosys* (2008).

[28] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[29] LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J., WU, M., KAASHOEK, M. F., AND ZHANG, Z. D3S: Debugging deployed distributed systems. In *NSDI* (2008).

[30] LOU, J.-G., FU, Q., YANG, S., XU, Y., AND LI, J. Mining invariants from console logs for system problem detection. *ATC* (2010).

[31] MATTERN, F. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms 1*, 23 (1989), 215–226.

[32] MUSUVATHI, M., AND ENGLER, D. R. Model checking large network protocol implementations. In *NSDI* (2004).

[33] NetCheck. Accessed 2/27/2014, https://netcheck.poly.edu/.

[34] On Mac / BSD sockets returned by accept inherit the parent's FD flags. Accessed 2/27/2014, http://bugs.python.org/issue7995.

[35] Posix-omni-parser. Accessed 2/27/2014, https://github.com/ssavvides/posix-omni-parser.

[36] Prevent socket hijacking on OSes that don't prevent it by default (Windows). Accessed 2/27/2014, https://tahoe-lafs.org/trac/tahoe-lafs/ticket/870.

[37] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).

[38] RITCHEY, R., AND AMMANN, P. Using model checking to analyze network vulnerabilities. In *Security and Privacy* (2000).

[39] ROTEM-GAL-OZ, A. Fallacies of distributed computing explained. *Accessed 2/27/2014, URL http://www. rgoarchitects. com/Files/fallacies. pdf* (2006).

[40] SAMBASIVAN, R. R., ZHENG, A. X., DE ROSA, M., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing performance changes by comparing request flows. In *NSDI* (2011).

[41] Security: SO_EXCLUSIVEADDRUSE should be enabled when binding to ports on Windows. Accessed 2/27/2014, http://twistedmatrix.com/trac/ticket/4195.

[42] SO_REUSEADDR broken on Windows. Accessed 2/27/2014, http://bugs.sun.com/bugdatabase/viewc_bug.do?bug_id=4476378.

[43] SO_REUSEADDR doesn't have the same semantics on Windows as on Unix. Accessed 2/27/2014, http://bugs.python.org/issue2550.

[44] SUBHLOK, J., AND XU, Q. Automatic construction of coordinated performance skeletons. In *IPDPS* (2008).

[45] TCPSocket readline doesn't raise if the socket is close'd in another thread. Accessed 2/27/2014, http://bugs.ruby-lang.org/issues/4390.

[46] UDREA, O., LUMEZANU, C., AND FOSTER, J. Rule-based static analysis of network protocol implementations. *Information and Computation 206*, 2 (2008), 130–157.

[47] Using SO_REUSEADDR and SO_EXCLUSIVEADDRUSE. Accessed 2/27/2014, http://msdn.microsoft.com/en-us/library/ms740621%28VS.85%29.aspx.

[48] webrick: Ruby Standard Library Documentation. Accessed 2/27/2014, http://www.ruby-doc.org/stdlib-1.9.3/libdoc/webrick/rdoc/index.html.

[49] When using NAT interface on Windows host, guest can't receive UDP datagrams larger than 8 KB. Accessed 2/27/2014, https://www.virtualbox.org/ticket/12136.

[50] Windows ntpd should secure UDP 123 with SO_EXCLUSIVEADDRUSE. Accessed 2/27/2014, https://support.ntp.org/bugs/show_bug.cgi?id=1149.

[51] Wireless Implementation Testbed Laboratory (WITest) at NYU-Poly. Accessed 2/27/2014, http://witestlab.poly.edu.

[52] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *SOSP* (2009).

[53] XUE, H., DAUTENHAHN, N., AND KING, S. Using replicated execution for a more secure and reliable web browser. In *NDSS* (2012).

[54] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI* (2009).

[55] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS* (2010).

[56] ZHUANG, Y., BESCHASTNIKH, I., AND CAPPOS, J. NetCheck Test Cases: Input Traces and NetCheck Output. Tech. Rep. TR–CSE–2013–03, NYU Poly, 2013.

Exalt: Empowering Researchers to Evaluate Large-Scale Storage Systems

Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, Mike Dahlin
The University of Texas at Austin

Abstract

This paper presents Exalt, a library that gives back to researchers the ability to test the scalability of today's large storage systems. To that end, we introduce Tardis, a data representation scheme that allows data to be identified and efficiently compressed even at low-level storage layers that are not aware of the semantics and formatting used by higher levels of the system. This compression enables a high degree of node colocation, which makes it possible to run large-scale experiments on as few as a hundred machines. Our experience with HDFS and HBase shows that, by allowing us to run the real system code at an unprecedented scale, Exalt can help identify scalability problems that are not observable at lower scales: in particular, Exalt helped us pinpoint and resolve issues in HDFS that improved its aggregate throughput by an order of magnitude.

1 Introduction

This paper presents Exalt, a library that gives back to researchers the ability to verify the scalability claims of today's large storage systems, which, ironically, have become hard to corroborate precisely because of the scale of these systems.

The advent of Big Data has strained the scalability of traditional storage systems, and several new architectures have been proposed to respond to this challenge [2–4, 7, 12, 13, 17, 22] by supporting up to hundreds of petabytes of storage and tens of thousands of storage nodes. Testing systems at such scale, however, requires access to tens of thousands of machines and at least as many disks, and few researchers have access to resources that plentiful: the rest of us have to design systems that are supposed to operate at a scale much larger than the infrastructure available to test them. Nor are such resource limitations affecting only academia: even industrial researchers who are within reach of clusters of the necessary size may not be able to reserve them for large scale experiments, since these clusters are a primary source of revenue.

These limitations are typically sidestepped in one of two ways. The first is to run experiments on a medium-sized cluster (100-200 machines) and extrapolate the results to larger scales. While this may work reasonably well in some cases, the fundamental assumption on which it rests—that resource consumption increases linearly with the load and the number of machines in the system—does not always hold, as we show in Section 2. To make matters worse, sources of non-linear growth are sometimes hard or impossible to observe in small deployments. For example, the time needed to add a new block to an HDFS file [22] increases with the file's size, but it is only after that size has grown beyond what is likely to be observable in small deployments that the slowdown becomes a limiting factor for the system's performance.

The second common approach for predicting the behavior of large-scale systems is simulation [18, 24, 26]. Unfortunately, the results of a simulation are only as accurate as the model on which the simulation relies; as systems grow in size and complexity, modeling them faithfully becomes prohibitive.

This paper proposes a third way: the Exalt library offers researchers the ability to test the scalability of a large-scale storage system by running its real code, but without requiring access to thousands of machines. The basic insight at the core of Exalt is that, in many large-scale experiments, how data is processed is not affected by the content of the data being written, but only by its size. Exalt leverages this freedom by virtualizing the data, while keeping the metadata intact to ensure that the system continues to function correctly. Specifically, Exalt clients write data in a specific format, *Tardis*, that has two key advantages. First, it allows Exalt to compress the behavior of the system in both space and time. Space compression is a powerful tool for performing large-scale experiments: for example, running 10,000 storage nodes on just 100 machines can bring to light previously unknown scalability bottlenecks in the metadata service. Since compressed

data takes much less time to write, compression in space can in turn result in compression in time: with the system running faster, bugs and performance issues can be discovered more rapidly.

The second key advantage of Tardis is that it addresses a fundamental challenge in virtualizing data: being able to distinguish data from metadata. While the content of the former is not important for the system to function correctly and can therefore be virtualized, the integrity of the latter is essential. This problem is particularly prominent in modern storage systems, which employ a two-layer architecture where the upper layer uses the lower layer as black-box storage: files written to the lower layer contain both data and metadata, which look indistinguishable to the lower layer. The need to ensure the integrity of the metadata is why approaches that virtualize data by altogether disposing of file contents (e.g. [1]) cannot be used in our context.

In summary, this paper makes the following contributions:

- We introduce Tardis, a data representation scheme that allows data to be identified and efficiently compressed even at lower-level storage layers that are not aware of the semantics and formatting used by higher levels of the system. Tardis provides transparent, lossless, computationally efficient compression of data and achieves high compression ratios.

- We present a methodology that utilizes Tardis to test the scalability and robustness of large-scale storage systems: our goal is not to predict every aspect of the performance of such systems (e.g. their power consumption) but, more modestly, to identify scalability problems. Our approach has a "Truman-show" [25] feel: the part of the system whose scalability is being tested processes real data and interacts with the rest of the system as it would in a true large-scale deployment, while the rest of the system uses Tardis to compress data and achieve high degrees of colocation, thereby emulating the behavior of a large number of nodes.

- We present our experience using Exalt, a library that implements Tardis and uses our methodology to identify scalability issues in large-scale storage systems. Using Exalt we found and fixed several such issues in two mature storage systems: HDFS [22] and HBase [2]. All the problems we identified manifest when the scale of the system becomes larger than a typical research cluster. In the case of HDFS, resolving these problems resulted in an order of magnitude improvement of the aggregate system throughput. Our ability to identify these issues was not, for better or worse, due to a prior deep understanding, but rather to the opportunity offered by Exalt to test them at an unprecedented scale.

The rest of the paper is organized as follows. Section 2 discusses the common practices for testing the scalability of large systems. Section 3 introduces the Tardis data representation scheme and Section 4 describes how it can be used to identify scalability problems in large-scale systems. Section 5 reviews the assumptions of Exalt and discusses its applicability in various contexts. Section 6 presents our experience using Exalt to identify performance problems in two mature systems: HDFS and HBase. Section 7 discusses related work and Section 8 concludes the paper.

2 Testing for scalability: common practices

When faced with the challenge of running experiments on a system whose scale vastly exceeds their infrastructure, researchers typically resort to one of two options: they either run the system at the largest scale they can afford and try to extrapolate their results, or they explicitly forgo running certain components of the system, substituting them with stubs that, ideally, maintain the interactions of the original components with the rest of the system, but are simpler and less resource-demanding to run. We discuss both options, and why they are not well-suited for performing scalability tests on large-scale systems.

2.1 Extrapolation

A common approach to estimate the behavior of systems that are too big to test is to run them at a small or medium scale and then to extrapolate, based on those results, how they will behave at a large scale. For example, if the CPU utilization of a bottleneck node is 10% in a 100-node experiment, extrapolation would lead one to estimate that the system will scale to about 1,000 nodes. While attractive for its simplicity, this approach has several drawbacks that make it inaccurate in practice.

First, extrapolation is based on the assumption that resource usage grows linearly with the scale of the system. However, because of design choices and implementation issues, this assumption is frequently violated in practice. For example, HDFS uses an array to maintain a sorted list of files within a directory. Using an array causes insertion to be an $O(N)$ operation, where N is the number of files in the directory. As more files are added to the directory, insertion becomes increasingly expensive: indeed, the cost of adding N files to a directory is $O(N^2)$. Note that a more efficient directory implementation (e.g. a sorted tree map) does not restore linear growth in resource usage, but simply reduces the growth rate to $O(N \cdot \log N)$. In general, once the load on the system is not linear, accurate

extrapolation becomes much harder, especially because, as we have seen, the system's performance may depend on the details of the implementation.

A second, more subtle drawback of extrapolation is that at small scales some important behaviors can easily escape notice. Consider again the above example of a workload of $O(N^2)$ complexity: as long as the value of N is low, the potential scalability bottleneck remains largely inconspicuous. To exacerbate the problem, measuring resource utilization is an inherently noisy process. For example, observing that a Java process uses 100 MB of memory does not, by itself, indicate how much memory is being used by the data structures of that process. Answering that question requires accurate information about the amount of memory used internally by the JVM, the amount of non-garbage-collected memory, etc. The uncertainty added by measurement noise is significantly more prominent at lower scales, where resource utilization is low.

The final drawback, which is closely related to the previous one, is that extrapolation cannot be used to predict behaviors that are only triggered when some resource utilization reaches a certain threshold. For example, HDFS has a blocking disk-scanning procedure that becomes increasingly expensive as the system grows in size. Beyond a certain size, running the procedure causes the corresponding DataNode to start missing heartbeats, which in turn can cause it to be evicted and force all its data to be re-replicated, with serious performance repercussions.

2.2 Using stubs

Another technique for predicting the performance of a system too big to test is to emulate, rather than actually run, some of its components. The emulated components are implemented as stubs, running either locally or remotely. For this approach to be successful, the stubs should be simple to implement and require much more modest resources than the original components they stand in for; at the same time, they should be able to correctly exercise the rest of the system, allowing it to be stress-tested at scale using relatively modest resources.

While attractive in theory, the promises of emulation are often elusive in practice: reproducing accurately the behavior of a non-trivial real system component is hard, and in the process the stub component can end up being almost as complex as the real one, defeating its purpose.

We faced this challenge first-hand when trying to test the scalability of the HDFS NameNode using stub DataNodes. Our goal was to create a large number of stub DataNodes and use them to stress-test the NameNode. Our first attempt did not involve the DataNodes in the protocol at all; to create files and add blocks to them, clients simply invoked `createFile` and `addBlock` at the NameNode. However, the system did not work, since the NameNode expects the DataNodes to confirm the receipt of each block. We therefore modified our clients to notify the stub DataNodes, so they could in turn appropriately notify the NameNode. This did not work, either: the NameNode, we discovered, also expects each DataNode to periodically report the list of blocks it stores on disk. After several frustrating iterations, we eventually came to realize that emulating the correct behavior of DataNodes would have required us to reimplement the full HDFS protocol, including all inter-DataNode communication, local bookkeeping, etc.

3 Compressing data with Tardis

Our approach is based on a simple intuition: for the purposes of testing the scalability of large-scale storage systems, it is typically the size of the data being written that matters, not its actual content. We are then free to *choose* what data clients write during our tests: our work explores the opportunities that this freedom affords.

Specifically, our approach is to design a data format that achieves fast and efficient compression and decompression. As we discuss in Section 3.3, using compressed data lets us colocate multiple nodes on the same machine, which in turn enables running large-scale experiments on a small infrastructure.

Before presenting Tardis, our compression scheme, we set forth the requirements that it must fulfill.

3.1 Compression scheme requirements

The scheme must be lossless. While compression can reduce resource usage and allow node colocation, the ability to recreate the original data is essential. Modern large-scale storage systems typically use a two-layer architecture, where the upper layer uses the lower as a black-box storage [2–4]. What appears like generic data to the lower storage layer may actually be metadata necessary for the correct functioning of the upper layer; it is critical that none of this metadata be lost.

The scheme must achieve a high compression ratio. The motivation for this requirement is straightforward, since the compression ratio directly affects the amount of colocation we can achieve.

The scheme must be computationally efficient. As a counterexample, consider a straw-man scheme in which clients simply write sequences of 0's. This scheme offers obvious opportunities for significant compression; however, if it is possible for the system to interleave client data with metadata, the compression algorithm would need to scan all the input bytes to determine where the sequence of 0's begins and where it ends. The disk and network bottlenecks would have been removed, but at the expense

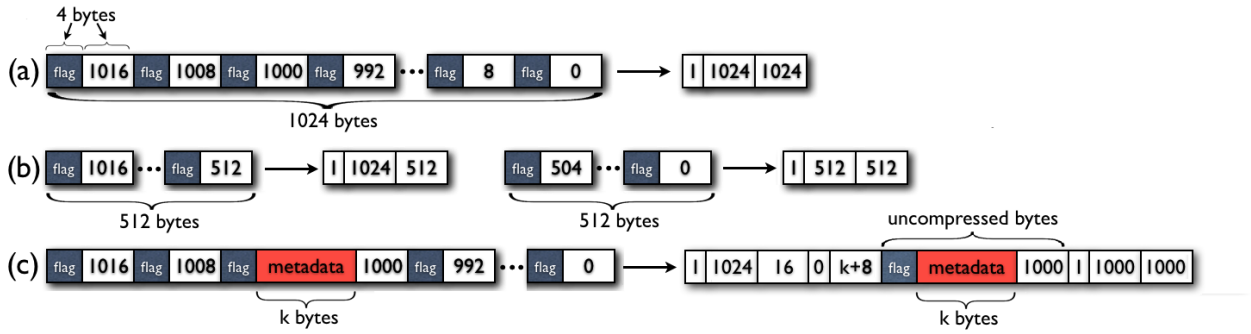


Figure 1: Examples of the Tardis format in compressed and uncompressed form.

of introducing a CPU bottleneck, severely limiting the scalability of this scheme.

Data chunks should be independently compressible. Modern storage systems do not necessarily store data as a single unit, but instead split it into multiple, separately stored chunks, which must be independently compressible. Meeting this requirement is challenging, however, since a client in general has no control over how data is divided into chunks. For example, in HBase the procedure of splitting data into chunks depends on a nondeterministic race between multiple threads.

3.2 Tardis compression

This paper introduces a novel compression scheme, called *Tardis*, that satisfies the above requirements. Tardis consists of a data format and an algorithm for compressing and decompressing the data. Intuitively, Tardis aims to achieve the following two complementary goals. When no metadata is inserted in the middle of the data, the compression algorithm should be able to compress the entire data after scanning only a small fraction of it. Otherwise, the compression algorithm should be able to quickly identify the location of the inserted metadata.

Data format Clients write data as a series of `<flag>` `<marker>` entries, where `<flag>` is a predefined byte sequence that does not appear in the system metadata, and `<marker>` denotes the number of remaining bytes in the data. For example, using a 4-byte flag and 4-byte markers, 1 KB of data would be formatted as:

```
<flag>1016<flag>1008...<flag>8<flag>0
```

In this example, the first marker denotes that there are 1016 bytes remaining in the sequence, since the (first) flag and the marker itself are 4 bytes each. Of course, the size of flags and markers need not be the same: our prototype uses 8-byte flags and 4-byte markers.

Compressed data format Given a byte sequence in the above format, the compression algorithm would simply need to return its length. However, to enable data chunks to be independently compressible, the algorithm actually returns two numbers: the starting byte of the sequence as well as its length. In the above example (also illustrated in Figure 1a), if the entire 1 KB of data were being compressed, the result would be the pair (1024,1024). If, however, the data were split into two chunks of 512 bytes each (Figure 1b), the first chunk would be compressed as (1024,512) and the second as (512,512).

As we discussed above, in modern storage systems data and metadata are frequently stored together. Figure 1c shows an example where metadata is inserted in the middle of a Tardis sequence. In this case, the metadata splits the original sequence into two subsequences, of length 20 and 1004, respectively. Ideally, we would like to compress each of these sequences separately, leaving the metadata uncompressed. However, since in this case the metadata is inserted in the middle of a flag-marker pair, we simply leave these 8 bytes—the flag and the corresponding marker—uncompressed.[1] This shortens the first subsequence to a length of 16 and the second subsequence to 1000. Note that even if the metadata were not aligned with the flags and markers, the result would be the same: only the flag-marker pair that is split by the metadata is left uncompressed and the rest of the data is compressed as two separate subsequences.

To distinguish between compressed and uncompressed data during decompression, an uncompressed sequence is preceded by a 0 and a 4-byte integer denoting its length, while a compressed sequence is preceded by a 1.

Compression Figure 2 shows the pseudocode for the Tardis compression algorithm. The main function, `TardisCompress`, iteratively calls the `FindSubsequence` function until all input data has been consumed. When `FindSubsequence` returns a new subsequence (line 7), the main function appends the appropriate bytes to the compressed data buffer. We detect the presence of metadata between two subsequences by checking whether the starting position of the new subsequence (*pos*) is after the end of the previous subsequence (*index*). If so, we append a 0 to denote the beginning of an uncompressed sequence,

[1] It is actually possible to include the flag in the compressed sequence, but we omit this optimization for simplicity of presentation.

```
1   #define unit_size = flag_size + marker_size

3   (compressed_data) TardisCompress(data)
4       result = empty buffer
5       index = 0
6       while index < data.len
7           (pos,marker,len)=FindSubsequence(data,index)
8           if pos == -1
9               result.AppendMeta(data,index,data.len-index)
10              return result
11          else
12              if pos > index
13                  result.AppendMeta(data,index,pos-index)
14              result.AppendTardis(data,pos,marker,len)
15              index = pos+len
16      return result

18  (pos,marker,length) FindSubsequence(data,startIndex)
19      (pos,marker) = ScanForFlag(data,startIndex)
20      if pos == -1
21          return (-1,-1,-1)
22      lastMarker = data.len-(unit_size+(data.len-
            position)%unit_size)
23      target = min(pos+marker,lastMarker)
24      marker2 = BinarySearch in data from pos to target
25          for the rightmost flag-marker pair such that:
26          (pos2,marker2) = ScanForFlag(data,target) and
27          pos2 != -1 and pos2-pos == marker-marker2
28          return (pos, marker, marker-marker2+unit_size)
29      if no such marker2 is found
30          return (-1,-1,-1)

32  (position, marker) ScanForFlag(data, startIndex)
33      index = linearly search data for (flag,marker)
            starting at startIndex
34      if index >= 0
35          return (index, marker)
36      else
37          return (-1, -1)
```

Figure 2: Pseudocode for Tardis compression.

followed by the length of the metadata, and finally by the metadata itself, uncompressed (*AppendMeta*, line 13).

It is then time to add the new subsequence. To denote that what follows is compressed, we append a 1 before the compressed form of the Tardis subsequence (which, recall, consists of the starting point and length of the subsequence) (*AppendTardis*, line 14).

Function `FindSubsequence` is the core of the algorithm: its task is to identify a Tardis subsequence. Two factors complicate this task: the sequence may have been split into multiple chunks and metadata may have been inserted somewhere in the sequence. Given a starting index in the data, `FindSubsequence` first scans the data to find the first flag, indicating the start of a Tardis sequence, and reads the corresponding marker (line 19). Then, it checks whether some metadata has been added in the middle of this sequence. The check is simple: if no metadata is inserted between two markers with values A and B, then these markers should be placed $B-A$ bytes apart. The purpose of lines 22-23 is to determine which marker should serve as marker B. If the original sequence is not split across chunks, then B is marker 0, which should be m bytes after the first marker, where m is the value of the first marker. Otherwise, B is set to the last marker of the current chunk. If the difference between the values of markers B and A is indeed equal to the byte distance between the markers, the algorithm has found an uninterrupted Tardis subsequence. If that is not the case, the algorithm performs a binary search to find the rightmost flag-marker pair that satisfies the above condition, leveraging the fact that the values of the markers form a sorted sequence (lines 24-30).

In practice, the common case is very simple: as long as there is no metadata inserted in the byte sequence, the compression algorithm needs only to check the first and last number of the sequence. This allows Tardis to compress data much faster than off-the-shelf compression algorithms. For example, when compressing data chunks of 1 MB, Tardis is about 33,000 times faster than Gzip [11] and 2,300 times faster than the straw man compression scheme where client data consists only of 0's and the compression algorithm simply scans the data and compresses sequences of 0's into an integer denoting their length. Of course, the comparison to GZip is not apples-to-apples, since Gzip is a generic compression algorithm; what it does show, however, is that being able to choose the data format drastically reduces the CPU overhead of our approach.

Decompression The decompression algorithm is straightforward. Given a compressed sequence, it iterates through each sequence, whether compressed (preceded by a 1) or uncompressed (preceded by a 0 and the length of the sequence). Uncompressed sequences are copied without modification, while compressed sequences are expanded to their uncompressed form.

Choosing the flag To prevent portions of metadata from being accidentally compressed, the flag sequence should never appear in the metadata. If it did and, by unlucky coincidence, the length value following the fake flag pointed to another flag followed by a 0, all that sequence of bytes would be compressed. Although we could altogether eliminate this danger,[2] it seems unnecessary: Exalt is not intended for production use, and an accidental compression would simply require us to rerun the affected experiment. With a sufficiently large flag, the odds of a false positive can be driven arbitrarily low: our pragmatic approach was to choose as flag an 8-byte random sequence and take our chances. Our experiments have yet to produce a false positive.

3.3 Using compression to enable large-scale tests

Since we are attempting to run a large number of nodes on a much smaller number of machines, we will nec-

[2] It would suffice to escape the flag sequence in the metadata. However, this would require intrusive modifications to the server code, as all metadata insertions would need to be aware of the escaping logic.

essarily have to colocate multiple nodes on the same machine. However, such colocation will cause significant contention on the physical resources of the machine. Specifically, the disk- and memory capacity, and the disk- and network bandwidth available to each machine are typically enough to support only a single node, making straightforward colocation infeasible.

Data compression can help here: storing compressed data on disk decreases the disk capacity and bandwidth requirements of each node, as well as memory capacity and network bandwidth. Of course, data compression is not without cost; in this case, the cost is CPU utilization.

This tradeoff, however, is very attractive for storage systems, where CPU cycles are plentiful and bandwidth and storage capacity are typically the system's bottleneck. It also opens the door to emulating the behavior of storage systems too big to test using HPC computation clusters: indeed, as we will see in Section 6, our analysis of the scalability of HDFS/HBase has been performed by running Exalt on the Stampede high performance cluster at the Texas Advanced Computing Center (TACC) [23].

If data compression is used without colocation, it results in a system that is "compressed" in time, rather than space, since each write will take less time to complete. Running the system at an accelerated pace offers the potential of identifying bugs or performance problems much faster: Section 6.1.4 discusses a case where time compression allowed us to identify a problematic behavior about 100 times faster than a real deployment.

3.4 Implementation

Our implementation of Exalt performs data compression for three key resources: disk, network, and memory. Our goal is to be minimally intrusive. While in-memory compression does require minor modifications to the source code of the storage system being tested, we achieve fully transparent disk and network compression by using byte code instrumentation (BCI) to modify the relevant Java library classes (Socket, SocketInputStream, SocketOutputStream, SocketChannel for network compression; File, FileInputStream, FileOutputStream, RandomAccessFile, and FileChannel for on-disk compression).

File compression is more challenging than network compression because the file interface allows a user to partially update existing data. When that data is already compressed, updating it in place is not straightforward. A naive solution would be to decompress the existing data, update it, and compress it again. However, if the old and the newly compressed data have different sizes, all following data chunks would have to be moved. To address this problem, similarly to the Log-Structured File System (LFS) [20], we transform in-place update operations into append operations. This allows us to efficiently process in-place updates, with only a small bookkeeping overhead to keep track of the latest version of each range of bytes.

Memory compression In-memory data structures do not use a well-defined interface, such as the File or Socket abstraction used by the disk and network. As a result, transparently modifying these data structures to compress and decompress data at the application layer is very hard.[3] Instead, when the in-memory data needs to be compressed, we manually modify the source code of the system. Fortunately, this process is quite simple. One need only identify the data structures that hold the client data. When data is stored in the data structure, it is compressed; when data is retrieved from the data structure, it is decompressed. For example, compressing the in-memory key-value store of HBase required adding 71 lines of code across 4 files.

4 Finding scalability bottlenecks

Data compression gives us the ability to colocate multiple nodes on a single physical machine: in this section, we discuss how we can selectively use this ability to draw meaningful conclusions about the scalability of a large-scale storage system. We will view the system as a collection of *real* and *emulated* nodes. A real node runs the system's actual code and handles unmodified data. An emulated node still runs the system's actual code, but, as needed to support colocation, may (i) store compressed data on its disk, (ii) send compressed data over the network, when it communicates with other emulated nodes, and (iii) store compressed data in memory.

4.1 Exalt methodology

We use this combination of real and emulated nodes as a microscope of sorts that we can focus on a part of the system to identify performance bottlenecks. To ensure that the part of the system "under inspection" behaves exactly as it would in a real large-scale deployment, we leave the corresponding nodes real, while using emulated nodes for the rest of the system. This approach works particularly well at identifying performance issues at centralized components that can become a bottleneck as the scale of the system increases (e.g. HDFS NameNode, HBase Master). Section 6 discusses our experience using this technique to find scalability problems in real systems.

A downside of this methodology is that it may not discover scalability problems that arise at the nodes that are being emulated. To address this issue, after having stress-tested the part of the system under inspection by using the maximum amount of colocation for emulated nodes, we perform a new set of experiments where a small

[3] Transparent compression of in-memory data could be potentially implemented at the kernel level, but it would sacrifice portability.

subset of formerly emulated nodes are also run as real, while the rest is kept emulated. This hybrid configuration makes it possible to identify scalability problems also at nodes that are not under inspection, while maintaining a high degree of colocation, but it is not a panacea: for example, it is still unable to detect performance issues that only manifest when a large number of nodes that are not under inspection perform some collective action (e.g. system-wide recovery).

5 Limitations and applicability

Exalt relies on a number of assumptions to provide high-degrees of node colocation. This section reviews these assumptions and discusses which of them can be weakened to widen the applicability of our approach.

Exalt is primarily designed to evaluate I/O-intensive applications like distributed file systems storing large files [7, 22] or key-value stores with relatively large values [3, 17]. Applications that are not I/O-intensive or store small values can not benefit significantly from Tardis, as they gain little by compressing data. In Section 6.2 we explore in more detail how the size of the value in a key-value store affects the colocation ratio of Exalt.

Our current implementation of Exalt makes two additional assumptions: first, that the target application does not modify the data written by the clients, although it can split the data and insert metadata; and second, that experiments are not sensitive to the contents of the data, so that clients can operate with synthetic data.

While these assumptions hold for the systems we have so far applied Exalt to, they are not fundamentally required for Exalt to be applicable. We consider below some popular techniques that violate these assumptions and discuss how our implementation of Exalt can be modified to work in conjunction with them.

Encryption and erasure coding Both techniques involve encoding data into a different format, violating the assumption that client data is immutable. To handle these cases, Exalt would compress the data using Tardis *before* encoding it, and then add *filler bytes* as necessary to match the length of the (encoded) original data. Filler bytes would use the same format as Tardis (making them highly compressible), but with a different flag sequence (so that they can be distinguished from real data). When reading the data, Exalt would remove the filler bytes before performing decryption and then decompress the Tardis sequence to obtain the client data.

Deduplication Deduplication compares the contents of different data units (files, chunks, etc.) to eliminate duplicates and, by making execution dependent on the actual data, violates our second assumption. Indeed, deduplication schemes that directly compare the units' data are incompatible with Exalt. However, Exalt can still be applied to deduplication approaches that only compare hashes of data units. Exalt would first compute the hash of the client data and then replace the client data with data formatted using an extended version of Tardis, which inserts the hash of the data unit between the flag and the marker. The deduplication module could then use this hash directly to identify duplicate data units.

Compression If the system being tested already uses compression, it is in general not possible to use synthetic data at the clients, since the compression ratio depends on the actual data. If, however, compression is performed only at the client side, Exalt could apply a technique similar to the one used to handle encryption and erasure coding: the client would first compress the real data to determine its compressed size, then create synthetic (Tardis) data, compress it using Tardis' compression and finally append the right amount of filler bytes to match the length of the (compressed) real data.

Data sensitive applications Many applications use SQL-like languages for their queries. The execution of these queries depends on the data, since SQL predicates can be expressed as a function of the data. The rest of the data, which does not affect the processing of the queries, can be synthetic. The efficiency of Exalt in these cases depends on the ratio of sensitive to non-sensitive data.

6 Case studies

Exalt allows us to evaluate storage systems at an unprecedented scale. This section presents our experience applying Exalt to evaluate two real-world storage systems: the Hadoop Distributed File System (HDFS) and the HBase key-value store [2, 22]. We chose these systems for several reasons. First, not only they are popular in their own right, especially among researchers, but their architecture is representative of the majority of existing large-scale storage systems. Second, both systems are open-source, which allows us to perform code modifications where necessary (i.e. for in-memory compression). Finally, both systems have a large development community that has produced a mature and stable codebase. Despite the maturity of the code, we identified several performance issues that arise as the scale of the system increases. Our ability to diagnose these issues was not due to a prior deeper understanding of these systems, but simply to the ability to evaluate them at an unprecedented scale.

In our evaluation, we run HDFS and HBase at a scale about 100 times larger than the size of the infrastructure available to us. For example, one of our experiments uses 96 machines to run an HDFS cluster with 9600 DataNodes. Our experiments identify a number of performance problems that arise at such large scales. Some of these problems pertain to low-level implementation details, while others are due to high-level design choices.

For example, we find that storing many files on an HDFS directory causes file creations to that directory to become increasingly slow; and that keeping less than $\frac{3}{4}$ of the region data in the memory of an HBase region server causes its performance to degrade precipitously. Using Exalt, we were able to identify and fix many of these problems, improving as a result the aggregate HDFS throughput by an order of magnitude.

Unfortunately, we have not yet been able to validate our results by running the actual systems at a large scale: after all, it is the very reason that we do not have access to such plentiful resources that has motivated our work in the first place. The largest validation we have performed involved running HDFS/HBase on 1,500 nodes of the Stampede cluster at the Texas Advanced Computing Center [23]: while our results confirm the prediction of Exalt for that configuration, the scale of the system is still too small to exhibit even the first of the scalability issues identifies by Exalt. Our current confidence in Exalt's effectiveness stems from two sources. First, for each problem that Exalt has identified, we have traced the cause of the problem in the source code, fixed it, and run the modified system to confirm that its performance has been improved. Second, some of our findings have been confirmed by engineers at Facebook, among the few who have access to a large-scale deployment of HDFS [6].

Most of our experiments were performed on the Stampede cluster at TACC, whose machines have 16 cores, 32 GB of memory, but only 80 GB of local disk storage. Since our access to TACC was limited, we ran some of our experiments on three local machines with 16 cores, 64 GB memory and 10 1 TB disks each. These machines were used to test the capacity limitations of individual storage nodes.

6.1 Case study: HDFS

HDFS [22] is an open source implementation of the Google File System (GFS) [7]. Each HDFS cluster contains a single NameNode that stores the file system namespace information and several DataNodes that store the file contents. Each file is split into multiple blocks and each block is stored on three DataNodes. When a client creates a file or adds a block to an existing file, it first contacts the NameNode, which responds with a list of the DataNodes that will store the new block. The client can then directly write the block contents to these DataNodes.

We mainly focus on write workloads since they are more likely to cause scalability problems. Unless otherwise specified, in our experiments each client creates a file in its own directory, writes 192MB of data to it (as suggested by the HDFS developers in their white paper on how to test HDFS' scalability [21]), closes the file, and then starts a new file. This workload achieves the highest scalability among all workloads that we tried; Section 6.1.3 describes the performance problems caused by other workloads. We use a block size of 128 MB and the default 3-way replication (again, as suggested in [21]). Unless otherwise specified, we run DataNodes and clients in emulated mode while the NameNode runs in real mode.

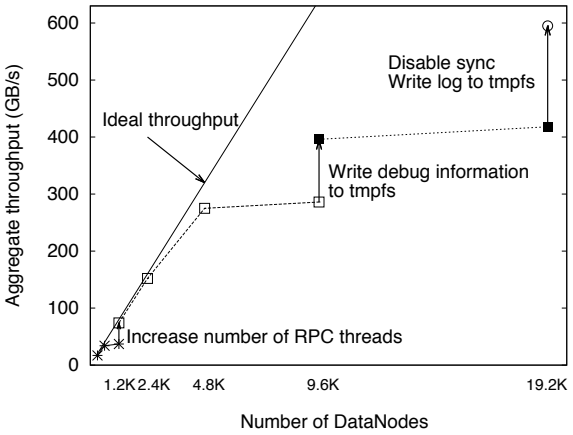


Figure 3: HDFS throughput scalability.

For the above workload, Tardis achieves a compression ratio of over 500, but in practice the degree of colocation is limited by CPU utilization: we colocate 100 DataNodes on one machine and achieve an effective write bandwidth of 10 GB/sec on a disk with 100 MB/sec physical bandwidth. For experiments with modest storage capacity requirements, we can increase the write bandwidth to 20 GB/sec by writing to tmpfs, an in-memory file system. Our largest configuration experiment uses 192 server machines, to emulate an HDFS cluster with 19,200 DataNodes.

6.1.1 HDFS throughput scalability

In some respect, the result of our experiments to test the scalability of HDFS is not surprising: the bottleneck of the system is the centralized NameNode. What is perhaps surprising is that, thanks to Exalt, we were able to increase the system throughput by an order of magnitude without changing the architecture of the system.

Figure 3 reports the results of our experiments. On the x-axis we increase the number of DataNodes and on the y-axis we plot the aggregate throughput of the system, as observed by the clients. The vertical arrows represent the process of fixing an issue that was limiting the system throughput. When an issue is fixed, we rerun the experiment for the same number of DataNodes, to verify that the system indeed achieves a higher throughput. For reference, we also plot a straight line that shows the ideal throughput achievable by a perfectly scalable system that leverages the full bandwidth of all disks (100 MB/s).

Our first experiment shows that the original HDFS sys-

Memory size	1GB	2GB	4GB	8GB
HDFS Capacity	1.15PB	2.35PB	4.76PB	9.49PB

Figure 4: HDFS space scalability as a function of NameNode memory size.

tem quickly saturates at around 37 GB/s. We discovered through profiling that the default number of RPC threads at the NameNode was limiting the achievable throughput; increasing the number of RPC threads from 10 to 256 allows the NameNode to achieve much higher throughput.

After fixing the first issue, the system saturates at around 286 GB/s. Further profiling showed that the I/O accesses at the NameNode were becoming the system bottleneck. More specifically, the NameNode debug information was being stored on the same disk as its log file. This prevented both files from sequentially accessing the disk, thereby introducing a large number of `seek` calls that reduced performance. Our solution was to write the debug information to tmpfs instead, thereby making sure that the NameNode log file was sequentially accessing the disk. Alternatively, one could store the debug information on another disk, if one were available.

Applying the second fix increases the system throughput to 418 GB/s, at which point the system again becomes saturated. This finding is consistent with the scalability assessment of the HDFS developers that "assuming each client has a write throughput of 40 MB/s, the system can support no more than 10,000 clients", which corresponds to an aggregate throughput of 400 GB/s. While this assessment was obtained using extrapolation, we consider it reasonably accurate since it is based on a large deployment of 4,000 nodes.

Since we suspected disk I/O to be the system bottleneck at this point, we performed a final experiment in which disk `sync` is disabled and the NameNode writes all logs to tmpfs. The purpose of this experiment is to project the scalability of the system in the presence of a fast storage medium (e.g. NVRAM,SSD). In this configuration, the system throughput increases by a further 50%, to a maximum throughput of 595 GB/s.

Of course, we do not claim that Exalt's throughput predictions are perfectly accurate; on the contrary, we acknowledge the limitations of running a system whose resources are partially emulated. Nonetheless, the benefits of Exalt are clear: it allowed us to test the system's real code and identify and resolve performance issues at a scale that would have otherwise remained the sole province of a few large companies.

6.1.2 HDFS capacity

The capacity of an HDFS cluster is limited by the amount of memory available to the NameNode. In this experiment, we try to measure how much memory the NameNode needs per 1 PB of HDFS storage space. Figure 4 shows that the capacity of HDFS grows linearly with the amount of memory at the NameNode. In particular, 1 GB of NameNode memory can support about 1.2 PB of raw HDFS space (400 TB of data, since blocks are 3-way replicated). This result is close to the estimation of HDFS developers: "1 GB of metadata $\approx$ 1 PB of physical storage" [21].

Using Exalt allows us to perform this experiment using only 16 TB of disk storage, while a real deployment would require a total of 10 PB of disk storage.

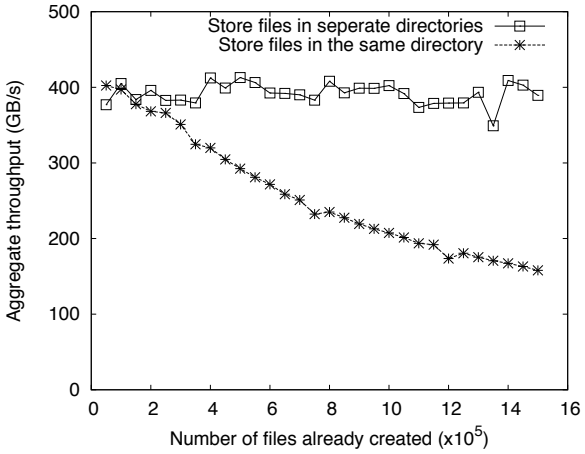


Figure 5: HDFS throughput degradation as the size of directories increases.

6.1.3 Performance degradation in HDFS

The above experiments use a workload that provides high scalability. Other workloads are not as accommodating. We evaluate two such workloads that can drastically degrade the HDFS performance.

In the first workload, all clients create files in the same directory. As shown in Figure 5, the aggregate system throughput steadily decreases as more files are created. Further profiling allowed us to identify the cause of this behavior in the source code: the NameNode uses an ArrayList data structure to maintain an alphabetically sorted list of the files inside a directory. Adding an element to a sorted array is an O(N) operation, since it requires a suffix of the sorted array to be shifted by one position. Therefore, the bigger the directory, the longer it takes to add a file to it. As a double check, we verified that, if we limit the number of files written to each directory, creating more files does not cause a performance degradation.

In the second workload, one client creates a file and keeps appending data to it. As shown in Figure 6, once the file grows sufficiently large, the aggregate system throughput decreases steadily. Note that in this experiment there are only a few clients and the system is not fully sat-

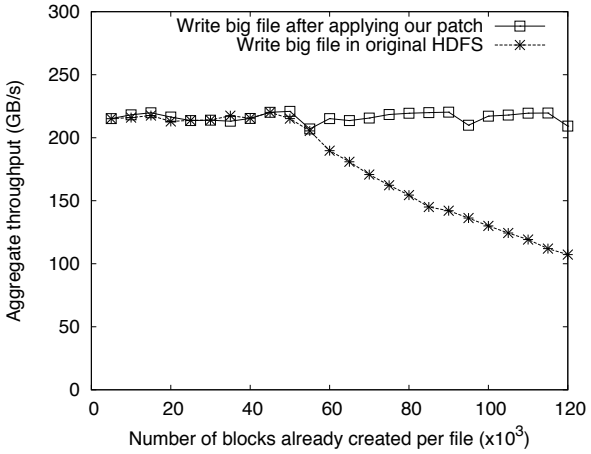


Figure 6: HDFS throughput degradation as the size of files increases.

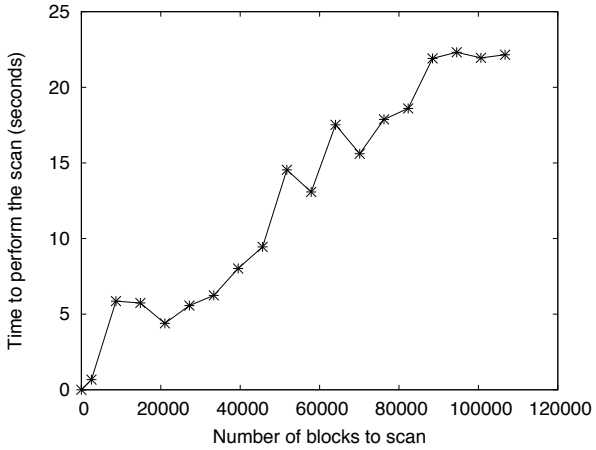


Figure 7: Time of the block-scan procedure on a DataNode, as the number of blocks increases.

urated, which accounts for the fact that the aggregate system throughput is lower than in the previous experiment. Profiling led us to the cause of the problem: before the NameNode creates a new block for a file, it needs to calculate the file's length. It does this by scanning all existing blocks and computing the sum of the lengths of all blocks. This, too, is an O(N) operation. We fixed this problem by adding a length field to each file and updating the field when a block is added or updated. As Figure 6 shows, after applying our fix the system throughput no longer decreases as the files grow in size.

As before, Exalt allows us to identify these performance issues without requiring access to a large amount of disk storage. Running this experiment in a real deployment would require 900 TB of disk storage; with Exalt, we only need 1.5 TB.

6.1.4 DataNode scalability

As disk capacities increase every year, and most HDFS deployments use multiple disks per DataNode, it is important for the DataNode's performance to not decrease as more storage capacity is added to it. While running HDFS in hybrid mode—keeping some DataNodes real— we observed uncommonly high latencies for some requests. Our profiling indicated that the source of the problem was a disk scan that the DataNode periodically performs on all its blocks. Figure 7 shows that the time a real node takes to perform this scan increases linearly with the number of blocks stored on the disk. Unfortunately, this scan is a blocking operation, preventing write requests and heartbeats from being sent or received. As the duration of this scan becomes longer, it can have serious performance consequences, including timeouts at the clients or even missed heartbeats, which would cause unnecessary re-replication of the DataNode's data. This issue is confirmed by Facebook engineers; to address it, they modified HDFS to allow the block scan to be performed in parallel with heartbeats and write requests [6].

While reproducing this problem is easy, triggering it in a real deployment would require 8 TB of disk storage on a single DataNode; using Exalt, we triggered this problem using an 80 GB disk. After identifying the problem, we reproduced it on a real DataNode with 8 TB of disk storage (Figure 7).

Note that although it could be triggered with only a few machines, this problem would be hard to identify and tedious to reproduce during debugging, since it would take at least a few hours for the latency increase to be observable. Exalt's time compression helps in this case. If emulated nodes have exclusive access to a machine's resources, the system works at an accelerated speed: in this example, the problem would manifest itself in a matter of minutes.

6.2 HBase

HBase [2] is a distributed key-value store built upon HDFS. The basic data unit of HBase is a region, which corresponds to a continuous key range in a table. An HBase cluster includes a Master, responsible for assigning regions to different region servers. Client requests to a specific region are directed to the corresponding region server. The region server processes write requests by logging them to HDFS while also keeping them in a memory buffer called *memcache*. When the size of the memcache exceeds a threshold, the region server writes the whole memcache into a checkpoint file on HDFS, so that it can garbage collect the previous log files. A checkpoint is also taken if the total memory usage across all regions exceeds some limit; in this case, a region server checkpoints the region with the largest memcache. When necessary to

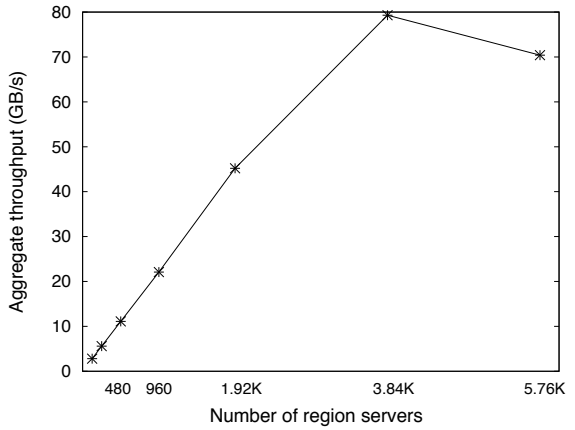


Figure 8: HBase throughput scalability.

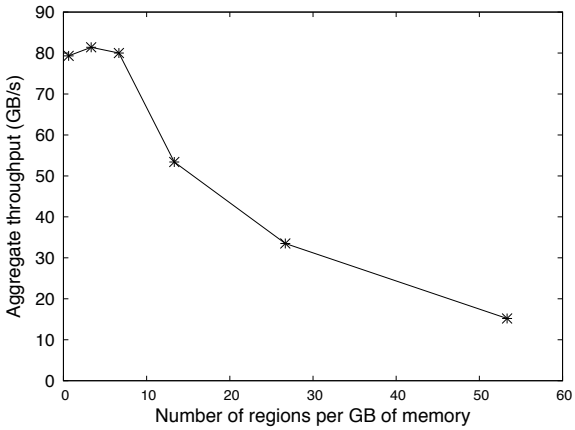


Figure 9: HBase aggregate throughput as the number of regions per GB of memory changes.

free up space, the region server performs *compaction* to merge several checkpoints. In essence, a region server transforms the random access patterns of a key-value store into the append-only interface of HDFS. When a region grows large, HBase splits that region into two for load balancing; conversely, if two adjacent regions are too small, they are merged into one. Apart from the Master and the region servers, an HBase cluster incorporates a ZooKeeper ensemble that performs lease management.

We evaluate HBase using a simple workload that can achieve a high throughput: we create enough regions so that each region server stores about 10 regions. We start multiple clients that randomly write key-value pairs to those regions. The key size is 4 bytes and the value size is 1 MB. To measure the maximum achievable throughput, we disable split, merge, and compaction operations—to ensure that split and merge operations do not occur, we limit the number of key-value pairs written to a region. We plan to study the effects of split, merge, and compaction in the future.

In our experiments, we keep the HBase Master, HDFS NameNode and ZooKeeper cluster real, while all DataNodes and region servers are emulated. In each experiment we assign 500 MB of physical memory to region servers. However, we perform in-memory compression, which effectively increases each region server's memory to 16 GB.

Figure 8 demonstrates the throughput scalability of HBase as the number of available region servers increases. Note that the raw throughput of HBase is much lower than that of HDFS (see Figure 3). This is due to two reasons: first, HBase needs to write data twice to HDFS—once for logging and once for checkpointing. Second, region servers are relatively more CPU-intensive than DataNodes and therefore can not benefit as much from colocating multiple nodes on the same machine.

HBase can achieve a maximum write throughput of about 80 GB/s. Considering that HBase writes data twice, this translates to a 160 GB/s throughput at the HDFS layer, which is about 40% of the maximum throughput achievable by HDFS. Our profiling shows that the `sync` calls to disk at the HDFS NameNode are still the bottleneck of the system. The reason for this 60% performance loss is that the region servers perform many additional directory operations, other than simply creating and closing files. For example, when a log file is garbage-collected, the region server first moves it to an "old log" directory as a backup and only deletes it after some time has elapsed.

In Figure 8 each region server has 16 GB of memory and holds 10 regions: since the default maximum size of a region is 200MB, all data can be cached in memory. Our next experiment evaluates how the performance of HBase is affected when we decrease the memory size per region. As shown in Figure 9, HBase throughput drops significantly when the number of regions per GB of memory exceeds 7, which translates to about 150 MB of memory per region. In other words, in order for HBase to work efficiently in a large-scale deployment, each region server must be equipped with a considerably large amount of memory: enough to hold at least $\frac{3}{4}$ of its on-disk data. The reason for this performance drop is that region servers flush their regions to HDFS files when their memory usage exceeds a certain threshold. If the number of regions per GB of memory is high, this will create a large number of small files on HDFS, which stresses the HDFS NameNode. Resolving this problem requires a significant redesign of HBase, which is beyond the scope of this paper. Note that this performance drop is only observed at large scales, since small deployments can not generate enough load to saturate the HDFS NameNode.

Our last experiment explores the effect of writing small values on the colocation ratio achievable in Exalt (Figure 10). Not surprisingly, Exalt achieves high colocation ratios when the value sizes are large (around 500 KB),

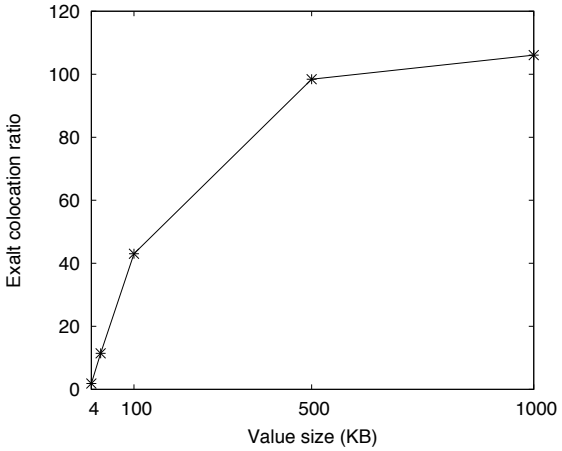


Figure 10: Colocation ratio of Exalt.

but does not fare equally well for small values. It is worth noting that the achievable colocation ratio for a given workload is not infinite; eventually CPU utilization becomes the bottleneck of the system. For HBase, this happens at a colocation ratio of about 110.

7 Related work

As we mentioned earlier, two common approaches to evaluating the scalability of large storage systems are using extrapolation and stub components. For example, extrapolation is used, among others, in RAMCloud [19], Spanner [12], and Salus [27], while the stub approach is used in HDFS [21, 22]. Section 2 discusses these approaches in detail, so we do not discuss them further here.

Several tools have been proposed to address the gap between the size of the experiments that researchers would like to run and the resources available to them.

In DieCast [9] this experimental gap is addressed using time dilation [10]. DieCast runs multiple processes inside virtual machines on a single host and slows down each process by a constant factor. It compensates for this slow-down by multiplying the measured throughput by the same factor. DieCast can achieve some degree of colocation when CPU utilization is the bottleneck, but does nothing to reduce the large amount of disk space necessary to evaluate large-scale storage systems.

The system that comes closer to addressing the experimental gap for storage systems is David [1]. David leverages the observation that to evaluate a local file system it is not necessary to store the actual data. Thus, David only stores the file system's metadata: the data is simply discarded. This technique allows David to evaluate local file systems of much larger size than that of the local disk on which they are run. Unfortunately, this approach cannot be easily applied to distributed storage services. For example, when users write a key-value pair to HBase, the region server adds a timestamp and a region identifier to the write request and stores this metadata, together with the users' data, on the local file system of an HDFS DataNode. Since data and metadata look indistinguishable to the HDFS layer, David would discard metadata critical for the correct operation of the system.

Memulator [8] emulates nonexistent storage components by storing data in memory and accurately predicting how long each operation takes. Its purpose is to test the behavior of the system on devices that the researchers do not have access to. Unlike Exalt, it does not save any resource usage, which makes it not applicable to our goal.

Finally, simulation is a technique used by several systems to evaluate the performance of large-scale deployments. The approaches vary from disk simulation [24], network simulation [15, 18], to simulation of large-scale P2P systems [26]. A well-known drawback of simulation is that its results are only as good as its model of how the system works. Unfortunately, as systems grow in complexity, coming up with a model that accurately captures all their features becomes prohibitively hard.

There exist several compression algorithms [5, 14, 16, 28, 29] one may consider using in our context. However, all these algorithms are designed to be general-purpose and as such they need to scan all the input bytes. Tardis, on the other hand, owes its efficiency largely to the fact that it does not have to scan most of the input bytes.

8 Conclusion

Exalt is a library that gives back to researchers the ability to evaluate the scalability of large storage systems. Exalt is based on the Tardis compression scheme, which leverages a specific data format to achieve efficient compression and high degrees of colocation, which in turn allows researchers to perform large-scale experiments on as few as one hundred machines. We have used Exalt to identify several performance problems in HDFS and HBase. Fixing these problems allowed the system to significantly increase its maximum achievable throughput. We plan to use Exalt to evaluate the performance of more large-scale systems (e.g. Cassandra [13]).

Finally, we plan to further explore the relationship between space and time compression to quickly diagnose problems that might otherwise require several days or weeks of testing.

Acknowledgements

We thank Mark Silberstein for his insight during early discussions of this work, our shepherd Miguel Castro and the anonymous reviewers for their helpful comments, and TACC for providing access to the Stampede cluster. This work was supported in part by NSF grant CiC-FRCC-1048269 and by a Google Graduate Fellowship.

References

[1] N. Agrawal, L. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Emulating Goliath Storage Systems with David. In *FAST*, 2011.

[2] Apache HBASE. `http://hbase.apache.org/`.

[3] B. Calder et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[4] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[5] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 1984.

[6] Private communication with Facebook engineers Siying Dong and Liyin Tang.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, pages 29–43. ACM Press, 2003.

[8] J. L. Griffin, J. S. andSteven W. Schlosser, J. S. Bucy, and G. R. GangerNitin. Timing-accurate Storage Emulation. In *FAST*, 2002.

[9] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *NSDI*, 2008.

[10] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *NSDI*, 2006.

[11] Gzip. http://www.gzip.org/.

[12] J. Corbett et al. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[13] A. Lakshman and P. Malik. Cassandra – A decentralized structured storage system. In *LADIS*, 2009.

[14] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, Nov. 1990.

[15] Network Emulation with the NS Simulator. http://www.isi.edu/nsnam/ns/ns-emulation.html.

[16] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7(1):67–82, July 1997.

[17] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat Datacenter Storage. In *OSDI*, 2012.

[18] NS-2. `http://www.isi.edu/nsnam/ns/`.

[19] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[20] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, Feb. 1992.

[21] K. Shvachko. HDFS scalability: the limits to growth. http://c59951.r51.cf2.rackcdn.com/5424-1908-shvachko.pdf.

[22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[23] Texas Advanced Computing Cener (TACC). https://www.tacc.utexas.edu/.

[24] The DiskSim Simulation Environment. http://www.pdl.cmu.edu/DiskSim/.

[25] The Truman Show. http://www.imdb.com/title/tt0120382/.

[26] K. Wang. Exploring the Design Tradeoffs for Exascale System Services through Simulation. http://datasys.cs.iit.edu/ kewang/documents/presentation_1.pptx.

[27] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus scalable block store. In *NSDI*, 2013.

[28] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

[29] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, Sept. 1978.

ipShield: A Framework For Enforcing Context-Aware Privacy

Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan,
Yasser Shoukry, Matt Millar, Mani Srivastava
University of California, Los Angeles

Abstract

Smart phones are used to collect and share personal data with untrustworthy third-party apps, often leading to data misuse and privacy violations. Unfortunately, state-of-the-art privacy mechanisms on Android provide inadequate access control and do not address the vulnerabilities that arise due to unmediated access to so-called *innocuous* sensors on these phones. We present ipShield, a framework that provides users with greater control over their resources at runtime. ipShield performs *monitoring* of every sensor accessed by an app and uses this information to perform privacy *risk assessment*. The risks are conveyed to the user as a list of possible inferences that can be drawn using the shared sensor data. Based on user-configured lists of allowed and private inferences, a *recommendation* consisting of binary privacy actions on individual sensors is generated. Finally, users are provided with options to override the recommended actions and manually configure context-aware *fine-grained* privacy rules. We implemented ipShield by modifying the AOSP on a Nexus 4 phone. Our evaluation indicates that running ipShield incurs negligible CPU and memory overhead and only a small reduction in battery life.

1 Introduction

Smartphones have evolved from mere communication devices into sensing platforms supporting a sprawling ecosystem of apps which thrive on the continuous and unobtrusive collection of personal sensory data. This data is often used by the apps to draw inferences about our personal, social, work and even physiological spaces [10, 41, 16, 53, 9, 58, 49, 51] often under the pretext of providing personalized experiences and customized recommendations. However, not all app developers are equally trustworthy, and this coupled with user naïveté leads to data misuse and privacy concerns.

To safeguard user privacy, Android requires developers to specify the permissions needed by their apps. At install time, a user can either grant access to all the requested resources or opt to not use the app at all. But despite these provisions, cases of privacy violations by third-party apps are rampant [33, 6, 55]. We observe multiple problems with the current privacy mechanisms in Android. First, only a select set of sensors such as GPS, camera, bluetooth are considered to be privacy-prone and have their access mediated through protected APIs [3]. Other onboard sensors such as accelerometer, gyroscope, light, etc. are considered to be innocuous, requiring no user permission. This specific vulnerability of unrestricted access to accelerometer and gyroscope data has been exploited to mount keylogging attacks [43], and for reconstruction of travel trajectories [31]. Second, various studies [52, 28], to understand users' perception of privacy in general and their understanding of Android permissions in particular, reveal that users are often oblivious to the implications of granting access to a particular type of sensor or resource on their phone at install time. However, the perception quickly changes to one of concern when apprised of the various sensitive inferences that could be drawn using the shared data. Finally, users only have a binary choice of either accepting all the requested permissions or not installing the app at all. Once installed, users do not have any provision to revoke or modify the access restrictions during runtime.

Prior research have tried to address some of the above problems. TaintDroid [24] extends the Android OS by adding taint bits to sensitive information and then tracking the flow of those bits through third-party apps to detect malicious behavior. However, tainting sensor data continuously for all apps has high runtime overhead, and is often conservative as data sensitivity typically depends on user context. Moreover, TaintDroid stops at detection and does not provide any recommendation on countering the privacy threat. MockDroid [18] is a modified Android OS designed to allow users the ability to mock resources requested by the app at runtime. Mocking is used to simulate the absence of resources (e.g., lack of GPS

fix, or Internet connectivity), or provide fixed data. However, MockDroid only works for resources explicitly requested by an app (innocuous sensors are not handled), is binary because a user can either mock a resource or provide full access to it and finally MockDroid falls short on providing any guidance to the user regarding which sensors to mock. PMP [12] is a system that runs on iOS and allows users to control access to resources at runtime. It uses a crowdsourced recommendation engine to guide users towards effective privacy policies. However, PMP does not handle sensor data.

In this paper, we present ipShield [5], a privacy-enforcing framework on the Android OS. ipShield allows users to specify their privacy preferences in terms of semantically-meaningful inferences that can be drawn from the shared data, auto generates privacy actions on sensors, and supports an advanced mode for manual configuration of fine-grained privacy rules for accessed sensors on a per app basis at runtime. We build on prior work in [7, 18] and make the following contributions.

- We modified the Android OS to monitor all the sensors accessed by an app regardless of whether they are specified explicitly (e.g., in the manifest file for Android apps) at install time. As per our knowledge, ours is the first system that tracks innocuous sensors.
- We took an important step towards presenting the privacy risks in a more user-understandable format. Instead of listing sensors, we list the inferences that could be made using the accessed sensors. Users can specify their privacy preferences in the form of a prioritized blacklist of private inferences and a prioritized whitelist of allowed inferences.
- We implemented a recommendation engine to translate the blacklist and the whitelist of inferences into lower-level binary privacy actions (suppression, allow) on individual sensors.
- Finally, we provided the user with options to configure context-aware fine-grained privacy actions on different sensors on a per app basis at runtime. These actions range in complexity from simple suppression to setting constant values, adding noise of varying magnitude, and even play-back of synthetic sensor data.

ipShield is open source and implemented by modifying Android Open Source (AOSP) [2] version 4.2.2_r1. We evaluated it using computation intensive apps requiring continuous sensor data. Our results indicate that ipShield has negligible CPU and memory overhead and the reduction in battery life is $\sim 8\%$ in the worst case.

2 Case Studies

Using two typical scenarios we illustrate below how ipShield can help app users protect their privacy.

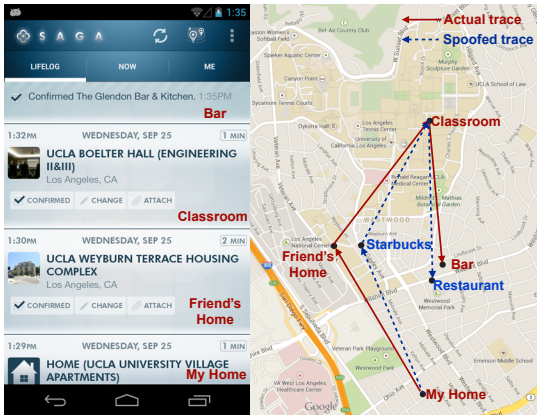


Figure 1: Left: Saga app showing actual trace of the user. Right: Both actual trace and spoofed trace on the map.

2.1 Transportation Mode and KeyLogging: Accelerometer/Gyroscope

Activity recognition algorithms [16, 53] are used by various fitness apps to infer the users' Transportation Mode (e.g., to predict one of three labels: *walking*, *motorized* or *still*). For example, the Ambulation app in [53] combines accelerometer and GPS data to infer the labels with over 90% accuracy. However, the same data can also be used to infer other labels sensitive to the user. For example, accelerometer together with gyroscope data can be used to perform keylogging and to infer keystrokes (Onscreen Taps) on the softkeyboard [43] (and separately to also infer Location [31]) with over 80% accuracy. This leads to the leakage of sensitive information like password and PIN entered on the phone.

Using ipShield, a user would add the Transportation Mode and the Onscreen Taps to the whitelist and the blacklist, respectively. This will block the accelerometer and gyroscope data from reaching the Ambulation app preventing keylogging. However, this will also cause the app to stop performing activity recognition. In Section 8 we will show how ipShield can be used to configure fine-grained rules and maximize the utility of the app.

2.2 Saga: Location

Saga [10] is a life logging app which runs in the background and keeps track of the places visited by a user. By analyzing the location trace of a user, Saga can infer useful information such as average daily commute time, time spent at work, etc. However, it can also derive sensitive inferences about locations such as *home, office, hospital* private to the user. Fig. 1(left) shows a mobility trace recorded using Saga. The user starts from home, picks up her friend and drives to school for class; later she also visits a nearby bar and wants to keep the visit private. In addition to this direct privacy requirement, there is also an indirect privacy concern. Saga

reveals the home location of the user's friend. The location information can be coupled with other online resources to identify the home owner, and infer that the friend had gone to the bar too. Thus, privacy of both the user and her friend is compromised, even though the friend is not using Saga. We therefore want ipShield to allow spoofing of location traces to protect visits to sensitive places. A plausible spoofed trace is shown on the map in Fig. 1(right). We illustrate how ipShield achieves this in Section 8.

3 Inference Privacy Problem

Inferences are labels (of activity, behaviour, places etc.) associated with data. We group labels of a similar (semantic) type into an *inference category*. The category names and the grouping are based on prior work (see Table 3). For example, *hospital, home, office* are grouped under Location category. An adversary tries to infer/predict these labels from the shared data. The prediction accuracy of an inference category corresponds to correctly predicting a label in that category. We now define the inference privacy problem.

Problem statement: Data is typically shared with an app for a specific set of inference categories. For example, in Section 2.1, data is shared for inferring the Transportation Mode, and in Section 2.2 it is for inferring travel statistics. These categories and their labels form a whitelist which the user wants to allow and obtain utility. However, the same data can be used to infer keystrokes and sensitive locations – inferences sensitive to the user. The sensitive categories and their labels form the blacklist which the user wants to keep private. Each inference category can also be associated with a user specified priority level (Section 6.2). The privacy problem is to design a system which will take as input a whitelist and a blacklist of prioritized inference categories and translate them into privacy actions on sensors such that the two lists are balanced as per user specified priorities.

Side-Channel Attacks: Traditionally, side channel attacks are designed to exploit the information revealed by execution of cryptographic algorithms to recover their secret key. These attacks typically use information channels which include but are not limited to running time, cache behavior, power consumption pattern, acoustic emanations and timing information [36, 54, 29] during execution. Sometimes, even with no program execution, information side channels can exist due to physical signals emanating from a hardware while it is being used by a user. For example, acoustic [13, 42] and electromagnetic [57] emanations from a keyboard has been used to infer keystrokes and recover sensitive passwords and PINs. The feasibility of such attacks on the smart phone using sensor data has been demonstrated in [14, 43].

Privacy Analysis	Kirin [25], SOM [17], Stowaway [27]
Privacy Detection	**Static**: BlueSeal [32]
	Dynamic: TaintDroid [24]
Privacy Mitigation	**Mobile Based**: Dr. Android Mr. Hide [34], PMP [12], Apex [45], MockDroid [18], AppFence [33], pDroid [7], πBox [38]
	Cloud Based: Lockr [56], PDV [44], Persona [15]

Table 1: Categorization of prior work.

Our inference privacy problem differs from traditional side-channel attacks in several ways. First, the shared data used for the attack are not unintended physical signals emanated from the hardware, or covert timing information but sensor data intended for the recipient. Second, in our setting the recipient is also the adversary whereas in case of side-channel attacks the adversary is typically different from the intended recipient. Finally, at least in principle, the side-channel attacks can be prevented by placing the computational hardware in physically isolated and secure chamber whose boundaries the electromagnetic, acoustic and such emanations cannot cross which is not the case in our scenario. In [14], inferring the keystrokes is referred as a side-channel attack. However, we call it a blacklist inference as the sensor data are intended to be shared with the app for the whitelisted inferences and are not a side-channel.

4 Related Work

We group prior work on systems for protecting privacy under three broad categories as shown in Table 1. The Privacy Analysis category summarizes contributions towards analysis of the Android permission model, conformance of the various apps to this model, the usage pattern of permissions across apps, and finally the expressibility of the permission model [25, 17, 27]. Under Privacy Detection we have tools such as BlueSeal [32] which use static analysis of the app bytecode to detect if sensitive information is being leaked over the network interface and inform it to the user at install time. Other systems like TaintDroid [24] use dynamic flow tracking to detect malicious app behavior. However, both techniques can only alert the users of malicious behavior (BlueSeal at install time, and TaintDroid at runtime), and do not provide suitable mechanisms to prevent information leakage.

Under the Privacy Mitigation category we group privacy systems implemented on mobile platforms. Dr. Android and Mr. Hide [34] instrument and modify the app's dex bytecode to ensure that access to all private resources is made available only through their trusted interface. AppFence [33] builds on TaintDroid and provides *shadow* or synthetic data to untrusted apps and measures the effect of such data on app utility. Other systems in [18, 7, 45, 12, 38] provide users with the

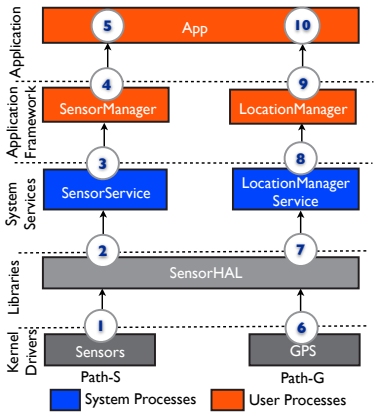


Figure 2: The data flow path from sensors to apps. Same colored blocks represent components within the same process.

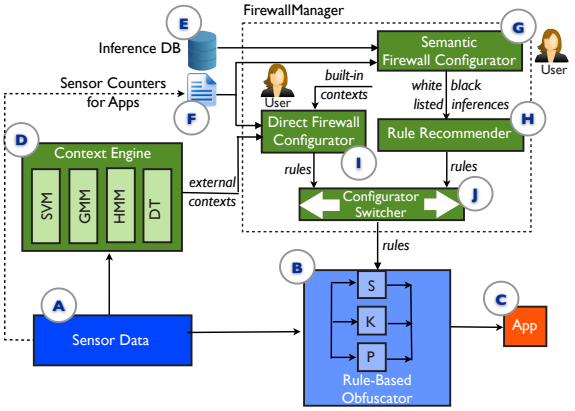


Figure 3: ipShield data flow.

ability to control access to their resources at runtime - a feature that is currently being integrated into the latest Android release [1]. However, the above systems provide binary access control to resources, do not monitor access to innocuous sensors, and lack high-level user-understandable privacy abstractions (inferences). Cloud-based solutions in [56, 44, 15] are for protecting privacy of data streams but require additional infrastructure. A detailed exposition of other various privacy preserving techniques, and initial ideas on ipShield can also be found in [21].

5 Background: Android

Below we describe data paths, from sensors to apps and also highlight Android's security model [3] to understand the process level isolation of the components.

5.1 Android Sensor Data Flow Path

We consider two data paths as shown in Fig. 2. Path-S is used by sensors such as accelerometer, gyroscope, light and so on. Path-G is from the GPS to apps. Note that the paths are simplified representation showing only the components of the Android OS that are relevant to ipShield. `SensorService` and `LocationManagerService` are system services (running continuously in the background) which are started by the Android OS at boot time. These services run as separate threads within the `system_server` process, poll the `SensorHAL` layer for sensor data and are responsible for pushing the data to the apps. The apps typically do not communicate directly with the services. Each system service has a corresponding `Manager` which acts as its proxy. Thus, to access sensor data an app instantiates either a `SensorManager` or a `LocationManager` object and uses the object's public methods to register an event listener for the desired sensor. As shown in Fig. 2, both the app and the manager objects are part of the same process (which also runs the Dalvik Virtual Machine).

5.2 Android Security Model

Application Sandboxing: The core of the Android OS is built on top of the Linux kernel, and this allows Android to re-purpose the traditional security controls built into Linux to protect user data, system resources, and to provide app isolation. Android enforces kernel level *Application Sandboxing* for every software that runs above the kernel which includes all apps, OS libraries, OS-provided app framework and app runtime. The Android system sets up the sandbox and enforces security between apps by assigning a unique user ID (UID) to each app and by running it as that user in a separate process. Running apps within a sandbox environment ensures that any memory corruption error will only allow arbitrary code execution in the context of that particular app and with the permissions established by the OS. User-specific privileges also ensure that files created by one app cannot be read or altered by another app.

Secure IPC: Android not only supports traditional mechanisms such as filesystem, sockets and signals but also implements newer and more secure mechanisms such as Binder and Intents.

Access Control Using Manifest: Finally, Android controls app access to resources by designating certain APIs (such as camera, location, bluetooth etc.) as protected [3]. To use these resources an app needs to define its requirements in its manifest (a control file provided by every app). The user can either grant all of the requested permissions as a block or not install the app at all.

6 Architectural Design

The design of ipShield is guided by four objectives – better monitoring of sensor access, meaningful privacy abstraction, privacy rule recommendation and fine-grained control over shared data. The architectural requirements to achieve the above functionalities are

shown in Fig. 3 and can be broken down into four major blocks – (i) Databases (ii) Context Engine (iii) FirewallManager (iv) Rule-Based Obfuscator. We describe each of the blocks and their components in detail below.

Databases: We maintain two databases: Sensor Counters and Inference DB. Currently, apps have unrestricted access to the class of innocuous sensors. One of our goals in ipShield is to instrument the OS to monitor the number of sensors accessed by an app. The information is populated in the Sensor Counters database (marked (F)) and is provided as an input to the FirewallManager block. The database needs to be updated when a new sensor is accessed by an installed app or when an app in uninstalled.

Motivated by the database of virus signatures maintained by antivirus software, we maintain a similar database for mapping the list of inference categories (and their labels) that could be predicted using a combination of sensors, together with the prediction accuracy and the machine learning algorithm employed (Table 2 shows a small subset of the Inference DB). Advances in sensing coupled with increases in the sophistication of learning algorithms result in newer inference categories and improved accuracy. The inference DB (marked (E)) thus needs to be kept updated.

Context Engine: For granularity of rules, ipShield allows trusted Context Engines (marked (D)) to register and provide as input context labels. A context engine is a set of machine learning algorithms, which take as input raw sensor data and output the current context label. An inference label is same as the context label but it is inferred from the shared data by the adversary. A user can configure privacy rules to trigger on context labels.

FirewallManager interacts with the user and is responsible for generating the privacy rules. There are four different sub-blocks within FirewallManager (marked (G) through (J)).

The Semantic Firewall Configurator ((G)) takes as input the sensors accessed by an app and queries the Inference DB to present the user with a list of possible inference categories that can be predicted by the app. Using inferences instead of sensors allow us to better communicate the privacy risks to the user [52]. The user then configures a whitelist and a disjoint blacklist from the enumerated list of inference categories.

The Rule Recommender ((H)) (Section 6.2) takes as input the privacy preferences of the user expressed in terms of the whitelist and blacklist and translates them to actual privacy actions on the sensors. We observe that the privacy actions are dependent on the inference labels, the learning algorithm employed and the features used. Therefore, to keep the recommender simple and generic we limit the auto-generated privacy actions to Normal and Suppress (Section 6.1). While the auto-generated rules are binary and conservative, we provide the user with the flexibility to override them.

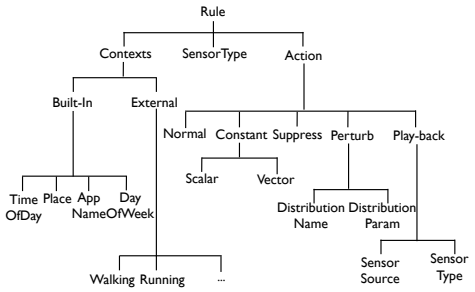


Figure 4: Tree showing all the possible options currently implemented in ipShield for constructing privacy rules. The leaf nodes of the tree are instantiated to form the privacy rules.

The Direct Firewall Configurator ((I)) allows the user to manually configure fine-grained context-aware rules. The contexts used can either be ones provided by ipShield, or ones which are externally obtained from the trusted Context Engine. ipShield is designed to operate in the Semantic mode. The Direct Configurator is an optional mode, which provides flexibility of rule configuration at the cost of increased human interaction.

Finally, the Configurator Switcher block ((J)) allows the user to switch between the Semantic and Direct Configurator modes and configure rules.

Rule-Based Obfuscator ((B)) implements the different privacy actions. It takes as input privacy rules and sensor data and, depending on the app, applies the appropriate rules to the data before releasing them.

6.1 Taxonomy of Privacy Rules

The complete list of choices for configuring privacy rules is illustrated in Fig. 4. A rule has three basic parts: Context, SensorType, and Action. We also allow conjunction (denoted by the $\wedge$ operator) of the context labels within a rule. The general form of a rule is if $(\wedge_{i=1}^{n} Context_i)$ then apply $Action$ on $SensorType$. For example, if $((TimeOfDay \text{ in } [10am-5pm]) \wedge (Place = school) \wedge (AppName = facebook))$ then apply $Action = Suppress$ on $SensorType = gps$. As shown in Fig. 4 some of the simple contexts such as TimeOfDay, DayOfWeek, Place and AppName are built into ipShield. External contexts provided by a registered Context Engine can also be used to configure rules. SensorType refers to the sensor (e.g., accelerometer, GPS, gyroscope) on which the action is to be applied.

Excluding the default action of releasing data without any changes (Normal), ipShield currently supports four different privacy actions. The Suppress action (S-block in (B)) when applied blocks data from reaching an app and the app is unable to detect any sensor event. The Constant action (K-block in (B)) allows user to replace

actual data with a constant value. The user-specified constant can be vector or scalar valued depending on the type of sensor whose data is being replaced. The Perturb action (P-block in (B)) can be used to add noise to sensor data. The noise values can be drawn from different probability distributions, the parameters of which are input to this action. Finally, the Play-back action can be used to suppress the data from the actual sensor hardware and instead send synthetic sensor measurements from an external service to the requesting app (U-shaped datapath). The synthetic data source and sensor type are input to this action. The Play-back option can be used for generating any arbitrary transformation on the data offline.

6.2 Rule Recommender

The Rule Recommender takes as input the whitelist and the blacklist of inference categories and generates a configuration for enabling or blocking of sensors accessed by an app. The goal is to ensure that only those inference labels which form part of the whitelist are allowed and those in the blacklist are blocked.

6.2.1 Problem Formulation

Let N be the number of sensors used by an app (obtained from Sensor Counters) and $\mathbf{s} = [s_1, \ldots, s_N]$ represent the sensor state vector where $s_i \in \{0,1\}$ represents the state of the i^{th} sensor. Setting s_i to 0 indicates that the sensor is disabled and a value of 1 indicates that the sensor is enabled. We denote the set of inference categories by $\mathscr{L} = \{l_1, \ldots, l_{|\mathscr{L}|}\}$. We define a mapping $M : \{0,1\}^N \times \mathscr{L} \rightarrow [0,1]$ where $M(\psi, l) = 0$ indicates that there exists no learning algorithm which can use the data streams from the sensors which are enabled as per state vector ψ and infer a label in category l. A non-zero value of $M(\psi, l)$ correspond to the maximum accuracy among all the learning algorithms that can be used to infer category l from the data streams released as per the state vector ψ. A value of 1 indicates that l can be perfectly inferred using the enabled sensors. Note, we might use different learning algorithms to predict the same labels using different sensor state vectors. The mapping M is obtained from the Inference DB. Learning algorithms typically work on features extracted from the raw sensor data. But, our current model is agnostic to features because we are sharing the raw sensor data itself and hence every required feature can be extracted from it. The set of whitelisted categories $\mathscr{W} \subseteq \mathscr{L}$, and the set of blacklisted categories $\mathscr{B} \subseteq \mathscr{L}$, are as specified by the users such that $\mathscr{W} \cap \mathscr{B} = \emptyset$. Finally, let $p_l \in \{0, \ldots, P_{max}\}$ denote the priority level set for category l by the user such that a higher value of p_l indicates higher priority. The priority levels represent a relative gradation of risk as perceived by the user. For example, $P_{max} = 3$ could correspond to *low*, *medium* and *high* levels of perceived risks. We use the above notations to formulate the inference-privacy problem as the following constrained optimization problem

$$\max_{\psi \in 2^N} \quad \sum_{l \in \mathscr{W}} M(\psi, l) 2^{p_l} - \sum_{l \in \mathscr{B}} M(\psi, l) 2^{p_l} \quad (1)$$

$$\text{s.t.} \quad \sum_{\substack{l \in \mathscr{B} \\ p_l = P_{max}}} M(\psi, l) = 0. \quad (2)$$

The objective function in Eqn. 1 is designed to maximize the prediction accuracy of the whitelisted labels and minimize the prediction accuracy of the blacklisted labels. The priorities are exponentially scaled up to account for whitelisted labels which can be detected with low accuracy than other labels but have a higher priority. The constraint in Eqn. 2 ensures that users can force blacklisted inferences to be blocked by setting their priority to P_{max}. We note that the search space in the optimization problem shown in Eqn. 1 is constrained to the vector of elements with 0's and 1's corresponding to the enabled and blocked sensors respectively. It then follows that the search space is constrained to the vertices of a hypercube. It is also easy to show that this search space is non-convex. Moreover, the optimization function depends on the relation M on which we impose no structure or even linearity. Thus, our program is non-linear integer program which is non-convex and NP-complete. We observe from our investigation of prior work and apps from Google Play (Section 8) that $N \leq 6$ for almost all the apps. Therefore, to solve a specific instance of the optimization problem above (a given choice of whitelist, blacklist, N, and priorities) we apply brute force and enumerate all possible state vector combinations. We filter out all state vectors which satisfy the blacklisted constraint and maximize the objective function over this reduced space. The output vector ψ shows which sensors should be enabled or disabled mapping preferences on inferences to privacy actions on sensors. There will be scalability issues for large N (> 15) but in practice we do not think there will be a single inference made using 15 different sensor types on a phone in the near future.

6.2.2 Numerical Example

We return to the motivating example (Section 2.1) and express it in terms of the notation described above. Thus, $\mathscr{L} = \{\text{Transportation Mode}, \text{Location}, \text{Onscreen Taps}\}$, $\mathscr{W} = \{\text{Transportation Mode}\}$ and $\mathscr{B} = \{\text{Location}, \text{OnscreenTaps}\}$. The mapping M is presented in Table 2 (under Inference Categories). We set the maximum priority level $P_{max} = 10$ throughout this example and represent a user specified priority vector as a tuple $(p_{transport}, p_{location}, p_{tap})$. We apply the algorithm above for different choices of priority vectors and report the evaluation results also in Table 2 (under column titled Evaluation).

Sensor Combination	Inference Categories			Evaluation		
	Transportation Mode	Location	Onscreen Taps	Priority1 $\{10,4,10\}$	Priority2 $\{10,0,7\}$	Priority3 $\{5,9,9\}$
GPS+ Acc + Gyro	95%	97%	80%	0	**869.4**	-875.8
GPS+WiFi	83.1%	97%	0%	**835.4**	849.9	-470.0
GPS+GSM	81.7%	98.2%	0%	820.9	835.6	-476.6
GSM+WiFi	72.9%	94.03%	0%	731.45	745.5	**-458.1**
GSM+Wifi+Acc+Gyro	92%	94.03%	80%	0	838.7	-861.6
Wifi+Acc+Gyro	91.1%	23.08%	80%	0	830.2	-498.6
GSM+Acc+Gyro	88.1%	94.03%	80%	0	798.8	-862.8
GPS	75.8%	97%	0%	760.7	775.2	-472.4
GSM	61.8%	94.03%	0%	617.8	631.9	-461.7
Acc+Gyro	84.6%	23.08%	80%	0	763.7	-500.7

Table 2: Left: A portion of the Inference DB (mapping M). Each entry (in %)is the maximum prediction accuracy for the inference category using the sensor combination. Right: The objective function (Eqn. 1) evaluated for different priority vectors and $P_{max} = 10$.

Consider $Priority1 = (10,4,10)$ as the selected priority vector. The user is not too concerned about revealing his Location and sets $p_{location}$ to 4. She however wants to strictly suppress the detection of Onscreen Taps and sets p_{tap} to 10. A high priority is also given to the whitelisted inference category by setting $p_{transport} = 10$. The objective function values for the different sensor combinations is shown in the column under heading Priority1. The maximum occurs for the combination corresponding to GPS+WiFi and is selected by the recommender. The selected sensor state vector is such that accelerometer data is suppressed in order to guarantee no leakage of the Onscreen Taps information. We also note that GPS+WiFi configuration provides higher accuracy in predicting the Transportation Mode and lower accuracy for Location prediction compared to other sensor combinations.

We consider another scenario with priority vector $Priority2 = (10,0,7)$. In this case, the user does not worry about Location disclosure, but wants to increase the prediction accuracy of the Transportation Mode while blocking the Onscreen Taps if possible. The objective function values are shown under column Priority2. The recommender selects the GPS+Accelerometer combination which is biased towards performance. In addition to meeting blacklist requirements the combination also provides the best accuracy.

Finally, the third user has high levels of concern about revealing both Location and Onscreen Taps information. She would like to trade the performance with privacy and thus selects a priority vector $Priority3 = (5,9,9)$. The resulting sensor combination chosen is GSM+WiFi. The rule recommender starts by suppressing the accelerometer data (to prevent tap inference). It then selects the combination which results in the worst Location inference from among the remaining set of combinations (GSM and GSM+WiFi), while simultaneously maximizing the whitelist accuracy.

Model-Based Augmentation of Rule Recommender: Prior research has shown that a user's various context labels and transitions between them can be captured by a Markov chain [30], by using a Dynamic Bayesian Network [47], or explicitly enumerated [20]. A user specifies a whitelist and a blacklist of inference categories, and depending on the current context label and the learned model the system can determine whether to release a context with a particular probability. In other words, the probability of release of a context should not increase the adversarial accuracy of predicting a blacklisted inference label. We envision that such model-based techniques can also be included in our recommendation system for generating a richer set of privacy rules.

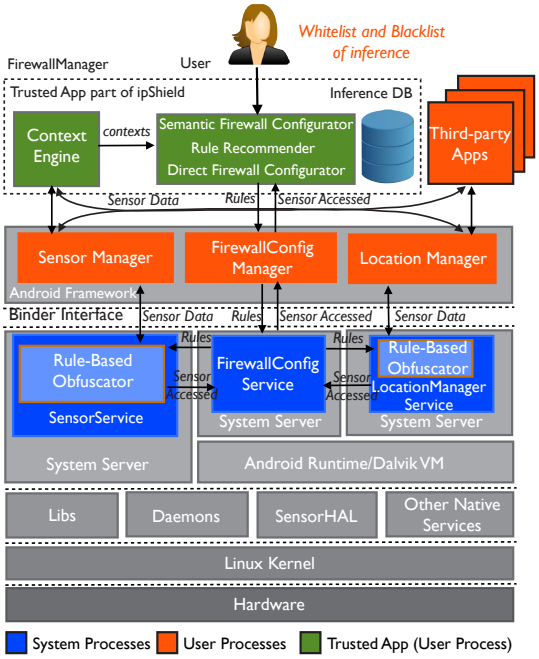


Figure 5: Implementation of ipShield on Android.

7 Implementation

The implementation of ipShield within the Android stack (Fig. 5) is described below.

7.1 Trust Model

We assume that the user installed third-party apps (e.g., from Google Play) are untrusted but do not collude with each other and share information. We trust the Linux kernel on which Android OS is built and also the Application Sandbox implemented by the kernel (Section 5). We extend the chain of trust to include the OS libraries and system services which run within the Application Sandbox and are protected by UID and group ID privileges. However, recent successful exploits from Facebook on modifying the internal data structures of the Dalvik VM [11] leads us to not trust the Application Framework components which run within the same process as the Dalvik VM.

7.2 Intercepting Data: Possible Choices

As indicated by the markers (①-⑩) in Fig. 2, there exist different operating points in both the data paths (Path-S and Path-G) at which we can intercept the sensor data, apply privacy actions, and obfuscate it. However, also associated with an operating point is the implementation complexity of the Rule-Based Obfuscator block (Ⓑ in Fig. 3) at that point and also its vulnerability to security attacks. We discuss below the trade offs in selecting an operating point.

Points ① and ⑥, correspond to modifying the kernel drivers to obfuscate data. While the drivers are protected by kernel security mechanisms, they require our implementation to be vendor specific. It is also hard to push app and rule information to the drivers and periodically update rules inside a driver.

Points ② and ⑦ correspond to changes in the SensorHAL layer. The HAL provides the abstraction between device specific kernel drivers and the Android system above. However, changing the HAL like the kernel driver has a high implementation complexity in terms of pushing app and rule information.

Points ③ and ⑧, correspond to modifying the Android system services, namely `SensorService` and `LocationManagerService` which are responsible for handling the different sensors and the GPS respectively. These services as shown in Fig. 2 run in a process separate from the app and hence are protected by the Application Sandbox. Both `SensorService` and `LocationManagerService` maintain information about installed apps, and can be easily signaled using binder calls and as we show later in Section 8 they incur low overhead while updating rules.

Points ④ and ⑨ correspond to changing the `SensorManager` and `LocationManager`, respectively. These points have the least implementation complexity, however both `SensorManager` and `LocationManager` run within the same process as the app, and hence they are not protected by process-level isolation. Recent exploits have used the above security vulnerability to modify code data structures at runtime [11].

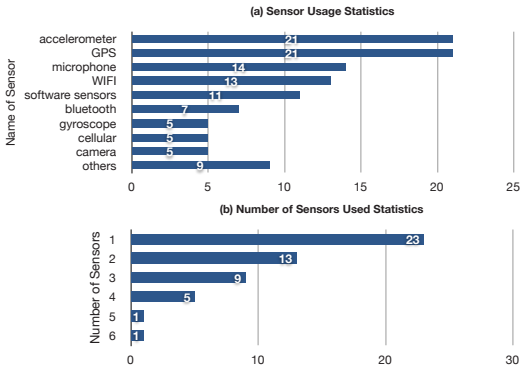


Figure 6: Statistics of sensor usage from the Inference DB.

Finally, points ⑤ and ⑩ correspond to static analysis of the app code to understand privacy violations [32]. However, the information flow approaches are often conservative, incur large instrumentation and runtime overhead, and typically stop at identification of a malicious app. Based on the available choices we decided to implement the Rule-Based Obfuscator block within the OS in the `SensorService` and `LocationManagerService` blocks in the respective data paths.

7.3 ipShield Code Blocks

ipShield is an open source project. The code for each of the blocks together with complete instructions for downloading and installing ipShield are available at [5].

7.3.1 Databases

Sensor Counters: This database, implemented as a file, maintains a counter for each sensor on a per-app basis. The counter for a sensor represents the number of events from the sensor that have been sent to the app. We use an `unsigned` $64-bit$ `long int` for our counter. Even at the maximum sampling rate of a sensor, under continuous sensing, the counter will not overflow within the lifetime of a phone. The entry for an app together with the counters are deleted when the app is uninstalled from the phone. A counter value of zero indicates that the sensor is not being used by the app. These counters are maintained by {`Sensor`, `LocationManager`}Service and are periodically written to the `/data/sensor-counter` file every minute. The permissions on the file are such that it can be read by any app but can be written to only by system services.

Inference DB: A knowledge repository generated from a survey of 60+ papers published in relevant conferences and journals over the past $3-5$ years. This

Inference Category	Labels
Transportation Mode [53]	still, walking, motorized
Device Placement [48]	hand, ear, pocket, bag
Onscreen Taps [43]	location of taps on screen
Location [19] [35] [46]	home, work, public, restaurant...
Emotion [50] [22]	happy, sad, fear, anger, neutral
Speaker [39] [46]	male/female, identity
Text Entered on Phone [43]	alphabets
Stress [40] [22]	stressful or not

Table 3: Selected inference categories from Inference DB.

database captures a wide variety of inference categories a small set of which is shown in Table 3. For each inference category, we store the prediction accuracy over the constituents labels for a particular sensor combination. If there are multiple papers using the same sensor combinations predicting the same set of labels we store details of the one with highest accuracy. We also maintain information about the set of sensors used, the features extracted from the sensor data, the classifiers used, and finally the paper title under which the results were published. In Fig. 6, we show statistics of sensor usage computed using the inference database. Based on our survey, we found that (a) GPS and accelerometer sensors are the most commonly used; (b) the number of sensors accessed by any app is almost always less than 6 (we do not include papers which use external body worn sensors in the plot, but even externally worn sensors are less than 6 types). While newer inferences are being made, we do not expect the database to change rapidly. To enable crowdsourcing of the inference DB, we have designed and published a web interface where people can contribute entries [5]. Currently, we rely on manual screening of the received entries before adding them to the Inference DB.

7.3.2 Context Engine

To allow fine-grained context-aware rules, ipShield allows trusted external context engines to register contexts that they can provide using the interface in Fig. 7(e). The user can then configure rules which will be triggered on a particular context. ipShield expects the context engines to use Android supported intents (`action`=label) as the IPC mechanism for providing the context labels.

Contexts such as battery status, contact list, ringer status etc., do not require access to sensor data and can be obtained through APIs provided by the Android OS. However, for contexts that require sensor data, the external context engine must have access to raw sensor data. To implement this when a data buffer from the HAL is received by the `SensorService` and/or the `LocationManagerService` it is first sent to the context engine to get the current context label. On receiving the context, the associated rules are then loaded and used by the Rule-Based Obfuscator to obfuscate the data buffer.

We modified the Transportation Mode app [53] to implement an activity context engine and test its integration with ipShield. In our implementation, the context engine used `SensorManager` for subscribing to accelerometer data at the rate of *SENSOR_DELAY_GAME*. This resulted in sensor data at a rate of $50Hz$ or a sample every $0.02s$. We used data buffered over a sliding window of $1s$ for inferring the activity context. On an average, the engine took about $8ms$ to generate activity context from a $1s$ accelerometer window. Even with additional overhead due to binder call and rule loading, we found that the associated rules can take effect before the next sensor data sample. This meant that our buffer size could be equal to $1s$ of data without losing any sample. In general, for keeping the buffer size bounded we observe that the processing time of the context engine together with the rule update time should be less than the inter arrival time between two data samples.

7.3.3 FirewallManager

The FirewallManager is a trusted Android app which has three different components described below.

Semantic Firewall Configurator: This is an Android activity. It reads the Sensor Counters for the installed apps and queries the inference DB for possible inference categories for each app. When launched it displays this information (Fig. 7 (a)) for the user. Once the user selects an app she is presented with the inference categories with an option to classify each into a whitelist or a blacklist (Fig. 7 (b)). The Configurator then passes the data user preferences to the Rule Recommender.

Rule Recommender: The algorithmic aspects of the rule recommender are described in detail in Section 6.2. It is implemented within the Semantic Firewall Configurator. It then uses the `FirewallConfigManager` to write the rules to `/data/firewall-config` file and also use a binder call to signal the `SensorService` and `LocationManagerService` to reload the new rules.

Direct Firewall Configurator: In this mode the user can configure context-aware privacy rules (Fig. 7 (c) and (d)). The user can specify actions on sensors used by apps, and for each action also associate either built-in contexts such as TimeOfDay, DayOfWeek, Place, or external contexts as triggers. For defining the Place context, the user can drop a marker on the map as shown in Fig. 7 (f) to annotate a $< latitude, longitude >$ tuple with a significant place name. For external contexts the Configurator implements a `BroadcastReceiver` which listens for intents. When an intent containing a particular label is received, the `BroadcastReceiver` invokes a rule loader service which passes a pre-configured set of rules associated with the label to the `FirewallConfigManager`. The `FirewallConfigManager` in turn writes the rules into the `/data/firewall-config` file and signals

both SensorService and LocationManagerService to reload the new rules. Note that the user can also explicitly request for loading a new set of rules.

7.3.4 Rule-Based Obfuscator

The Rule-Based Obfuscator block is responsible for enforcing the actions specified by the privacy rules described in Section 6.1. This block is implemented both within LocationManagerService and SensorService with the same functionality. The rules are read from the /data/firewall-config file and inserted into a HashMap for faster access. The serialization and deserialization of both rules and Sensor Counter is implemented using Google Protocol Buffer [8]. The hash for each rule is computed on the fields {appName, UID, sensorType, ruleSeqNum} where ruleSeqNum is a sequence number assigned to a rule for a sensorType. This allows multiple rules for a sensor implementing the OR operation on contexts (AND operation is implemented by allowing multiple contexts for each rule). UID is assigned to an app by Android at install time.

FirewallConfigManager, FirewallConfigService: The FirewallConfigManager interfaces with both the Semantic and the Direct Configurator modules of FirewallManager app and communicates the privacy rules to the FirewallConfigService through the binder interface. The service runs within a system process and writes the rules to the /data/firewall-config file and signals the LocationManagerService as well as the SensorService to reload the new rules.

8 Evaluation

We implemented ipShield by modifying the Android Open Source Project [2] (AOSP, branch 4.2.2_r1). We deployed and performed all our tests on the Google Nexus 4 phone (1.5GHz quad-core Qualcomm Snapdragon™ Pro, 2GB RAM).

8.1 Performance Overhead

We measured the overhead incurred by running ipShield to highlight that it is feasible to deploy it on current mobile platforms without impacting user experience in terms of battery life and app responsiveness.

8.1.1 Rule Access

Android supports four different sampling rates. On the Nexus 4 we found that on average SENSOR_DELAY_NORMAL and SENSOR_DELAY_UI are less than 10Hz, SENSOR_DELAY_GAME is around 50Hz and SENSOR_DELAY_FASTEST is around 200Hz. In Fig. 8 (a), the blue bars show the times taken to load the rules from the file (/data/firewall-config) into the

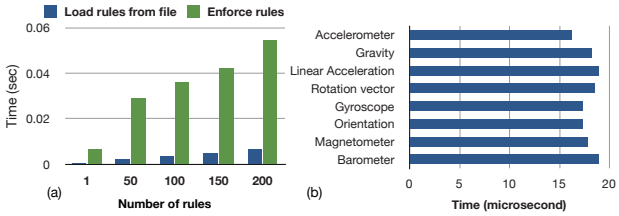


Figure 8: (a) Time taken in for the rules to load into memory and take effect. (b) Time overhead to fetch one sensor data sample sampled at SENSOR_DELAY_FASTEST.

HashMap, which are negligible. The green bars in the figure represent total time for the rules to take effect after configuration. For SENSOR_DELAY_NORMAL and SENSOR_DELAY_UI no data sample will be released before the new rules take effect even for 200 rules. In reality, we believe that the number of privacy rules will typically be less than 50, therefore for SENSOR_DELAY_GAME and SENSOR_DELAY_FASTEST less than 2 and 6 samples will be released before the 50 rules take effect, respectively.

8.1.2 Sensor Data Access

The overhead i.e., difference in time for fetching one data sample using ipShield compared to that on unmodified AOSP is shown in Fig. 8 (b). The overhead is computed by taking the average of fetching 30000 samples. Each sensor is sampled at SENSOR_DELAY_FASTEST (200Hz). The time for ipShield is averaged over the time for performing each of Constant, Perturb, and Normal (no change) actions on every accessed sample. We can see that the access overhead per sample is less than 20μsec – negligible even for the fastest sampling rate.

8.1.3 CPU and Memory Overhead

The Rule-Based Obfuscator block is part of both SensorService and LocationManagerService which run as threads inside the system_server process. For each data sample, the Rule-Based Obfuscator block is called to apply the privacy actions. We compare the overhead of the Obfuscator block with AOSP by profiling the average CPU utilization of the phone while running the Ambulation app [53] which continuously requests sensor data (GPS, accelerometer) at a rate of 1Hz on a Nexus 4 phone. CPU utilization with AOSP averaged 2%. CPU utilization with various privacy actions averaged 2.5% and never exceeded 3%. It should be noted that the CPU utilization (and hence energy consumption) will scale with the sampling rate. As shown in the energy analysis that follows this section, we believe that the overhead of ipShield is small enough to have negligible effects on overall system performance.

Memory overhead for the transformations is shown in Fig. 9 (a). The highest overhead is for Perturb and is less than 0.5MB. There is a dip in memory usage for Sup-

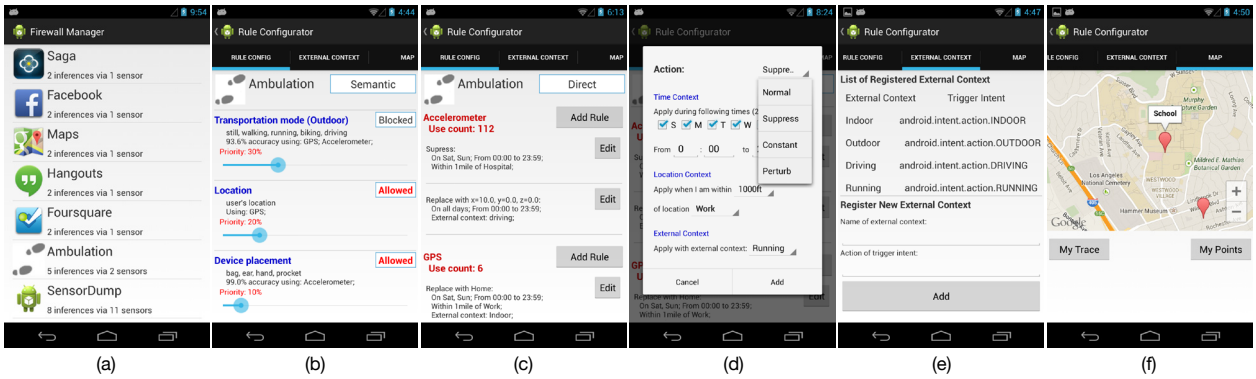


Figure 7: (a) List of installed apps showing number of sensors and number of possible inferences. (b) Semantic Firewall Configurator showing list of inference categories with option to block or allow. (c) List of rules configured for different sensors. Multiple rules with combination of contexts can be configured for each sensor. (d) Direct Firewall Configurator for privacy actions and their parameters. (e) List of external contexts registered with FirewallManager and ability to add new ones. (f) Screen to annotate significant places on the map (provides built-in Location context for rules).

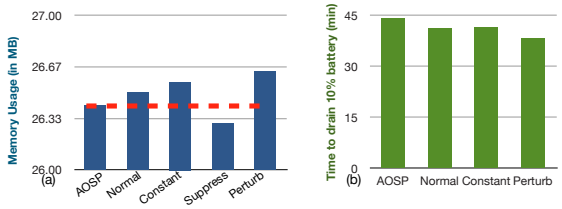


Figure 9: (a) Memory Overhead. (b) Energy Overhead.

press action which lowers the average memory overhead over all the operations in ipShield to around 0.07MB.

8.1.4 Energy Overhead

We compare the energy overhead of ipShield to AOSP by plotting the time to drain the phone battery from 100% to 90%, while the Ambulation app is continually running in the foreground for Transportation Mode inference. All network interfaces and radios are turned off, and the screen display is on at the lowest brightness. We acquire a CPU wakelock in the app to prevent the phone from sleeping. The inference frequency of the app is set to 4Hz. We measure the drain due to three actions: Normal, Constant, and Perturb which will consume more power than the AOSP. Fig. 9 (b) shows the results: ipShield on average drains the battery 3min 37s ($\sim 8.2\%$) faster than AOSP, which we consider as a marginal overhead. In typical usage scenarios where the screen is at a higher brightness setting and the network subsystem is active we expect the energy overhead for ipShield to be relatively lower.

8.2 Vulnerability of Current Apps

We did a survey of the top 60 free apps from Google play store to find the different sensors used by these apps. We installed and executed each of the apps from the play store, and noted the permissions to sensors requested by the apps at install time, and also the sensors which were being accessed without permission using ipShield. We also made use of the description of the app provided at the app store for additional information (if any). This provided the list of sensors used by each app. We then used the Inference DB to create the association between app and possible inference categories as shown in Table 4. The results from this survey validates our claim that GPS and accelerometer which are the most used sensors in academic research (Fig. 6) are also the most widely used sensors in apps. We further note, that many of these apps have access to data from innocuous sensors, combinations of which can be maliciously used to predict a lot more inferences than what they advertise.

8.3 Case Studies: Revisited

We now illustrate how ipShield can be used to configure simple rules to overcome the privacy issues outlined in the examples in Section 2.

***Transportation Mode and KeyLogging*:** While suppressing accelerometer at all times is a naive solution, to obtain better utility from the app, the user can use the Direct Rule Configurator to select an external context KEYBOARD_UP, and use it to define the following rules: If $((TimeOfDay \text{ in } [12am - 11:59pm]) \wedge (ExternalContext = KEYBOARD\_UP) \wedge (AppName = Ambulation))$ then apply $action = Suppress$ on $SensorType = acc$; and a similar rule for suppressing $SensorType = gyro$. We exploit the fact that it is sufficient to block the accelerometer and gyroscope data while the softkeyboard is active to protect against keylogging. On executing the above rules, the act of suppression will inform an

Sensor Combination	App Name	Inference Categories
GPS	Twitter, iHeartRadio, Calorie Counter, Amazon, eBay, MyTracks, Google Earth (and 12 more..)	A1: Loc, Speed, Route
Acc	Despicable me, Subway Surfers, Accupedo Pedometer	A2: TM, Device Placement, Text on Keyboard
Audio	Snapchat, Vine, Google Translate, Cadiograph	A3: Speaker, Loc, Emotions, Stress
Acc + GPS	Picsart, Noom Weight Loss Coach, Temple Run	A1 + A2
GPS + Audio	FB, FB Messenger, Tango, Whatsapp, Shazam, GoSMS	A1 + A3
Acc + GPS + Audio	Instagram, Neon motocross	A1 + A2 + A3
Acc + Pro + GPS + Audio	Skype	A1 + A2 + A3
Acc + Rot + GPS	Maps	A1 + A2, Onscreen Taps, Text Entered on Phone
Acc + Gyro + Pre + GPS	Saga Lifelogging	A1 + A2, Onscreen Taps, Text Entered on Phone

Table 4: Sensors and possible inferences from top apps in Google Play Store (all have access to network). Loc:Location, Acc:acclerometer, Pro:proximity, Rot:rotation vector, Gyro:gyroscope, Pre:pressure, TM:Transportation Mode.

adversary that the user is entering text, but she cannot infer anything more. The Ambulation app will now continue to work at all times when the user is not entering text, maximizing its utility to the user.

***Saga and Location*:** A user would often like to keep some of his visits to sensitive places private. ipShield allows the user to configure the following rules to spoof her location trace: (1) If $((TimeOfDay$ in $[12am-11:59pm]) \wedge (Place = Friend'sHome) \wedge (AppName = Saga))$ then apply $action = Constant$ and $value = Starbucks$ on $SensorType = GPS$; (2) If $((TimeOfDay$ in $[12am-11:59pm]) \wedge (Place = Bar) \wedge (AppName = Saga))$ then apply $action = Constant$ and $value = Restaurant$ on $SensorType = GPS$; As we mentioned earlier user can configure labels such as *Starbucks, friend's home, bar* using the map interface in ipShield. To ensure plausibility of the shared location data the perturbation performed, or even the constant value provided, should conform to a map [37].

9 Concluding Remarks

While phones have evolved into sophisticated sensing platforms the corresponding sensing stack where starting at the raw sensor data meaningful data abstractions are created at each layer (akin to a communication stack) [26] has not yet taken shape. Efforts like CondOS [23], together with architectural changes such as dedicated co-processors for context detection [4] are steps in the direction towards introducing greater semantic clarity for shared data. With such a stack in place it is then a natural design choice to have a privacy system built within the OS itself exploiting the semantic granularity of data for improved privacy.

With ipShield we advocated the above design philosophy and took the first step towards creating a framework with architectural changes built within the Android OS to protect user privacy. We introduced better monitoring of accessed resources, proposed a user-understandable privacy abstraction in the form of possible inferences, allowed users to configure semantic privacy rules, and ensured that user preferences are securely enforced.

Orthogonal to the enforcement of rules is their creation. In future, to minimize user interaction in rule formulation it is imperative that systems are able to learn rules based on the semantic similarity of shared data and basic user preferences. With respect to granularity of rules, even with user participation privacy rules can often tend to become conservative impacting the app utility. To this end, careful integration of ipShield with various static analysis tools [32] could provide better insight into the working of apps and in the creation of balanced rules.

The other pertinent question is regarding the selection of a suitable set of privacy actions. Integration of cryptographic solutions would enrich the spectrum of available actions. In addition, currently ipShield does not handle traditional side-channels attacks and it will be an interesting extension to the current system.

Finally, any such system should be able to run on resource constrained platforms. Our experiments with ipShield indicate that it has low performance overhead and can run continuously on various mobile platforms without impacting app responsiveness.

Acknowledgement

We thank our shepherd Jonathan M. Smith, the anonymous reviewers, and our collaborators Haksoo Choi, Nicolas Bitouze and Lara Dolecek for their valuable comments. This work was supported in part by the U.S. ARL, U.K. Ministry of defense (MoD) under Agreement Number W911NF-06-3-0001, by the NSF under awards CNS-0910706 and CNS-1213140, by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via US Navy (USN) SPAWAR Systems Center Pacific (SSC-Pac). Any findings in this material are those of the author(s) and do not reflect the views of any of the above funding agencies. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] Android 4.3's Hidden App Permission Manager. http://www.androidpolice.com/2013/07/25/app-ops-android-4-3s-hidden-app-permission-manager-control-permissions-for-individual-apps/.

[2] Android Open Source Project. http://source.android.com/.

[3] Android Security Overview. http://source.android.com/devices/tech/security/.

[4] Apple M7 Co-Processor. http://en.wikipedia.org/wiki/Apple_M7.

[5] ipShield: A Framework For Enforcing Context-Aware Privacy. http://tinyurl.com/ipshieldgit.

[6] Pausing Google Play: More Than 100,000 Android Apps May Pose Security Risks. https://www.bit9.com/download/reports/Pausing-Google-Play-October2012.pdf.

[7] PDroid patch for Android Jelly Bean. http://github.com/gsbabil/PDroid-AOSP-JellyBean.

[8] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[9] RunKeeper. http://www.runkeeper.com/.

[10] Saga Lifelogging. http://www.getsaga.com/.

[11] Under the Hood: Dalvik patch for Facebook for Android. https://www.facebook.com/notes/facebook-engineering/under-the-hood-dalvik-patch-for-facebook-for-android/10151345597798920.

[12] AGARWAL, Y., AND HALL, M. Protectmyprivacy: Detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (2013), MobiSys '13, pp. 97–110.

[13] ASONOV, D., AND AGRAWAL, R. Keyboard acoustic emanations. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on* (2004), pp. 3–11.

[14] AVIV, A. J., SAPP, B., BLAZE, M., AND SMITH, J. M. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), ACSAC '12, pp. 41–50.

[15] BADEN, R., BENDER, A., SPRING, N., BHATTACHARJEE, B., AND STARIN, D. Persona: An online social network with user-defined privacy. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (2009), SIGCOMM '09, pp. 135–146.

[16] BAO, L., AND INTILLE, S. S. Activity recognition from user-annotated acceleration data. *Pervasive LNCS 3001* (2004), 1–17.

[17] BARRERA, D., KAYACIK, H. G., VAN OORSCHOT, P. C., AND SOMAYAJI, A. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 73–84.

[18] BERESFORD, A. R., RICE, A., SKEHIN, N., AND SOHAN, R. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* (2011), HotMobile '11, pp. 49–54.

[19] BROUWERS, N., AND WOEHRLE, M. Detecting dwelling in urban environments using gps, wifi, and geolocation measurements. In *Proc. 2nd Intl Workshop on Sensing Applications on Mobile Phones* (2011), pp. 1–5.

[20] CHAKRABORTY, S., BITOUZÉ, N., SRIVASTAVA, M., AND DOLECEK, L. Protecting data against unwanted inferences. In *Information Theory Workshop (ITW), 2013 IEEE* (2013), pp. 1–5.

[21] CHAKRABORTY, S., RAGHAVAN, K. R., JOHNSON, M. P., AND SRIVASTAVA, M. B. A framework for context-aware privacy of sensor data on mobile systems. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications* (2013), HotMobile '13, pp. 11:1–11:6.

[22] CHANG, D. F. K.-H., AND CANNY, J. Ammon: A speech analysis library for analyzing affect, stress, and mental health on mobile phones. *Proceedings of PhoneSense 2011* (2011).

[23] CHU, D., KANSAL, A., LIU, J., AND ZHAO, F. Mobile apps: it's time to move up to condos. HotOS.

[24] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10, pp. 1–6.

[25] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09, pp. 235–245.

[26] ESTRIN, D., AND SIM, I. Open mHealth Architecture: An Engine for Health Care Innovation. *Science 330*, 6005 (2010), 759–760.

[27] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11, pp. 627–638.

[28] FELT, A. P., HA, E., EGELMAN, S., HANEY, A., CHIN, E., AND WAGNER, D. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security* (2012), SOUPS '12, pp. 3:1–3:14.

[29] GENKIN, D., SHAMIR, A., AND TROMER, E. Rsa key extraction via low-bandwidth acoustic cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013.

[30] GÖTZ, M., NATH, S., AND GEHRKE, J. Maskit: Privately releasing user context streams for personalized mobile applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD '12, pp. 289–300.

[31] HAN, J., OWUSU, E., NGUYEN, L., PERRIG, A., AND ZHANG, J. Accomplice: Location inference using accelerometers on smartphones. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on* (2012), pp. 1–9.

[32] HOLAVANALLI, S., MANUEL, D., NANJUNDASWAMY, V., ROSENBERG, B., AND SHEN, F. Flow permissions for android. In *Tech Report*.

[33] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11, pp. 639–652.

[34] JEON, J., MICINSKI, K. K., VAUGHAN, J. A., REDDY, N., ZHU, Y., FOSTER, J. S., AND MILLSTEIN, T. Dr. android and mr. hide: Fine-grained security policies on unmodified android.

[35] KIM, D. H., KIM, Y., ESTRIN, D., AND SRIVASTAVA, M. B. Sensloc: Sensing everyday places and paths using less energy. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems* (2010), SenSys '10, pp. 43–56.

[36] KÖPF, B., AND BASIN, D. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (2007), CCS '07, pp. 286–296.

[37] KRUMM, J. A survey of computational location privacy. *Personal Ubiquitous Comput. 13*, 6 (Aug. 2009), 391–399.

[38] LEE, S., WONG, E. L., GOEL, D., DAHLIN, M., AND SHMATIKOV, V. πbox: A platform for privacy-preserving apps. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), NSDI'13, pp. 501–514.

[39] LIU, B., JIANG, Y., SHA, F., AND GOVINDAN, R. Cloud-enabled privacy-preserving collaborative learning for mobile sensing. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems* (2012), Sensys '12, pp. 57–70.

[40] LU, H., FRAUENDORFER, D., RABBI, M., MAST, M. S., CHITTARANJAN, G. T., CAMPBELL, A. T., GATICA-PEREZ, D., AND CHOUDHURY, T. Stresssense: Detecting stress in unconstrained acoustic environments using smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing* (2012), Ubicomp '12, pp. 351–360.

[41] LU, H., PAN, W., LANE, N. D., CHOUDHURY, T., AND CAMPBELL, A. T. Soundsense: Scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services* (2009), MobiSys '09, pp. 165–178.

[42] MARQUARDT, P., VERMA, A., CARTER, H., AND TRAYNOR, P. (sp)iphone: Decoding vibrations from nearby keyboards using mobile phone accelerometers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11, pp. 551–562.

[43] MILUZZO, E., VARSHAVSKY, A., BALAKRISHNAN, S., AND CHOUDHURY, R. R. Tapprints: Your finger taps have fingerprints. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (2012), MobiSys '12, pp. 323–336.

[44] MUN, M., HAO, S., MISHRA, N., SHILTON, K., BURKE, J., ESTRIN, D., HANSEN, M., AND GOVINDAN, R. Personal data vaults: A locus of control for personal data streams. In *Proceedings of the 6th International COnference* (2010), Co-NEXT '10, pp. 17:1–17:12.

[45] NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (2010), ACM, pp. 328–332.

[46] NIRJON, S., DICKERSON, R. F., ASARE, P., LI, Q., HONG, D., STANKOVIC, J. A., HU, P., SHEN, G., AND JIANG, X. Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (2013), MobiSys '13, pp. 403–416.

[47] PARATE, A., CHIU, M.-C., GANESAN, D., AND MARLIN, B. M. Leveraging graphical models to improve accuracy and reduce privacy risks of mobile sensing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services* (2013), MobiSys '13, pp. 83–96.

[48] PARK, J.-G., PATEL, A., CURTIS, D., TELLER, S., AND LEDLIE, J. Online pose classification and walking speed estimation using handheld devices. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing* (2012), UbiComp '12, pp. 113–122.

[49] PLARRE, K., RAIJ, A., HOSSAIN, S., ALI, A., NAKAJIMA, M., AL'ABSI, M., ERTIN, E., KAMARCK, T., KUMAR, S., SCOTT, M., SIEWIOREK, D., SMAILAGIC, A., AND WITTMERS, L. Continuous inference of psychological stress from sensory measurements collected in the natural environment. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on* (2011), pp. 97–108.

[50] RACHURI, K. K., MUSOLESI, M., MASCOLO, C., RENTFROW, P. J., LONGWORTH, C., AND AUCINAS, A. Emotionsense: a mobile phones based adaptive platform for experimental social psychology research. In *Proceedings of the 12th ACM international conference on Ubiquitous computing* (2010), pp. 281–290.

[51] RAHMAN, M. M., ALI, A. A., PLARRE, K., AL'ABSI, M., ERTIN, E., AND KUMAR, S. mconverse: Inferring conversation episodes from respiratory measurements collected in the field. In *Proceedings of the 2Nd Conference on Wireless Health* (2011), WH '11, pp. 10:1–10:10.

[52] RAIJ, A., GHOSH, A., KUMAR, S., AND SRIVASTAVA, M. Privacy risks emerging from the adoption of innocuous wearable sensors in the mobile environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2011), CHI '11, pp. 11–20.

[53] REDDY, S., MUN, M., BURKE, J., ESTRIN, D., AND HANSEN, MARK A ND SRIVASTAVA, M. Using mobile phones to determine transportation modes. *ACM Trans. Sen. Netw. 6*, 2 (Mar. 2010), 13:1–13:27.

[54] SONG, D. X., WAGNER, D., AND TIAN, X. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10* (2001), SSYM'01, pp. 25–25.

[55] THURM, S., AND KANE, Y. Your apps are watching you. The Wall Street Journal, 2012.

[56] TOOTOONCHIAN, A., SAROIU, S., GANJALI, Y., AND WOLMAN, A. Lockr: Better privacy for social networks. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies* (2009), CoNEXT '09, pp. 169–180.

[57] VUAGNOUX, M., AND PASINI, S. Compromising electromagnetic emanations of wired and wireless keyboards. In *Proceedings of the 18th Conference on USENIX Security Symposium* (2009), SSYM'09, pp. 1–16.

[58] ZHAN, A., CHANG, M., CHEN, Y., AND TERZIS, A. Accurate caloric expenditure of bicyclists using cellphones. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems* (2012), SenSys '12, pp. 71–84.

Building web applications on top of encrypted data using Mylar

Raluca Ada Popa, Emily Stark,† Jonas Helfer, Steven Valdez,
Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan
MIT CSAIL and †Meteor Development Group

ABSTRACT

Web applications rely on servers to store and process confidential information. However, anyone who gains access to the server (e.g., an attacker, a curious administrator, or a government) can obtain all of the data stored there. This paper presents Mylar, a platform for building web applications, which protects data confidentiality against attackers with *full access to servers*. Mylar stores sensitive data encrypted on the server, and decrypts that data only in users' browsers. Mylar addresses three challenges in making this approach work. First, Mylar allows the server to perform keyword search over encrypted documents, even if the documents are encrypted with *different* keys. Second, Mylar allows users to share keys and encrypted data securely in the presence of an active adversary. Finally, Mylar ensures that client-side application code is authentic, even if the server is malicious. Results with a prototype of Mylar built on top of the Meteor framework are promising: porting 6 applications required changing just 36 lines of code on average, and the performance overheads are modest, amounting to a 17% throughput loss and a 50 ms latency increase for sending a message in a chat application.

1 INTRODUCTION

Using a web application for confidential data requires the user to trust the server to protect the data from unauthorized disclosures. This trust is often misplaced, however, because there are many ways in which confidential data could leak from a server. For example, attackers could exploit a vulnerability in the server software to break in [42], a curious administrator could peek at the data on the server [9, 10], or the server operator may be compelled to disclose data by law [20]. Is it possible to build web applications that protect data confidentiality against attackers with *full access* to servers?

A promising approach is to give each user their own encryption key, encrypt a user's data with that user's key in the web browser, and store only encrypted data on the server. This model ensures that an adversary would not be able to read any confidential information on the server, because they would lack the necessary decryption keys. In fact, this model has been already adopted by some privacy-conscious web applications [28, 40].

Unfortunately, this approach suffers from three significant security, functionality, and efficiency shortcomings. First, a compromised server could provide malicious client-side code to the browser and extract the user's key and data. Ensuring that the server did not tamper with the application code is difficult because a web application consists of many files, such as HTML pages, Javascript code, and CSS style sheets, and the HTML pages are often dynamically generated.

Second, this approach does not provide data sharing between users, a crucial function of web applications. To address this problem, one might consider encrypting shared documents with separate keys, and distributing each key to all users sharing a document via the server. However, distributing keys via the server is challenging because a compromised server can supply arbitrary keys to users, and thus trick a user into using incorrect keys.

Third, this approach requires that all of the application logic runs in a user's web browser because it can decrypt the user's encrypted data. But this is often impractical: for instance, doing a keyword search would require downloading all the documents to the browser.

This paper presents Mylar, a new platform for building web applications that stores only encrypted data on the server. Mylar makes it practical for many classes of applications to protect confidential data from compromised servers. It leverages the recent shift in web application frameworks towards implementing logic in client-side Javascript code, and sending data, rather than HTML, over the network [29]; such a framework provides a clean foundation for security. Mylar addresses the challenges mentioned above with a combination of systems techniques and novel cryptographic primitives, as follows.

Data sharing. To enable sharing, each sensitive data item is encrypted with a key available to users who share the item. To prevent the server from cheating during key distribution, Mylar provides a mechanism for establishing the correctness of keys obtained from the server: Mylar forms certificate paths to attest to public keys, and allows the application to specify what certificate paths can be trusted in each use context. In combination with a user interface that displays the appropriate certificate components to the user, this technique ensures that even

a compromised server cannot trick the application into using the wrong key.

Computing over encrypted data. Keyword search is a common operation in web applications, but it is often impractical to run on the client because it would require downloading large amounts of data to the user's machine. While there exist practical cryptographic schemes for keyword search, they require that data be encrypted with a single key. This restriction makes it difficult to apply these schemes to web applications that have many users and hence have data encrypted with many different keys.

Mylar provides the first cryptographic scheme that can perform keyword search efficiently over data encrypted with *different* keys. The client provides an encrypted word to the server and the server can return all documents that contain this word, without learning the word or the contents of the documents.

Verifying application code. With Mylar, code running in a web browser has access to the user's decrypted data and keys, but the code itself comes from the untrusted server. To ensure that this code has not been tampered with, Mylar checks that the code is properly signed by the web site owner. This checking is possible because application code and data are separate in Mylar, so the code is static. Mylar uses two origins to simplify code verification for a web application. The primary origin hosts only the top-level HTML page of the application, whose signature is verified using a public key found in the server's X.509 certificate. All other files come from a secondary origin, so that if they are loaded as a top-level page, they do not have access to the primary origin. Mylar verifies the hash of these files against an expected hash contained in the top-level page.

To evaluate Mylar's design, we built a prototype on top of the Meteor web application framework [29]. We ported 6 applications to protect confidential data using Mylar: a medical application for endometriosis patients, a web site for managing homework and grades, a chat application called kChat, a forum, a calendar, and a photo sharing application. The endometriosis application is used to collect data from patients with that medical condition, and was designed under the aegis of the MIT Center for Gynepathology Research by surgeons at the Newton-Wellesley hospital (affiliated with the Harvard Medical School) in collaboration with biological engineers at MIT; the Mylar-secured version is currently being tested by patients and is undergoing IRB approval before deployment.

Our results show that Mylar requires little developer effort: we had to modify an average of just 36 lines of code per application. We also evaluated the performance of Mylar on three of the applications above. For example, for kChat, our results show that Mylar incurs modest overheads: a 17% throughput reduction and a 50 msec latency increase for the most common operation (sending a message). These results suggest that Mylar is a good fit for multi-user web applications with data sharing.

2 Related work

Mylar is the first system to protect data confidentiality in a wide range of web applications against arbitrary server compromises. In the rest of this section, we relate Mylar to prior work on securing web applications, building systems using untrusted servers, and computing over encrypted data.

2.1 Web application security

Much of the work on web application security focuses on preventing security vulnerabilities caused by bugs in the application's source code, either by statically checking that the code follows a security policy [11, 44], or by catching policy violations at runtime [18, 24, 46]. In contrast, Mylar assumes that *any* part of the server can be compromised, either as a result of software vulnerabilities or because the server operator is untrustworthy, and protects data confidentiality in this setting.

On the browser side, prior work has explored techniques to mitigate vulnerabilities in Javascript code that allow an adversary to leak data or otherwise compromise the application [1, 16, 45]. Mylar assumes that the developer does not inadvertently leak data from client-side code, but in principle could be extended to use these techniques for dealing with buggy client-side code.

There has been some work on using encryption to protect confidential data in web applications, as we describe next. Unlike Mylar, none of them can support a wide range of complex web applications, nor compute over encrypted data at the server, nor address the problem of securely managing access to shared data.

A position paper by Christodorescu [12] proposes encrypting and decrypting data in a web browser before sending it to an untrusted server, but lacks any details of how to build a practical system.

Several data sharing sites encrypt data in the browser before uploading it to the server, and decrypt it in the browser when a user wants to download the data [14, 28, 35]. The key is either stored in the URL's hash fragment [28, 35], or typed in by the user [14], and both the key and data are accessible to any Javascript code from the page. As a result, an active adversary could serve Javascript code to a client that leaks the key. In contrast, Mylar's browser extension verifies that the client-side code has not been tampered with.

Several systems transparently encrypt and decrypt data sent to a server [7, 13, 33, 34]. These suffer from the same problems as above: they cannot handle active attacks, and

cannot compute over encrypted data at the server without revealing a significant amount of information.

Cryptocat [40], an encrypted chat application, distributes the application code as a browser extension rather than a web application, in order to deal with active attacks [39]. Mylar's browser extension is general-purpose: it allows verifying the code of web applications without requiring users to install a *separate* extension for each application. Cryptocat could also benefit from Mylar's search scheme to perform keyword search over encrypted data at the server.

2.2 Untrusted servers

SUNDR [25] protects file system integrity, providing fork consistency in the face of a malicious server. SPORC [15] and Depot [27] extend SUNDR's design to build applications on top of an encrypted server. For example, SPORC provides conflict resolution using operational transforms, and consistently handles access control changes. These systems do not allow an application to perform server-side computation, such as Mylar's server-side keyword search. Furthermore, with SPORC, the application logic is hard-coded into the client, whereas with Mylar, the application logic is determined at runtime, based on the URL that the user visits.

CryptDB [32] protects confidential data in a SQL database server by running SQL queries over encrypted data. However, in a typical database-backed web application, the application server gets access to unencrypted data, and receives each user's key when the user logs in. Consequently, while CryptDB protects against attacks on the database server, it provides no guarantees for users logged in during an attack on the application server. For example, if an administrator with access to all data is logged in when the application server is compromised, then the attacker can compromise all data. Finally, CryptDB cannot compute over data encrypted with different keys as in Mylar's multi-key keyword search. On the other hand, CryptDB allows computing more functions over encrypted data than Mylar.

2.3 Computation over encrypted data

Theoretical results on fully homomorphic encryption and functional encryption have shown that it is possible for an untrusted server to compute arbitrary functions over encrypted data [17, 19, 26]. However, such general-purpose schemes are too slow to be practical.

Many schemes for performing keyword search over encrypted data have been proposed [21, 36]. All of these schemes for keyword search have assumed that the data is encrypted with a single key; Mylar provides the first practical scheme for performing keyword search over data encrypted with different keys. Proxy re-encryption [3] allows switching the key under which some data is encrypted in the context of public-key encryption, but this does not provide an efficient search scheme.

2.4 Trusted hardware

An alternative approach for computing over encrypted data is to rely on trusted hardware [2, 4, 22]. Such approaches are complementary to Mylar, and could be used to extend the kinds of computations that Mylar can perform over encrypted data at the server, as long as the application developer and the users believe that trusted hardware is trustworthy.

3 MYLAR ARCHITECTURE

There are three different parties in Mylar: the users, the web site owner, and the server operator. Mylar's goal is to help the site owner protect the confidential data of users in the face of a malicious or compromised server operator.

3.1 System overview

Mylar embraces the trend towards client-side web applications; Mylar's design is suitable for platforms that:

1. Enable client-side computation on data received from the server.
2. Allow the client to intercept data going to the server and data coming from the server.
3. Separate application code from data, so that the HTML pages supplied by the server are static.

AJAX web applications with a unified interface for sending data over the network, such as Meteor [29], fit this model. Such frameworks provide a clean foundation for security, because they send data separately from the HTML page that presents the data. In contrast, traditional server-side frameworks incorporate dynamic data into the application's HTML page in arbitrary ways, making it difficult to encrypt and decrypt the dynamic data on each page while checking that the fixed parts of the page have not been tampered with [37].

3.1.1 Mylar's components

The architecture of Mylar is shown in Figure 1. Mylar consists of the four following components:

Browser extension. It is responsible for verifying that the client-side code of a web application that is loaded from the server has not been tampered with.

Client-side library. It intercepts data sent to and from the server, and encrypts or decrypts that data. Each user has a private-public key pair. The client-side library stores the private key of the user at the server, encrypted with the user's password.[1] When the user logs in, the client-side

[1] The private key can also be stored at a trusted third-party server, to better protect it from offline password guessing attacks and to recover from forgotten passwords without re-generating keys.

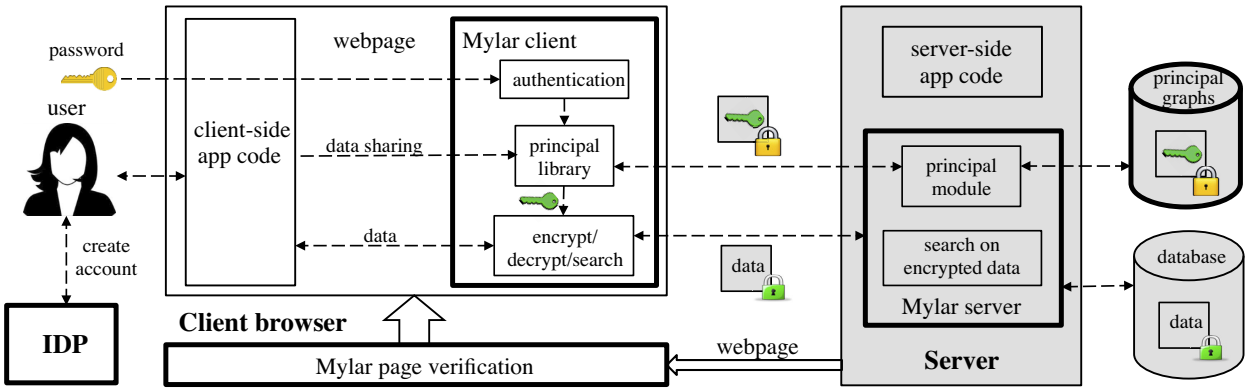


Figure 1: System overview. Shaded components have access only to encrypted data. Thick borders indicate components introduced by Mylar.

library fetches and decrypts the user's private key. For shared data, Mylar's client creates separate keys that are also stored at the server in encrypted form.

Server-side library. It performs computation over encrypted data at the server. Specifically, Mylar supports keyword search over encrypted data, because we have found that many applications use keyword search.

Identity provider (IDP). For some applications, Mylar needs a trusted identity provider service (IDP) to verify that a given public key belongs to a particular username. An application needs the IDP if the application has no trusted way of verifying the users who create accounts, and the application allows users to choose whom to share data with. For example, if Alice wants to share a sensitive document with Bob, Mylar's client needs the public key of Bob to encrypt the document. A compromised server could provide the public key of an attacker, so Mylar needs a way to verify the public key. The IDP helps Mylar perform this verification by signing the user's public key and username. An application does not need the IDP if the site owner wants to protect against only passive attacks (§3.4), or if the application has a limited sharing pattern for which it can use a static root of trust (see §4.2).

An IDP can be shared by many applications, similar to an OpenID provider [30]. The IDP does not store per-application state, and Mylar contacts the IDP only when a user first creates an account in an application; afterwards, the application server stores the certificate from the IDP.

3.2 Mylar for developers

The developer starts with a regular (non-encrypted) web application implemented in Mylar's underlying web platform (Meteor in our prototype). To secure this application with Mylar, a developer uses Mylar's API (Figure 2), as we explain in the rest of this paper. First, the developer uses Mylar's authentication library for user login and account creation. If the application allows a user to choose what other users to share data with, the developer should also specify the URL and public key of a trusted IDP.

Second, the developer specifies which data in the application should be encrypted, and who should have access to it. Mylar uses principals for access control; a principal corresponds to a public/private key pair, and represents an application-level access control entity, such as a user, a group, or a shared document. In our prototype, all data is stored in MongoDB collections, and the developer annotates each collection with the set of fields that contain confidential data and the name of the principal that should have access to that data (i.e., whose key should be used).

Third, the developer specifies which principals in the application have access to which other principals. For example, if Alice wants to invite Bob to a confidential chat, the application must invoke the Mylar client to grant Bob's principal access to the chat room principal.

Fourth, the developer changes their server-side code to invoke the Mylar server-side library when performing keyword search. Our prototype's client-side library provides functions for common operations such as keyword search over a specific field in a MongoDB collection.

Finally, as part of installing the web application, the site owner generates a public/private key pair, and signs the application's files with the private key using Mylar's bundling tool. The web application must be hosted using `https`, and the site owner's public key must be stored in the web server's X.509 certificate. This ensures that even if the server is compromised, Mylar's browser extension will know the site owner's public key, and will refuse to load client-side code if it has been tampered with.

3.3 Mylar for users

To obtain the full security guarantees of Mylar, a user must install the Mylar browser extension, which detects tampered code. However, if a site owner wants to protect against only passive attacks (§3.4), users don't have to install the extension and their browsing experience is entirely unchanged.

Function	Semantics
idp_config(*url*, *pubkey*)	Declares the *url* and *pubkey* of the IDP and returns the principal corresponding to the IDP.
create_user(*uname*, *password*, *auth_princ*)	Creates an account for user *uname* which is certified by principal *auth_princ*.
login(*uname*, *password*)	Logs in user *uname*.
logout()	Logs out the currently logged-in user.
collection.**encrypted**({*field*: *princ_field*}, ...)	Specify that *field* in *collection* should be encrypted for the principal in *princ_field*.
collection.**auth_set**([*princ_field*, *fields*], ...)	Authenticate the set of *fields* with principal in *princ_field*.
collection.**searchable**(*field*)	Mark *field* in *collection* as searchable.
collection.**search**(*word*, *field*, *princ*, *filter*, *proj*)	Search for *word* in *field* of *collection*, filter results by *filter* and project only the fields in *proj* from the results. Use *princ*'s key to generate the search token.
princ_create(*name*, *creator_princ*)	Create principal named *name*, sign the principal with *creator_princ*, and give *creator_princ* access to it.
princ_create_static(*name*, *password*)	Create a static principal called *name*, hardcode it in the application, and wrap its secret keys with *password*.
princ_static(*name*, *password*)	Return the static principal *name*; if a correct password is specified, also load the secret keys for this principal.
princ_current()	Return the principal of currently logged in user.
princ_lookup($name_1$, ..., $name_k$, *root*)	Look up principal named $name_1$ as certified by a chain of principals named $name_i$ rooted in *root* (e.g., the IDP).
granter.**add_access**(*grantee*)	Give the *grantee* principal access to the *granter* principal.
grantee.**allow_search**(*granter*)	Allow matching keywords from *grantee* on *granter*'s data.

Figure 2: Mylar API for application developers split in three sections: authentication, encryption/integrity annotations, and access control. All of the functions except `princ_create_static` and `searchable` run in the client browser. This API assumes a MongoDB storage model where data is organized as collections of documents, and each document consists of fieldname-and-value pairs. Mylar also preserves the generic functionality for unencrypted data of the underlying web framework.

3.4 Threat model

Threats. Both the application and the database servers can be *fully* controlled by an adversary: the adversary may obtain all data from the server, cause the server to send arbitrary responses to web browsers, etc. This model subsumes a wide range of real-world security problems, from bugs in server software to insider attacks.

Mylar also allows some user machines to be controlled by the adversary, and to collude with the server. This may be either because the adversary is a user of the application, or because the adversary broke into a user's machine.

We call this adversary *active*, in contrast to a *passive* adversary that eavesdrops on all information at the server, but does not make any changes, so that the server responds to all client requests as if it were not compromised.

Guarantees. Mylar protects a data item's confidentiality in the face of arbitrary server compromises, as long as none of the users with access to that data item use a compromised machine. Mylar does not hide data access patterns, or communication and timing patterns in an application. Mylar provides data authentication guarantees, but does not guarantee the freshness or correctness of results from the computation at the server.

Assumptions. To provide the above guarantees, Mylar makes the following assumptions. Mylar assumes that the web application as written by the developer will not send user data or keys to untrustworthy recipients, and cannot be tricked into doing so by exploiting bugs (e.g., cross-site scripting). Our prototype of Mylar is built on top of Meteor, a framework that helps programmers avoid many common classes of bugs in practice.

Mylar also assumes that the IDP correctly verifies each user's identity (e.g., email address) when signing certificates. To simplify the job of building a trustworthy IDP, Mylar does not store any application state at the IDP, contacts the IDP only when a user first registers, and allows the IDP to be shared across applications.

Finally, Mylar assumes that the user checks the web browser's security indicator (e.g., the `https` shield icon) and the URL of the web application they are using, before entering any sensitive data. This assumption is identical to what users must already do to safely interact with a *trusted* server. If the user falls for a phishing attack, neither Mylar

nor a trusted server can prevent the user from entering confidential data into the adversary's web application.

3.5 Security overview

At a high level, Mylar achieves its goal as follows. First, it verifies the application code running in the browser (§6), so that it is safe to give client-side code access to keys and plaintext data. Then, the client code encrypts the data marked sensitive before sending it to the server. Since users need to share data, Mylar provides a mechanism to securely share and look up keys among users (§4). Finally, to perform server-side processing, Mylar introduces a new cryptographic scheme that can perform keyword search over documents encrypted with many different keys, without revealing the content of the encrypted documents or the word being searched for (§5).

4 Sharing data between users

Many web applications share data between users according to some policy. A simple example is a chat application, where messages are shared between the sender and the recipients. In Mylar's threat model, an application cannot trust the server to enforce the sharing policy, because the server is assumed to be compromised. As a result, the application must encrypt shared data using a key that will be accessible to just the right set of users.

Mylar allows an application to specify its security policy in terms of application-defined principals. In particular, each principal has an application-chosen *name*, a *public key* used to encrypt data for that principal, and a *private key* used to decrypt that principal's data.

In addition to allowing the application to create principals, and to use the principals' keys to encrypt and decrypt data, Mylar provides two critical operations to the application for managing principals:

- Find a principal so that the application can use the corresponding private key to decrypt data. The goal is to ensure that only authorized users can get access to the appropriate private key.
- Find a principal so that the application can use the corresponding public key to encrypt or share data with other users. The goal is to ensure that a malicious server cannot trick Mylar into returning the wrong public key, which could lead the application to share confidential data with the adversary.

Mylar cryptographically enforces the above goals by forming two graphs on top of principals: an *access graph*, which uses key chains to distribute the private keys of shared principals to users, and a *certification graph*, which uses certificate chains to attest to the mapping between a principal name and its public key.

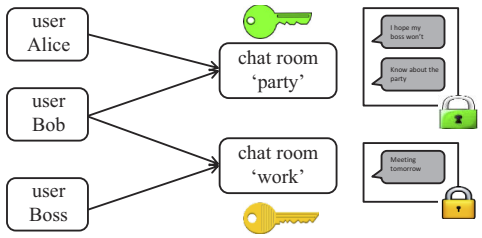


Figure 3: Example access graph for a chat application. Rounded rectangles represent principals, and arrows represent access relationships. Alice and Bob share the chat room "party" so they both have access to the principal for this room. Messages in each chat room are encrypted with the key of the room's principal.

4.1 Access graph

To ensure that only authorized users can access the private key of a principal, Mylar requires the application to express its access control policy in terms of *access* relationships between principals. Namely, if principal *A* can access principal *B*'s private key, then we say *A has access to B*. The *has access to* relation is transitive: if *B* in turn has access to *C*, then *A* can access *C*'s private key as well. To express the application's policy in the access graph, the application must create appropriate *has access to* relationships between principals. The application can also create intermediate principals to represent, say, groups of users that all should have access to the same private keys.

As an example, consider a chat application where messages in each chat room should be available only to that room's participants. Figure 3 shows the access graph for this scenario. Both Alice and Bob have access to the key encrypting the "party" room, but the boss does not.

Key chaining. To enforce the access graph cryptographically, Mylar uses key chaining, as in CryptDB [32]. When an application asks to add a new *has access to* edge from principal *A* to principal *B*, Mylar creates a *wrapped key*: an encryption of *B*'s private keys under the public key of principal *A*. This ensures that a user with access to *A*'s private key can decrypt the wrapped key and obtain *B*'s private key. For example, in Figure 3, the private key of the "party" chat room is encrypted under the public key of Alice, and separately under the public key of Bob as well. The server stores these wrapped keys, which is safe since the keys are encrypted.

In practice, *has access to* relationships are rooted in user principals, so that a user can gain access to all of their data when they initially log in and have just the private key of their own user principal. When Mylar needs to decrypt a particular data item, it first looks up that data item's principal, as specified by the **encrypted** annotation (Figure 2). Mylar then searches for a chain of wrapped keys, starting from the principal of the currently logged in user, and leading to the data item's principal.

4.2 Certification graph

Mylar applications must look up public keys of principals when sharing data, for two broad purposes: either to encrypt data with that key, or to give some principal access to that key. In both cases, if a compromised server tricks the client application into using the public key of the adversary, the adversary will gain access to confidential data. For example, in the chat example, suppose Bob wants to send a confidential message to the "work" chat room. If the server supplies the adversary's public key for the chat room principal and the application client uses it, the adversary will be able to decrypt the message. Preventing such attacks is difficult because all of the wrapped keys are stored at the server, and the server may be malicious.

To prevent such attacks, Mylar relies on a certification graph, which allows one principal to vouch for the name and the public key of another principal. The nodes of this graph are principals from the access graph together with some *authority* principals, which are principals providing the root of trust (described in §4.3). Applications create certificate chains for principals, rooted in an authority principal. For instance, in the chat example, the application can sign the "chatroom:work" principal with the key of the "user:boss" principal that created the chat room. Using the certification graph, applications can look up the public key of a principal by specifying the name of the principal they are looking for, along with a chain of certifications they expect to find.

Since the server is not trusted, there is no single authority to decide on the public key for a given principal name: in our chat example, both the real boss and a malicious server may have created chat rooms named "work." To prevent such naming ambiguity, one approach is to display the names in a certification chain to the user, similar to how web browsers display the hostname from an X.509 certificate for `https` web sites. As we describe later in §8, if the chat application displays the email address of the chat room creator (who signed the chat room principal), in addition to the name of the chat room, the user could distinguish a correct "work" chat room, created by the boss, from an impostor created by an attacker. This requires Mylar applications to unambiguously map human-meaningful names, such as the "work" chat room and the identity of the Boss user, onto principal names, such as "chatroom:work" and "user:boss."

Mylar's certificate chains are similar to X.509; the difference is that X.509 typically has fixed roots of trust and fixed rules for what certificate chains are allowed, whereas Mylar allows the application to specify different roots of trust and acceptable chains for each lookup.

4.3 Principals providing the root of trust

The authority principals can be either the IDP or *static principals*. Static principals are access control entities fixed in the application's logic. For example, the endometriosis medical application has a group called "surgeons" representing the surgeons that have access to all patient data. Similarly, the homework submission application has a group called "staff" representing staff members with access to all student homework submissions and grades. In these applications, static principals can altogether remove the need for an IDP.

A developer can create a static principal by running **princ_create_static**(*name*, *password*) with the help of a command-line tool. This generates fresh keys for a principal, and encrypts the secret keys with *password*, so they can be retrieved only by providing *password* to **princ_static**. The resulting public key and encrypted secret key are hardcoded into the application's source code. This allows the application to refer to the static principal by name without relying on the IDP.

Static principals can also certify other principals. For example, in the endometriosis application, all user accounts are manually created by surgeons. This allows all user principals to be certified by the static "surgeons" principal, avoiding the need for an IDP to do the same.

4.4 User principals

To create an account for a new user, the application must invoke **create_user**, as shown in Figure 2. This causes the Mylar client to generate a new principal for the user, encrypt the secret key with the user's password, and store the principal with the encrypted secret key on the server.

To enable the application to later look up this user's public key, in the presence of active adversaries, the principal must be certified. To do this, the application supplies the *auth_princ* argument to **create_user**. This is typically either a static principal or the IDP. For static principals, the certificate is generated directly in the browser that calls **create_user**; the creator must have access to the private key of *auth_princ*. For example, the endometriosis application, where all users are manually created by a surgeon, follows this model. If *auth_princ* is the IDP, the Mylar client interprets *uname* as the user's email address, and contacts the IDP, which verifies the user's email address and signs a certificate containing the user's public key and email address.

Even though multiple applications can share the IDP, a buggy or malicious application will not affect other applications that use the same IDP (unless users share passwords across applications). This property is ensured by never sending passwords or secret keys to the IDP, and explicitly including the application's origin in the certificate generated by the IDP.

4.5 Data integrity

To prevent an attacker from tampering with the data, Mylar provides two ways to authenticate data, as follows.

First, all encrypted data is authenticated with a MAC (message authentication code),[2] which means that clients will detect any tampering with the ciphertext. However, an adversary can still replace the ciphertext of one field in a document with any other ciphertext that was encrypted using the same key.

To protect against such attacks, developers can specify an *authentication set* of fields whose values must be consistent with one other, using the **auth_set** annotation. This annotation guarantees that if a client receives some document, then all fields in each authentication set were consistent at some point, according to the corresponding principal. Mylar enforces authentication sets by computing a MAC over the values of all fields in each set.

For example, in a chat room application, each message has several fields, including the message body and the (client-generated) timestamp. By putting these two fields into an authentication set, the developer ensures that an adversary cannot splice together the body of one message with the timestamp from another message.

Mylar does not guarantee data freshness, or correctness of query results. An adversary can roll back the entire authentication set to an earlier version without detection, but cannot roll back a *subset* of an authentication set.

5 COMPUTING ON ENCRYPTED DATA

The challenge facing Mylar in computing over encrypted data is that web applications often have many users, resulting in data encrypted with many different keys. Existing efficient encryption schemes for computation over encrypted data, such as keyword search, assume that all data is encrypted with a single key [21, 36]. Using such a scheme in Mylar would require computation over one key at a time, which is inefficient.

For example, consider a user with access to N documents, where each document is encrypted with a different key (since it can be shared with a different set of users). Searching for a keyword in all of these documents would require the user to generate N distinct cryptographic search tokens, and to send all of them to the server. Even for modest values of N, such as 1000, this can result in noticeable computation and network costs for the user's machine. Moreover, if the N keys are not readily available in the client browser, fetching these keys may bring further overhead.

To address this limitation, Mylar introduces a *multi-key search* scheme, as described in the rest of this section.

5.1 Multi-key search

Mylar's multi-key search scheme provides a simple abstraction. If a user wants to search for a word in a set of documents on a server, each encrypted with a different key, the user's machine needs to provide only a single search token for that word to the server. The server, in turn, returns each encrypted document that contains the user's keyword, as long as *the user has access* to that document's key.

The intuition for our scheme is as follows. Say that the documents that a user has access to are encrypted under keys $k_1, \ldots, k_n$ and the user's own key is uk. The user's machine computes a search token for a word w using key uk, denoted tk^w_{uk}. If the server had $tk^w_{k_1}, \ldots, tk^w_{k_n}$ instead of tk^w_{uk}, the server could match the search token against the encrypted documents using an existing searchable encryption scheme.

Our idea is to enable the server *to compute these tokens by itself*; that is, to adjust the initial tk^w_{uk} to $tk^w_{k_i}$ for each i. To allow the server to perform the adjustment, the user's machine must initially compute *deltas*, which are cryptographic values that enable a server to adjust a token from one key to another key. We use $\Delta_{uk \to k_i}$ to denote the delta that allows a server to adjust tk^w_{uk} to $tk^w_{k_i}$. These deltas represent the user's access to the documents, and crucially, these deltas can be reused for every search, so the user's machine needs to generate the deltas only once. For example, if Alice has access to Bob's data, she needs to provide one delta to the server, and the server will be able to adjust all future tokens from Alice to Bob's key.

In terms of security, our scheme guarantees that the server does not learn the word being searched for, and does not learn the content of the documents. All that the server learns is whether the word in the search token matched some word in a document, and in the case of repeated searches, whether two searches were for the same word. Knowing which documents contain the word being searched for is desirable in practice, to avoid the overhead of returning unnecessary documents.

This paper presents the multi-key search scheme at a high level, with emphasis on its interface and security properties as needed in our system. We provide a rigorous description and a cryptographic treatment of the scheme (including formal security definitions and proofs) in a technical report [31]. Readers that are not interested in cryptographic details can skip to §5.3.

5.2 Cryptographic construction

We construct the multi-key search scheme using bilinear maps on elliptic curves, which, at a high level, are functions $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ are special groups of prime order p on elliptic curves. Let g be a generator of $\mathbb{G}_2$. Let H and H_2 be certain hash functions on the elliptic curves. e has the property that $e(H(w)^a, g^b) = e(H(w), g)^{ab}$. Figure 4 shows pseudocode for our multi-key search scheme.

[2] For efficiency, Mylar uses authenticated encryption, which conceptually computes both the ciphertext and the MAC tag in one pass.

Client-side operations:

procedure KEYGEN() ▷ Generate a fresh key
  $key \leftarrow$ random value from $\mathbb{Z}_p$
  return *key*

procedure ENC(*key*, *word*)
  $r \leftarrow$ random value from $\mathbb{G}_T$
  $c \leftarrow \langle r, H_2(r, e(H(word), g)^{key})\rangle$
  return *c*

procedure TOKEN(*key*, *word*)
  ▷ Generate search token for matching *word*
  $tk \leftarrow H(word)^{key}$ in $\mathbb{G}_1$
  return *tk*

procedure DELTA(key_1, key_2)
  ▷ Allow adjusting search token from key_1 to key_2
  $\Delta_{key_1 \to key_2} \leftarrow g^{key_2/key_1}$ in $\mathbb{G}_2$
  return $\Delta_{key_1 \to key_2}$

Server-side operations:

procedure ADJUST(tk, $\Delta_{k_1 \to k_2}$)
  ▷ Adjust search token *tk* from k_1 to k_2
  $atk \leftarrow e(tk, \Delta_{k_1 \to k_2})$ in $\mathbb{G}_T$
  return *atk*

procedure MATCH(atk, $c = \langle r, h\rangle$)
  ▷ Return whether *c* and *atk* refer to same word
  $h' \leftarrow H_2(r, atk)$
  return $h' \stackrel{?}{=} h$

Figure 4: Pseudo-code for Mylar's multi-key search scheme.

5.3 Indexing search

One efficiency issue with this algorithm is that the server has to scan through every word of every document to identify a match. This can be slow if the documents are large, but is unavoidable if the encryption of each word is randomized with a different r, as in Figure 4.

To enable the construction of an efficient index over the words in a searchable document, Mylar supports an indexable version of this multi-key search scheme. The idea is to remove randomness without compromising security. Intuitively, randomness is needed to hide whether two words encrypted under the same key are equal. But for words within one document, Mylar can remove the duplicates at the time the document is encrypted, so per-word randomness is not needed within a document.

Therefore, to encrypt a document consisting of words $w_1, \ldots, w_n$, the client removes duplicates, chooses one random value r, and then uses the same r when encrypting each of the words using ENC().

When searching for word w in a document, the server performs the adjustment as before and obtains *atk*. It then computes $v \leftarrow \text{COMBINE}(r, atk) = \langle r, H_2(r, atk)\rangle$ using the document's randomness r. If one of the words in the document is w, its encryption will be equal to v, because they use the same randomness r. Therefore, the server can perform direct equality checks on encrypted words. This means that it can build an index over the encrypted words in the document (e.g., a hash table), and then use that index and v to figure out in constant time if there is a match without scanning the document.

A limitation is that the server has to use an index per unique key rather than one holistic index.

5.4 Integrating search with the principal graph

Mylar integrates the multi-key search scheme with the principal graph as follows. When a principal P is created, Mylar generates a key k_P using KEYGEN (Figure 4). Whenever P receives access to some new principal A, Mylar includes k_A in the wrapped key for P. The first time a user with access to P comes online, the Mylar client in that user's browser retrieves k_A from the wrapped key, computes $\Delta_{k_P \to k_A} \leftarrow$DELTA($k_P$, k_A), and stores it at the server. This delta computation happens just once for a pair of principals.

To encrypt a document for some principal A, the user's browser encrypts each word w in the document separately using ENC(k_A, w). Since the multi-key search scheme does not support decryption, Mylar encrypts all searchable documents twice: once with the multi-key search scheme, for searching, and once with a traditional encryption scheme like AES, for decryption.

To search for a word w with principal P, the user's client uses TOKEN(k_P, w) to compute a token *tk*, and sends it to the server. To search over data encrypted for principal A, the server obtains $\Delta_{k_P \to k_A}$, and uses ADJUST(tk, $\Delta_{k_P \to k_A}$) to adjust the token from k_P to k_A, obtaining the adjusted token atk_A. Then, for each document encrypted under k_A with randomness r, the server computes $v \leftarrow \text{COMBINE}(r, atk_A)$ and checks if v exists in the document using an index. The server repeats the same process for all other principals that P has access to.

Integrating the access graph with keyword search brings up two challenges. The first comes from the fact that our multi-key search scheme allows adjusting tokens just once. In the common case of an access graph where all paths from a user to the data's encryption key consist of one edge (such as the graph in Figure 3), Mylar associates the search delta with the edge, and stores it along with the wrapped key. In our chat example, this allows a user's browser to search over all chat rooms that the user has access to, by sending just one search token.

Some applications can have a more complex access graph. For example, in the endometriosis application, all doctors have access to the *staff* principal, which in turn has access to all patient principals. Here, the optimal approach is to use the ADJUST() function on the server between principals with the largest number of edges, so as to maximize the benefit of multi-key search. For instance,

if a doctor wanted to search over patient records, the doctor's browser should fetch the *staff* principal it has access to, and produce a search token using the *staff* principal's private key. The server would then use ADJUST() to look for matches in documents encrypted with each patient's key. Because most of our applications have simple access graphs, our prototype does not automate this step, and a developer must choose the principal with which to search.

The second challenge comes from the fact that searching over data supplied by an adversary can leak the word being searched for. For example, suppose an adversary creates a document containing all the words in a dictionary, and gives the user access to that document. If the user searches for a word *w* in all of the documents he has access to, including the one from the adversary, the server will see which of the words in the adversary's document matches the user's token, and hence will know which dictionary word the user searched for. To prevent this, users must explicitly *accept* access to a shared document, and developers must invoke the **allow_search** function, provided by Mylar for this purpose, as appropriate.

6 VERIFYING CLIENT-SIDE CODE

Although Mylar uses encryption to protect confidential data stored on the untrusted server, the cryptographic keys and the plaintext data are both available to code executing in the user's web browser. The same-origin policy [47] ensures that applications from *other* origins running in the browser do not access the data in the Mylar application. However, Mylar must also ensure that code running in the application's origin has not been tampered with.

Since the code in a web page is static in Mylar, a strawman solution is to sign this code and verify the signature in the browser. The strawman does not suffice because of a combination of two factors. On the one hand, most web applications (including those using Mylar) consist of multiple files served by the web server. On the other hand, the only practical way to control what is loaded in a browser is to interpose on individual HTTP requests.

The problem arises because at the level of individual HTTP requests, it is difficult to reason about what code the browser will execute. For example, if an image is loaded in the context of an `<IMG SRC=...>` tag, it will not execute Javascript code. But if the same image is loaded as a top-level page, the browser's content-sniffing algorithm may decide the file is actually HTML, and potentially execute Javascript code embedded in the image [6]. Thus, a well-meaning developer must be exceedingly careful when including any content, such as images, in their web application. If the developer inadvertently includes a malicious image file in the application, an adversary can cause the browser to load that file as a top-level page [5] and trigger this attack. Similar problems can arise with other content types, including CSS style sheets, PDF files, etc.

```
procedure PROCESSRESPONSE(url, cert, response)
                                  ▷ url is the requested URL
                          ▷ cert is server's X.509 certificate
    if cert contains attribute mylar_pubkey then
        pk ← cert.mylar_pubkey
        sig ← response.header["Mylar-Signature"]
        if not VERIFYSIG(pk, response, sig) then
            return ABORT
    if url contains parameter "mylar_hash=h" then
        if hash(response) ≠ h then return ABORT
    return PASS
```

Figure 5: Pseudo-code for Mylar's code verification extension.

Two-origin signing. To address this problem, Mylar uses two origins to host an application. The *primary* origin hosts exactly one file: the application's top-level HTML page. Consequently, this is the only page that can gain access to the application's encryption keys and plaintext data in the browser. All other files, such as images, CSS style sheets, and Javascript code, are loaded from the *secondary* origin. Mylar verifies the authenticity of these files to prevent tampering, but if an adversary tries to load one of these files as a top-level page, it will run with the privileges of the secondary origin, and would not be able to access the application's keys and data.

To verify that the application code has not been tampered with, Mylar requires the site owner to create a public/private key pair, and to sign the application's top-level HTML page (along with the corresponding HTTP headers) with the private key. Any references to other content must refer to the secondary origin, and must be augmented to include a mylar_hash=*h* parameter in the query string, specifying the expected hash of the response. The hash prevents an adversary from tampering with that content or rolling it back to an earlier version. Rollback attacks are possible on the top-level HTML page (because signatures do not guarantee freshness), but in that case, the entire application is rolled back: hashes prevent the adversary from rolling back some but not all of the files, which could confuse the application.

This signing mechanism can verify only the parts of an application that are static and supplied by the web site owner ahead of time. It is up to the application code to safely handle any content dynamically generated by the server at runtime (§3.4). This model is a good fit for AJAX web applications, in which the dynamic content is only data, rather than HTML or code.

Browser extension. Each user of Mylar applications should install the Mylar browser extension in their web browser, which verifies that Mylar applications are properly signed before running them. Figure 5 shows the pseudo-code for the Mylar browser extension. The site owner's public key is embedded in the X.509 certificate

of the web server hosting the web application. Mylar assumes that certificate authorities will sign certificates for the web application's hostname only on behalf of the proper owner of the web application's domain (i.e., the site owner). Thus, as long as the site owner includes the public key in all such certificates, then users visiting the correct web site via `https` will obtain the owner's public key, and will verify that the page was signed by the owner.

7 Implementation

We implemented a prototype of Mylar by building on top of the Meteor web application framework [29]. Meteor allows client-side code to read and update data via MongoDB operations, and also to issue RPCs to the server. Mylar intercepts and encrypts/decrypts data accessed via the MongoDB interface, but requires developers to explicitly handle data passed via RPCs. We have not found this to be necessary in our experience.

We use the SJCL library [38] to perform much of our cryptography in Javascript, and use elliptic curves for most public-key operations, owing to shorter ciphertexts and higher performance. As in previous systems, Mylar uses faster symmetric-key encryption when possible [32]. For bilinear pairings, we use the PBC C++ library to improve performance, which runs either as a Native Client module (for Chrome), as a plugin (for Firefox), or as an NDK-based application (for Android phones). To verify code in the user's browser, we developed a Firefox extension. Mylar comprises ∼9,000 lines of code in total.

When looking up paths in the principal graphs, Mylar performs breadth-first search. We have not found this to be a bottleneck in our experience so far, but more efficient algorithms, such as meet-in-the-middle, are possible.

8 Building a Mylar application

To demonstrate how a developer can build a Mylar application, we show the changes that we made to the kChat application to encrypt messages. In kChat, users can create chat rooms, and existing members of a chat room can invite new users to join. Only invited users have access to the messages from the room. A user can search over data from the rooms he has access to. Figure 6 shows the changes we made to kChat, using Mylar's API (Figure 2).

The call to *Messages*.**encrypted** specifies that data in the "message" field of that collection should be encrypted. This data will be encrypted with the public key of the principal specified in the "roomprinc" field. All future accesses to the *Messages* collection will be transparently encrypted and decrypted by Mylar from this point. The call to *Messages*.**searchable** specifies that clients will need to search over the "message" field; consequently, Mylar will store a searchable encryption of each message in addition to a standard ciphertext.

```
// On both the client and the server:
idp = idp_config(url, pubkey);
Messages.encrypted({"message": "roomprinc"});
Messages.auth_set(["roomprinc", ["id", "message",
                                 "room", "date"]]);
Messages.searchable("message");

// On the client:
function create_user(uname, password):
    create_user(uname, password, idp);
function create_room(roomtitle):
    princ_create(roomtitle, princ_current());
function invite_user(username):
    global room_princ;
    room_princ.add_access(princ_lookup(username, idp));
function join_room(room):
    global cur_room, room_princ;
    cur_room = room;
    room_princ = princ_lookup(room.name,
                              room.creator, idp);
function send_message(msg):
    global cur_room, room_princ;
    Messages.insert({message: msg, room: cur_room.id,
                     date: new Date().toString(),
                     roomprinc: room_princ});
function search(word):
    return Messages.search(word, "message",
                           princ_current(), all, all);
```

Figure 6: Pseudo-code for changes to the kChat application to encrypt messages. Not shown is unchanged code for managing rooms, receiving and displaying messages, and login/logout (Mylar provides wrappers for Meteor's user accounts API).

When a user creates a new room (create_room), the application in turn creates a new principal, named after the room title and signed by the creator's principal. To invite a user to a room, the application needs to give the new user access to the room principal, which it does by invoking **add_access** in invite_user.

When joining a room (join_room), the application must look up the room's public key, so that it can encrypt messages sent to that room. The application specifies both the expected room title as well as the room creator as arguments to **princ_lookup**, to distinguish between rooms with the same title. By displaying both the room title and the creator email address, as in Figure 7, the application helps the user distinguish the correct room from an identically named room that an adversary created.

To send a message to a chat room, kChat needs to specify a principal in the **roomprinc** field of the newly inserted document. In this case, the application keeps the current room's principal in the *room_princ* global variable. Similarly, when searching for messages containing a word, the application supplies the principal whose key should be used to generate the search token. In this case, kChat uses the current user principal, **princ_current**().

Application	LoC before	LoC added for Mylar	Number and types of fields secured	Existed before?	Keyword search on
kChat [23]	793	45	1 field: chat messages	Yes	messages
endometriosis	3659	28	tens of medical fields: mood, pain, surgery, ...	Yes	N/A
submit	8410	40	3 fields: grades, homework, feedback	Yes	homework
photo sharing	610	32	5 fields: photos, thumbnails, captions, ...	Yes	N/A
forum	912	39	9 fields: posts body, title, creator, user info, ...	No	posts
calendar	798	30	8 fields: event body, title, date, user info, ...	No	events
WebAthena [8]	4800	0	N/A: used for code authentication only	Yes	N/A

Figure 8: Applications ported to Mylar. "LoC before" reports the number of lines of code in the unmodified application, not including images or Meteor packages. "Existed before" indicates whether the application was originally built independent of Mylar.

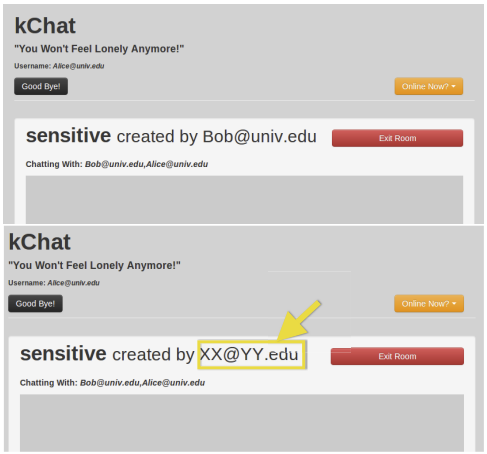


Figure 7: Two screenshots from kChat. On the top, Alice is chatting with Bob as intended. On the bottom, the server provided a fake "sensitive" chat room created by the adversary; Alice can detect this by checking the creator's email address.

9 Evaluation

This section answers two main questions: first, how much developer effort is required to use Mylar, and second, what are the performance overheads of Mylar?

9.1 Developer effort

To measure the amount of developer effort needed to use Mylar, we ported 6 applications to Mylar. Two of these applications plan to start using Mylar in production in the near future: a medical application in which endometriosis patients record their symptoms, and a web site for managing homework and grades for a class at MIT. We also ported an existing chat application called kChat, in which users share chat rooms by invitation and exchange private messages, and a photo sharing application. We also built a Meteor-based forum and calendar, which we then ported to Mylar. Finally, to demonstrate the generality of Mylar's code verification, we used it to verify the code for WebAthena [8], an in-browser Javascript Kerberos client.

Figure 8 summarizes the fields we secured with Mylar in the above applications, along with how much code the developer had to change. In the case of the endometriosis application, fields were stored in the database as field name and field value pairs, so encrypting the generic "value" field secured tens of different kinds of data. In the other apps, a field corresponded to one kind of sensitive data. The results show that Mylar requires little developer effort to protect a wide range of confidential data, averaging 36 lines of code per application.

9.2 Performance

Mylar's performance goal is to avoid significantly affecting the user experience with the web application. To evaluate whether Mylar meets this goal, we answer the following questions:

- How much latency does Mylar add to the web application's overall user interface?
- How much throughput overhead does Mylar impose on a server?
- Is Mylar's multi-key search important to achieve good performance?
- How much storage overhead does Mylar impose?

To answer these questions, we measured the performance of kChat, the homework submission application ("submit"), and the endometriosis application. Although kChat has only one encrypted field, every message sent exercises this field. We used two machines running recent versions of Debian Linux to perform our experiments. The server had an Intel Xeon 2.8 GHz processor and 4 GB of RAM; the client had eight 10-core Intel Xeon E7-8870 2.4 GHz processors with 256 GB of RAM. The client machine is significantly more powerful to allow us to run enough browsers to saturate the server. For browser latency experiments, we simulate a 5 Mbit/s client-server network with 20 msec round-trip latency. All experiments were done over `https`, using nginx as an `https` reverse proxy on the server. We used Selenium to drive a web browser for all experiments. We also evaluated Mylar on Android phones and found that performance remained acceptable, but we omit these results for brevity.

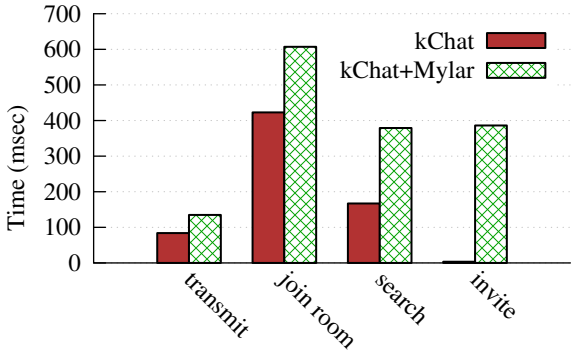


Figure 9: End-to-end latency of four operations in kChat. Transmit includes the time from when one user sends a message to when another user receives it.

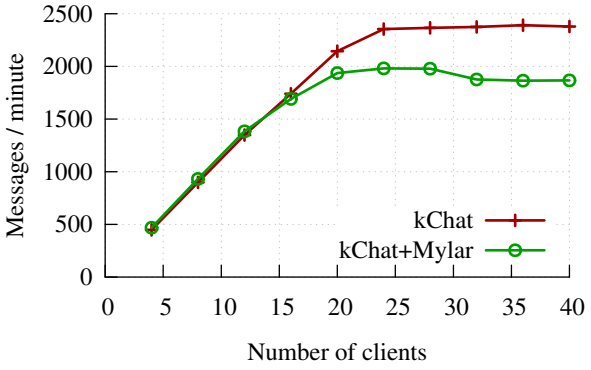


Figure 10: Server throughput for kChat.

End-to-end latency. Figure 9 shows the end-to-end latency Mylar introduces for four main operations in kChat: transmitting a message, joining a room, searching for a word in all rooms, and inviting a user to a room. For message transmission, we measured the time from the sender clicking "send" until the message renders in the recipient's browser. This is the most frequent operation in kChat, and Mylar adds only 50 msec of latency to it. This difference is mostly due to searchable encryption, which takes 43 msec. The highest overhead is for inviting a user, due to principal operations: looking up and verifying a user principal (218 msec) and wrapping the key (167 msec). Overall, we believe the resulting latency is acceptable for many applications, and subjectively the application still feels responsive.

We also measured the latency of initially loading a page. The original kChat application loads in 291 msec. The Mylar version of kChat, without the code verification extension, loads in 356 msec, owing to Mylar's additional code. Enabling the code verification extension increases the load time to 1109 msec, owing to slow signature verification in the Javascript-based extension. Using native code for signature verification, as we did for bilinear pairings, would reduce this overhead. Note that users experience the page load latency only when first navigating to the application; subsequent clicks are handled by the application without reloading the page.

We also measured the end-to-end latency of the most common operations in the endometriosis application (completing a medical survey and reading such a survey), and the submit application (a student uploading an assignment, and a staff member reading such a submission); the results are shown in Figure 11. For the submit application, we used real data from 122 students who used this application during the fall of 2013 in MIT's 6.858 class. Submit's latency is higher than that of other applications because the amount of data (student submissions) is larger, so encryption with search takes longer. For comparison, we also show the latency of submit when search is turned off. The search encryption can happen asynchronously so the user does not have to wait for it.

Throughput. To measure Mylar's impact on server throughput, we used kChat, and we set up many pairs of browsers—a sender and a receiver—where the sender continuously sends new messages. Receivers count the total number of messages received during a fixed interval. Figure 10 shows the results, as a function of the total number of clients (each pair of browsers counts as 2 clients). Mylar decreases the maximum server throughput by 17%. Since the server does not perform any cryptographic operations, Mylar's overhead is due to the increase in message size caused by encryption, and the encrypted search index that is added to every message to make it searchable.

Figure 11 also shows the server throughput of the endometriosis and class submit application when clients perform representative operations.

Search. To evaluate the importance of Mylar's multi-key search, we compare it to two alternative approaches for secure search. The first alternative is single-key server-side search, in which the client generates a token for every key by directly computing the adjusted token from our multi-key search. This alternative is similar to prior work on encrypted keyword search. In this case, the client looks up the principal for every room, computes a token for each, and the server uses one token per room. The second alternative is to perform the search entirely at the client, by downloading all messages. In this case, the client still needs to look up the principal for each room so that it can decrypt the data.

Figure 12 shows the time taken to search for a word in kChat for a fixed number of total messages spread over a varying number of rooms, using multi-key search and the two alternatives described above. We can see that multi-key search is much faster than either of the two alternatives, even with a small number of rooms. The performance of the two alternatives is dominated by

Application	Operation for latency	Latency w/o Mylar	Latency with Mylar	Throughput w/o Mylar	Throughput with Mylar	Throughput units
submit	send and read a submission	65 msec	606 msec	723	394	submissions/min
submit w/o search			70 msec		595	
endometriosis	fill in/read survey	1516 msec	1582 msec	6993	6130	field updates/min

Figure 11: Latency and throughput of different applications with and without Mylar. The latency is the end-to-end time to perform the most common operation in that application. For submit, the latency is the time from one client submitting an assignment until another client obtains that submission. For endometriosis, the latency is the time from one client filling out a survey until another client obtains the survey.

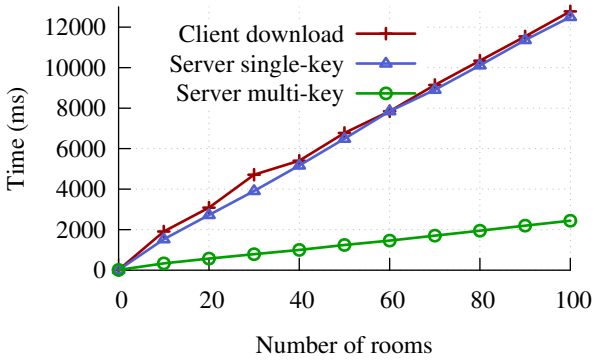


Figure 12: End-to-end latency of keyword search in kChat, searching over 100 6-word messages, spread over a varying number of rooms.

Encrypt	Delta	Token	Adjust	Match
6.5 ms	7.1 ms	0.9 ms	5.6 ms	0.007 ms

Figure 13: Time taken to run each multi-key search operation.

the cost of looking up the principal for each room and obtaining its private key. Multi-key search does not need to do this, because the server directly uses the deltas, and it achieves good performance because both ADJUST and MATCH are fast, as shown in Figure 13.

Storage overhead. For kChat, the server storage overhead after inserting 1,000 messages with Mylar was 4× that of unmodified kChat. This is due to three factors: principal graphs (storing certificates and wrapped keys), symmetric key encryption, and searchable encryption. Our prototype stores ciphertexts in base-64 encoding; using a binary encoding would reduce storage overheads.

10 DISCUSSION

Mylar focuses on protecting confidential data in web applications. However, Mylar's techniques for searching over encrypted data and for verifying keys are equally applicable to desktop and mobile phone applications; the primary difference is that code verification becomes simpler, since applications are explicitly installed by the user, instead of being downloaded at application start time.

Mylar relies on X.509 certificates to supply the web site owner's public key for code verification. Alternative schemes could avoid the need for fully trusted certificate authorities [41, 43], and the Mylar extension could allow users to manually specify site owner public keys for especially sensitive web sites.

Revoking access to shared data is difficult, because Mylar cannot trust the server to forget a wrapped key. Complete revocation requires re-encrypting shared data under a new key, and giving legitimate users access to the new key. In less sensitive situations, it may suffice to try deleting the key from the server, which would work if the server is not compromised at the time of the deletion.

11 CONCLUSION

Mylar is a novel web application framework that enables developers to protect confidential data in the face of arbitrary server compromises. Mylar leverages the recent shift to exchanging data, rather than HTML, between the browser and server, to encrypt all data stored on the server, and decrypt it only in users' browsers. Mylar provides a principal abstraction to securely share data between users, and uses a browser extension to verify code downloaded from the server that runs in the browser. For keyword search, which is not practical to run in the browser, Mylar introduces a cryptographic scheme to perform keyword search at the server over data encrypted with different keys. Experimental results show that using Mylar requires few changes to an application, and that the performance overheads of Mylar are modest.

Mylar and the applications discussed in this paper are available at `http://css.csail.mit.edu/mylar/`.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Mike Freedman, for their feedback. We also thank Linda Griffith, John Guttag, Nicolaas Kaashoek, and Michelle Park for providing a real medical use case for Mylar with their endometriosis application. This research was supported by NSF award IIS-1065219, by DARPA CRASH under contracts #N66001-10-2-4088 and #N66001-10-2-4089, by Quanta, and by Google.

REFERENCES

[1] D. Akhawe, P. Saxena, and D. Song. Privilege separation in HTML5 applications. In *Proceedings of the 21st Usenix Security Symposium*, Bellevue, WA, Aug. 2012.

[2] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, Jan. 2013.

[3] G. Ateniese, K. Fu, M. Green, and S. Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2006.

[4] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 205–216, Athens, Greece, June 2011.

[5] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Proceedings of the 17th Usenix Security Symposium*, San Jose, CA, July–Aug. 2008.

[6] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[7] F. Beato, M. Kohlweiss, and K. Wouters. Scramble! your social network data. In *Proceedings of the 11th Privacy Enhancing Technologies Symposium*, Waterloo, Canada, July 2011.

[8] D. Benjamin. Adapting Kerberos for a browser-based environment. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Sept. 2013.

[9] D. Borelli. The name Edward Snowden should be sending shivers up CEO spines. *Forbes*, Sept. 2013. `http://www.forbes.com/sites/realspin/2013/09/03/the-name-edward-snowden-should-be-sending-shivers-up-ceo-spines/`.

[10] A. Chen. GCreep: Google engineer stalked teens, spied on chats. *Gawker*, Sept. 2010. `http://gawker.com/5637234/`.

[11] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[12] M. Christodorescu. Private use of untrusted web servers via opportunistic encryption. In *Proceedings of the Web 2.0 Security and Privacy Workshop*, Oakland, CA, May 2008.

[13] CipherCloud. Cloud data protection solution. `http://www.ciphercloud.com`.

[14] Defuse Security. Encrypted pastebin. `https://defuse.ca/pastebin.htm`, Sept. 2013.

[15] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[16] R. Fischer, M. Seltzer, and M. Fischer. Privacy from untrusted web servers. Technical Report YALEU/DCS/TR-1290, Yale University, Department of Computer Science, May 2004.

[17] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 169–178, Bethesda, MD, May–June 2009.

[18] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[19] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 555–564, Palo Alto, CA, June 2013.

[20] Google, Inc. User data requests – Google transparency report, Sept. 2013. `http://www.google.com/transparencyreport/userdatarequests/`.

[21] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, Raleigh, NC, Oct. 2012.

[22] J. Kannan, P. Maniatis, and B.-G. Chun. Secure data preservers for web services. In *Proceedings of the 2nd USENIX Conference on Web Application*

Development, Portland, OR, June 2011.

[23] KiqueDev. kChat. https://github.com/KiqueDev/kChat/.

[24] M. Krohn. Building secure high-performance web services with OKWS. In *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, June–July 2004.

[25] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–106, San Francisco, CA, Dec. 2004.

[26] A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, New York, NY, May 2012.

[27] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[28] Mega. The privacy company. https://mega.co.nz/#privacycompany, Sept. 2013.

[29] Meteor, Inc. Meteor: A better way to build apps. http://www.meteor.com, Sept. 2013.

[30] OpenID Foundation. OpenID. http://openid.net, Sept. 2013.

[31] R. A. Popa and N. Zeldovich. Multi-key searchable encryption. Cryptology ePrint Archive, Report 2013/508, Aug. 2013. http://eprint.iacr.org/.

[32] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 85–100, Cascais, Portugal, Oct. 2011.

[33] K. Puttaswamy, C. Kruegel, and B. Zhao. Silverline: Toward data confidentiality in storage-intensive cloud applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, Cascais, Portugal, Oct. 2011.

[34] F. Y. Rashid. Salesforce.com acquires SaaS encryption provider Navajo Systems. *eWeek.com*, August 2011.

[35] S. Sauvage. ZeroBin - because ignorance is bliss. http://sebsauvage.net/wiki/doku.php?id=php:zerobin, Feb. 2013.

[36] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, pages 44–55, Oakland, CA, May 2000.

[37] E. Stark. From client-side encryption to secure web applications. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 2013.

[38] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in Javascript. In *Proceedings of the Annual Computer Security Applications Conference*, Honolulu, HI, Dec. 2009.

[39] The Cryptocat Project. Moving to a browser app model. https://blog.crypto.cat/2012/08/moving-to-a-browser-app-model/, Aug. 2012.

[40] The Cryptocat Project. Cryptocat. http://www.cryptocat.com, Sept. 2013.

[41] Thoughtcrime Labs. Convergence. http://convergence.io/, 2011.

[42] J. Tudor. Web application vulnerability statistics, June 2013. http://www.contextis.com/files/Web_Application_Vulnerability_Statistics_-_June_2013.pdf.

[43] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, MA, June 2008.

[44] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Usenix Security Symposium*, Vancouver, Canada, July 2006.

[45] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, Mar. 2009.

[46] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 291–304, Big Sky, MT, Oct. 2009.

[47] M. Zalewski. *The Tangled Web*. No Starch Press, 2012.

PHY Covert Channels: Can you see the Idles?

Ki Suh Lee, Han Wang, Hakim Weatherspoon
Computer Science Department, Cornell University
kslee,hwang,hweather@cs.cornell.edu

Abstract

Network covert timing channels embed secret messages in legitimate packets by modulating interpacket delays. Unfortunately, such channels are normally implemented in higher network layers (layer 3 or above) and easily detected or prevented. However, access to the *physical layer* of a network stack allows for timing channels that are virtually invisible: Sub-microsecond modulations that are undetectable by software endhosts. Therefore, covert timing channels implemented in the physical layer can be a serious threat to the security of a system or a network. In fact, we empirically demonstrate an effective covert timing channel over nine routing hops and thousands of miles over the Internet (the National Lambda Rail). Our covert timing channel works with cross traffic, less than 10% bit error rate, which can be masked by forward error correction, and a covert rate of 81 kilobits per second. Key to our approach is access and control over every bit in the physical layer of a 10 Gigabit network stack (a bit is 100 picoseconds wide at 10 gigabit per seconds), which allows us to modulate and interpret interpacket spacings at sub-microsecond scale. We discuss when and how a timing channel in the physical layer works, how hard it is to detect such a channel, and what is required to do so.

1 Introduction

Covert channels are defined as channels that are not intended for information transfer, but can leak sensitive information [21]. In essence, covert channels provide the ability to hide the transmission of data within established network protocols [37], thus hiding their existence. Covert channels are typically classified into two categories: Storage and timing channels. In *storage channels*, a sender modulates the value of a storage location to send a message. In *timing channels*, on the other hand, a sender modulates system resources over time to send a message [10].

Network covert channels send hidden messages over *legitimate* packets by modifying packet headers (storage channels) or by modulating interpacket delays (timing channels). Because network covert channels can deliver sensitive messages across a network to a receiver multiple-hops away, they impose serious threats to the security of systems. Network *storage* channels normally exploit unused fields of protocol headers [20, 28, 34, 35], and, thus, are relatively easy to detect and prevent [14, 19, 26]. Network *timing* channels deliver messages by modulating interpacket delays (or arrival time of packets). As a result, arrivals of packets in network timing channels normally create patterns, which can be analyzed with statistical tests to detect timing channels [11, 12, 16, 32], or eliminated by network jammers [17]. To make timing channels robust against such detection and prevention, more sophisticated timing channels mimic legitimate traffic with spreading codes and a shared key [24], or use independent and identically distributed (i.i.d) random interpacket delays [25].

In this paper, we present a new method of creating a covert timing channel that is *high-bandwidth*, *robust* against cross traffic, and *undetectable* by software endhosts. The channel can effectively deliver 81 kilobits per second with less than 10% errors over nine routing hops, and thousands of miles over the National Lambda Rail (NLR). We empirically demonstrate that we can create such a timing channel by modulating interpacket gaps at sub-microsecond scale: A scale at which sent information is preserved through multiple routing hops, but statistical tests cannot differentiate the channel from legitimate traffic. Unlike approaches mentioned above, our covert timing channel, *Chupja*[1], is implemented in the physical layer of a network protocol stack. In order to hide the existence of the channel, we mainly exploit the fact that statistical tests for covert channel detection rely on collected interpacket delays, which can be highly inaccurate in a 10 Gigabit Ethernet (GbE) network, whereas access to the physical layer provides fine-grained control over interpacket delays at nanosecond scale [15, 22]. As a result, a network monitoring application needs to have the capability of fine-grained timestamping to detect our covert channel. We argue that nanosecond level of resolution is key to do so.

The contributions of this paper are as follows:

- We discuss how to design and implement a covert timing channel via access to the physical layer.
- We demonstrate that a covert timing channel implemented in the physical layer can effectively deliver secret messages over the Internet.
- We empirically illustrate that we can quantify perturbations added by a network, and the quantified

[1] *Chupja* is equivalent to *spy* in Korean

perturbation is related to bit error rate of the covert timing channel.
- We show that in order to detect *Chupja*, fine-grained timestamping at nanosecond scale is required.

2 Network Covert Channels

Network covert channels are not new. However, implementing such a channel in the physical layer has never been tried before. In this section, we briefly discuss previous approaches to create and detect network covert channels, and why access to the physical layer can create a covert channel that is hard to detect. Although our focus of this paper is covert timing channels, we discuss both covert storage channels and covert timing channels in this section.

In a network covert channel, the *sender* has secret information that she tries to send to a *receiver* over the Internet. The sender has control of some part of a network stack including a network interface (L1∼2), kernel network stack (L3∼4) and/or user application (L5 and above). Thus, the sender can modify protocol headers, checksum values, or control the timing of transmission of packets. The sender can either use packets from other applications of the system or generate its own packets. Although it is also possible that the sender can use packet payloads to directly embed or encrypt messages, we do not consider this case because it is against the purpose of a covert channel: *hiding the existence of the channel*. The *adversary* (or warden), on the other hand, wants to detect and prevent covert channels. A *passive* adversary monitors packet information to detect covert channels while an *active* adversary employs network appliances such as network jammers to reduce the possibility of covert channels.

In network *storage* channels, the sender changes the values of packets to secretly encode messages, which is examined by the receiver to decode the message. This can be easily achieved by using unused bits or fields of protocol headers. The IP Identification field, the IP Fragment Offset, the TCP Sequence Number field, and TCP timestamps are good places to embed messages [20, 28, 34, 35]. As with the easiness of embedding messages in packet headers, it is just as easy to detect and prevent such storage channels. The adversary can easily monitor specific fields of packet headers for detection, or *sanitize* those fields for prevention [14, 19, 26].

In network *timing* channels, the sender controls the timing of transmission of packets to deliver hidden messages. The simplest form of this channel is to send or not send packets in a pre-arranged interval [12, 31]. Because interpacket delays are perturbed with noise from a network, synchronizing the sender and receiver is a major challenge in these on/off timing channels. However, synchronization can be avoided when each interpacket delay conveys information, i.e. a long delay is zero, and a short delay is one [11]. JitterBugs encodes bits in a similar fashion, and uses the remainder of modulo operation of interpacket delays for encoding and decoding [36]. These timing channels naturally create patterns of interpacket delays which can be analyzed with statistical tests for detection. For example, *regularity* tests [11, 12], *shape* tests [32], or *entropy* tests [16] are widely used for covert timing channel detection. On the other hand, to avoid detection from such statistical tests, timing channels can mimic patterns of legitimate traffic, or use random interpacket delays. Liu et al., demonstrated that with spreading codes and a shared key, a timing channel can be robust against known statistical tests [24]. They further developed a method to use independent and identically distributed (i.i.d) random interpacket delays to make the channel less detectable [25].

Access to the physical layer (PHY) allows the sender to create new types of both storage and timing channels. The sender of a covert storage channel can embed secret messages into special characters that only reside in the physical layer, which are discarded before the delivery of packets to higher layers of a network stack. As a result, by embedding messages into those special characters, higher layers of a network stack will have no way to detect the existence of the storage channel. In fact, idle characters (`/I/`s), which are used to fill gaps between any two packets in the physical layer, would make for great covert channels if they could be manipulated. The IEEE 802.3 standard requires that at least twelve `/I/` characters must be inserted after every packet [3]. Therefore, it is possible to create a high-bandwidth storage channel that cannot be detected without access to the PHY. Unfortunately, this covert storage channel can only work for one hop, i.e. between two directly connected devices, because network devices discard the contents of idle characters when processing packets. However, if a *supply chain attack* is taken into account where switches and routers between the sender and the receiver are compromised and capable of forwarding hidden messages, the PHY storage channel can be very effective. We have implemented a PHY covert storage channel and verified that it is effective, but only for one hop. To prevent the PHY storage channel, all special characters must be sanitized (or zeroed) at every hop.

Our focus, however, is not a PHY covert storage channel. Instead, we demonstrate that sophisticated covert timing channels can be created via access to the PHY. The idea is to control (count) the number of `/I/`s to encode (decode) messages. i.e. to modulate interpacket gaps in nanosecond resolution.

Any network component that has access to the PHY, and thus `/I/`s, can potentially detect PHY covert channels. Indeed, routers and switches have the capability to

access the physical layer (i.e. /I/s). Unfortunately, they are not normally programmable and do not provide an interface for access to /I/s. Instead, anyone that wants to detect PHY covert timing channels would need to apply statistical tests on interpacket delays (discussed earlier in this section). Of course, interpacket delays needs to be captured precisely before doing so.

In other words, PHY covert timing channels could be potentially detected if the time of packet reception could be accurately timestamped with fine-grained resolution (i.e. nanosecond precision; enough precision to measure interpacket gaps of 10 GbE networks). However, commodity network devices often lack precise timestamping capabilities. Further, although high-end network monitoring appliances [2, 9] or network monitoring interface cards [1, 6] are available with precise timestamping capabilities, deploying such high-end appliances in a large network is not common due to the volume of traffic they would need to process (they have limited memory/storage) and cost.

Given that programmatic access to the PHY and accurate timestamping capabilities in high-speed networks are not readily available, we assume a passive adversary who uses commodity servers with commodity network interface cards (NIC) for network monitoring. An example adversary is a network administrator monitoring a network using `pcap` applications. This implies that the adversary does not have access to the PHY of a network stack.

Can a passive adversary built from a commodity server and NIC detect a PHY timing channel? We will demonstrate that it cannot (Section 4.4). In particular, we will show how to exploit inaccurate timestamping of network monitoring applications in order to hide the existence of such a channel. It has been shown that access to the PHY allows very precise timestamping at sub-nanosecond resolution, whereas endhost timestamping is too inaccurate to capture the nature of 10 GbE [15, 22]. In particular, an endhost relies on its own system clock to timestamp packets which normally provides microsecond resolution, or hardware timestamping from network interface cards which provides sub-microsecond resolution. Unfortunately, packets can arrive much faster than the endhost can timestamp them. Therefore, inaccurate timestamping at an endhost can lead to an opportunity to create a timing channel. In this paper, we will discuss how to create a timing channel by modulating interpacket gaps precisely in a way that network monitoring applications cannot detect any regularities from them.

3 *Chupja*: PHY timing channel

In this section, we discuss the design and implementation of our physical layer (PHY) covert timing channel, *Chupja*. Since *Chupja* is implemented via access to the physical layer, we briefly discuss the IEEE 802.3 10 Gigabit Ethernet standard first, and present the design goal of *Chupja*, how to encode and decode secret messages, how *Chupja* is implemented, and other considerations.

3.1 10 GbE Physical Layer

According to the IEEE 802.3 standard [3], when Ethernet frames are passed to the PHY, they are reformatted before being sent across the physical medium. On the transmit path, the PHY encodes every 64 bits of an Ethernet frame into a 66-bit *block*, which consists of a two bit *synchronization header* (syncheader) and a 64-bit *payload*. As a result, a 10 GbE link actually operates at 10.3125 Gbaud ($10G \times \frac{66}{64}$). The PHY also scrambles each block before passing it down the network stack to be transmitted. The entire 66-bit block is transmitted as a continuous stream of *symbols* which a 10 GbE network transmits over a physical medium. As 10 GbE always sends 10.3125 gigabits per second (Gbps), each bit in the PHY is about 97 picoseconds wide. On the receive path, the PHY descrambles each 66-bit block before decoding it.

Idle characters (/I/) are special characters that fill any gaps between any two packets in the PHY. When there is no Ethernet frame to transmit, the PHY continuously inserts /I/ characters until the next frame is available. The standard requires at least twelve /I/s after every packet. An /I/ character consists of seven or eight bits, and thus it takes about 700∼800 picoseconds to transmit one /I/ character. /I/s are typically inaccessible from higher layers (L2 or above), because they are discarded by hardware.

3.2 Design Goal

The design goal of our timing channel, *Chupja*, is to achieve *high-bandwidth*, *robustness* and *undetectability*. By high-bandwidth, we mean a covert rate of many tens or hundreds of thousands of bits per second. Robustness is how to deliver messages with minimum errors, and undetectability is how to hide the existence of it. In particular, we set as our goal for robustness to a bit error rate (BER) of less than 10%, an error rate that is small enough to be compensated with forward error correction such as Hamming code, or spreading code [24]. We define BER as the ratio of the number of bits incorrectly delivered from the number of bits transmitted.

In order to achieve these goals, we precisely modulate the number of /I/s between packets in the physical layer. If the modulation of /I/s is large, the channel can effectively send messages in spite of noise or perturbations from a network (robustness). At the same time, if the modulation of /I/s is small, an adversary will not be able to detect regularities (undetectability). Further, *Chupja* embeds one timing channel bit per interpacket gap to achieve high-bandwidth. Thus, higher

overt packet rates will achieve higher covert timing channel rates. We focus on finding an optimal modulation of interpacket gaps to achieve high-bandwidth, robustness, and undetectability (Section 4).

3.3 Model

In our model, the sender of *Chupja* has control over a network interface card or a compromised switch[2] with access to and control of the physical layer. In other words, the sender can easily control the number of `/I/` characters of outgoing packets. The receiver is in a network multiple hops away, and taps/sniffs on its network with access to the physical layer. Then, the sender modulates the number of `/I/`s between packets destined to the receiver's network to embed secret messages.

Our model includes an adversary who runs a network monitoring application and is in the same network with the sender and receiver. As discussed in Section 2, we assume that the adversary is built from a commodity server and commodity NIC. As a result, the adversary does not have direct access to the physical layer. Since a commodity NIC discards `/I/`s before delivering packets to the host, the adversary cannot monitor the number of `/I/`s to detect the possibility of covert timing channels. Instead, it runs statistical tests with captured interpacket delays.

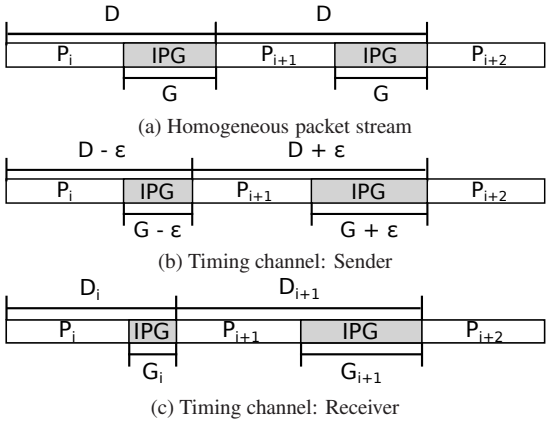
3.4 Terminology

We define *interpacket delay* (IPD) as the time difference between the first bits of two successive packets, and *interpacket gap* (IPG) as the time difference between the last bit of the first packet and the first bit of the next packet. Thus, an interpacket delay between two packets is equal to the sum of transmission time of the first packet and the interpacket gap between the two (i.e. IPD = IPG + packet size). A *homogeneous* packet stream consists of packets that have the same destination, the same size and the same IPGs (IPDs) between them. Furthermore, the variance of IPGs and IPDs of a homogeneous packet stream is always zero.

3.5 Encoding and Decoding

Chupja embeds covert bits into interpacket gaps of a homogeneous packet stream of an *overt* channel. In order to create *Chupja*, the sender and receiver must share two parameters: G and W. G is the number of `/I/`s in the IPG that is used to encode and decode hidden messages, and W (*Wait time*) helps the sender and receiver synchronize (Note that interpacket delay $D = G$ + packet size). Figure 1 illustrates our design. Recall that IPGs of a homogeneous packet stream are all the same (=G, Figure 1a). For example, the IPG of a homogeneous stream with 1518 byte packets at 1 Gbps is always 13738 `/I/`s; the variance is zero. To embed a secret bit sequence

[2]We use the term *switch* to denote both bridge and router.



(a) Homogeneous packet stream

(b) Timing channel: Sender

(c) Timing channel: Receiver

Figure 1: *Chupja* encoding and decoding.

$\{b_i,\ b_{i+1},\ \cdots\}$, the sender encodes 'one' ('zero') by increasing (decreasing) the IPG (G) by ε `/I/`s (Figure 1b):

$$G_i = G - \varepsilon \text{ if } b_i = 0$$
$$G_i = G + \varepsilon \text{ if } b_i = 1$$

where G_i is the ith interpacket gap between packet i and $i+1$. When G_i is less than the minimum interpacket gap (or 12 `/I/` characters), it is set to twelve to meet the standard requirement.

Interpacket gaps (and delays) will be perturbed as packets go through a number of switches. However, as we will see in Section 4.3, many switches do not significantly change interpacket gaps. Thus, we can expect that if ε is large enough, encoded messages will be preserved along the path. At the same time, ε must be small enough to avoid detection by an adversary. We will evaluate how big ε must be with and without cross traffic and over multiple hops of switches over thousands of miles in a network path (Section 4).

Upon receiving packet pairs, the receiver decodes bit information as follows:

$$b'_i = 1 \text{ if } G_i \geq G$$
$$b'_i = 0 \text{ if } G_i < G$$

b'_i might not be equal to b_i because of network noise. We use BER to evaluate the performance of *Chupja* (Section 4.2).

Because each IPG corresponds to a signal, there is no need for synchronization between the sender and the receiver [11]. However, the sender occasionally needs to pause until the next covert packet is available. W is used when there is a pause between signals. The receiver considers an IPG that is larger than W as a pause, and uses the next IPG to decode the next signal.

3.6 Implementation

We used SoNIC to implement and evaluate *Chupja*. SoNIC [22] allows users to access and control every bit

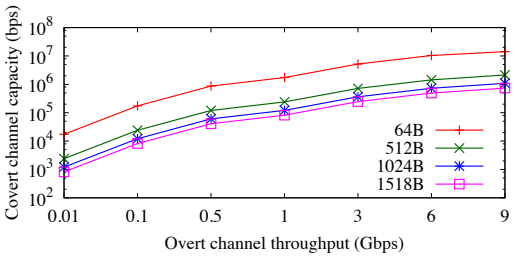


Figure 2: Maximum capacity of PHY timing channel

of 10 GbE physical layer, and thus we can easily control (count) the number of /I/s between packets. We extended SoNIC's packet generation capability to create a timing channel. Given the number of /I/ characters in the original IPG (G), the number of /I/s to modulate (ε) and a secret message, IPGs are changed accordingly to embed the message. On the receiver side, it decodes the message by counting the number of /I/s between packets in realtime. The total number of lines added to the SoNIC implementation was less than 50 lines of code.

The capacity of this PHY timing channel is equal to the number of packets being transmitted from the sender when there is no pause. Given a packet size, the maximum capacity of the channel is illustrated in Figure 2. For example, if an overt channel sends at 1 Gbps with 1518 byte packets, the maximum capacity of the covert channel is 81,913 bits per second (bps). We will demonstrate in Section 4.2 that *Chupja* can deliver 81 kilobits per second (kbps) with less than 10% BER over nine routing hops and thousands of miles over National Lambda Rail (NLR).

3.7 Discussion

Chupja uses homogeneous packet streams to encode messages, which creates a regular pattern of IPGs. Fortunately, as we will discuss in the following section, the adversary will be unable to accurately timestamp incoming packets when the data rate is high (Section 4.4). This means that it does not matter what patterns of IPGs are used for encoding at above a certain data rate. Therefore, we chose the simplest form of encoding for *Chupja*. The fact that the PHY timing channel works over multiple hops means that a non-homogeneous timing channel will work as well. For instance, consider the output after one routing hop as the sender, then the PHY timing channel works with a non-homogeneous packet stream. If, on the other hand, the sender wants to use other patterns for encoding and decoding, other approaches can easily be applied [12, 24, 25, 36]. For example, if the sender wants to create a pattern that looks more random, we can also use a shared secret key and generate random IPGs for encoding and decoding [25]. However, the focus of this paper is to demonstrate that even this simplest form of timing channel can be a serious threat to a system and not easily be detected.

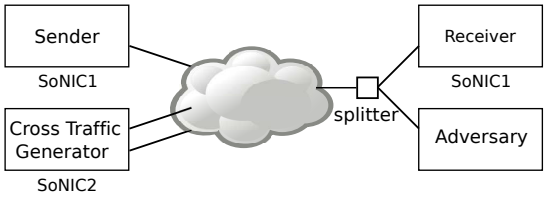


Figure 3: Network topology for evaluation. All lines are 10 gigabit fiber optic cables

Finally, note that a *Chupja* sender and receiver do not need to be endpoints of a network path, but could actually be within the network as middleboxes. Such a covert timing channel middlebox would require constant overt traffic in order to manipulate interpacket gaps.

4 Evaluation

In this section, we evaluate *Chupja* over real networks. We attempt to answer following questions.

- How *robust* is *Chupja* (Section 4.2)? How effectively can it send secret messages over the Internet?
- Why is *Chupja robust* (Section 4.3)? What properties of a network does it exploit?
- How *undetectable* is *Chupja* (Section 4.4)? Why is it hard to detect it and what is required to do so?

In Section 4.2, we first demonstrate that *Chupja* works effectively over the Internet, and achieves a Bit Error Rate (BER) less than 10% which is the design goal of *Chupja* (Section 3). In particular, we evaluated *Chupja* over two networks: A small network that consists of multiple commercial switches, and the National Lambda Rail (NLR). We discuss what is the optimal interpacket gap (IPG) modulation, ε, that makes *Chupja* work. Then, in order to understand why *Chupja* works, we provide a microscopic view of how network devices behave in Section 4.3. We conducted a sensitivity analysis over commercial switches. We mainly show how network devices preserve small interpacket delays along the path even with and without the existence of cross traffic. Lastly, we discuss how to detect a sophisticated timing channel such as *Chupja* in Section 4.4.

4.1 Evaluation Setup

For experiments in this section, we deployed two SoNIC servers [22] each equipped with two 10 GbE ports to connect fiber optic cables. We used one SoNIC server (SoNIC1) to generate packets of the sender destined to a server (the adversary) via a network. We placed a fiber optic splitter at the adversary which mirrored packets to SoNIC1 for capture (i.e. SoNIC1 was both the timing channel sender and receiver). SoNIC2 was used to generate cross traffic flows when necessary (Figure 3). Throughout this section, we placed none or multiple commercial switches between the sender and the adversary (the cloud within Figure 3).

Table 1 summarizes the commercial switches that we

	Type	40G	10G	1G	Full bandwidth	Forwarding
SW1	Core	0	8	0	160 Gbps	SF
SW2	ToR	4	48	0	1280 Gbps	CT
SW3	ToR	0	2	48	136 Gbps	SF
SW4	ToR	0	2	24	105.6 Gbps	SF

Table 1: Summary of evaluated network switches. "SF" is store-and-forward and "CT" is cut-through.

used. SW1 is a core / aggregate router with multiple 10 GbE ports, and we installed two modules with four 10 GbE ports. SW2 is a high-bandwidth 10 GbE top-of-rack (ToR) switch which is able to support forty eight 10 GbE ports at line speed. Moreover, it is a cut-through switch whose latency of forwarding a packet is only a few microseconds. SW3 and SW4 are 1 GbE ToR switches with two 10 GbE uplinks. Other than SW2, all switches are store-and-forward switches.

We used a Dell 710 server for the adversary. The server has two X5670 2.93GHz processors each with six CPU cores, and 12 GB RAM. The architecture of the processor is Westmere [4] that is well-known for its capability of processing packets in a multi-threading environment [13, 18, 27]. We used one 10 GbE Dual-port NICs for receiving packets. No optimization or performance tuning such as `irq` balancing, or interrupt coalescing, was performed except New API (NAPI) [7] which is enabled by default.

Packet size [Bytes]	Data Rate [Gbps]	Packet Rate [pps]	IPD [ns]	IPG [/I/]
1518	9	737028	1356.8	170
1518	6	491352	2035.2	1018
1518	3	245676	4070.4	3562
1518	1	81913	12211.2	13738
64	6	10416666	96.0	48
64	3	5208333	192.0	168
64	1	1736111	576.0	648

Table 2: IPD and IPG of homogeneous packet streams.

ε (/I/s)	16	32	64	128	256	512	1024	2048	4096
ns	12.8	25.6	51.2	102.4	204.8	409.6	819.2	1638.4	3276.8

Table 3: Evaluated ε values in the number of `/I/`s and their corresponding time values in nanosecond.

For most of the evaluation, we used 1518 byte and 64 byte packets for simplicity. We define packet size as the number of bytes from the first byte of the Ethernet header to the last byte of the Ethernet frame check sequence (FCS) field (i.e. we exclude seven preamble bytes and start frame delimiter byte from packet size). Then, the largest packet allowed by Ethernet is 1518 bytes (14 byte header, 1500 payload, and 4 byte FCS), and the smallest is 64 bytes. In this section, the data rate refers to the data rate of the overt channel that *Chupja* is embedded. Interpacket delays (IPDs) and interpacket gaps (IPGs) of homogeneous packet streams at different data rates and with different packet sizes are summarized

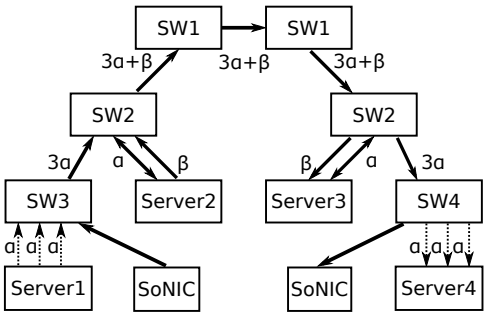


Figure 4: A small network. Thick solid lines are 10G connections while dotted lines are 1G connections.

in Table 2. Table 3 shows the number of `/I/`s (ε) we modulate to create *Chupja* and their corresponding time values in nanosecond. We set ε starting from 16 `/I/`s (= 12.8 ns), doubling the number of `/I/`s up to 4096 `/I/`s (= 3276.8 ns). We use a tuple (s, r) to denote a packet stream with s byte packets running at r Gbps. For example, a homogeneous stream with (1518B, 1Gbps) is a packet stream with 1518 byte packets at 1 Gbps.

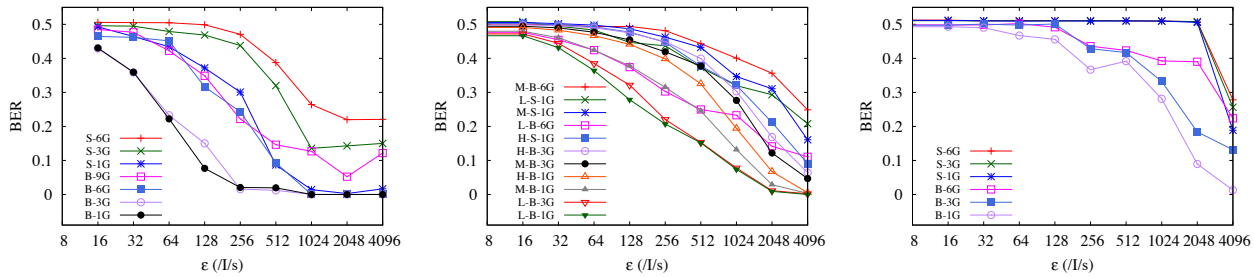
4.2 Efficiency of *Chupja*

The goal of a covert timing channel is to send secret messages to the receiver with minimum errors (robustness). As a result, Bit Error Rate (BER) and the achieved covert bandwidth are the most important metrics to evaluate a timing channel. Our goal is to achieve BER less than 10% over a network with high bandwidth. In this section, we evaluate *Chupja* over two networks, a small network and the Internet (NLR), focusing on the relation between BER and the number of `/I/`s being modulated (ε).

4.2.1 A small network

We created our own network by connecting six switches, and four servers (See Figure 4). The topology resembles a typical network where core routers (SW1) are in the middle and 1 GbE ToR switches (SW3 and SW4) are leaf nodes. Then, SoNIC1 (the sender) generates packets to SW3 via one 10 GbE uplink, which will forward packets to the receiver which is connected to SW4 via one 10 GbE uplink. Therefore, it is a seven-hop timing channel with 0.154 ms round trip time delay on average, and we measured BER at the receiver.

Before considering cross traffic, we first measure BER with no cross traffic. Figure 5a illustrates the result. The x-axis is ε modulated in the number of idle (`/I/`) characters (see Table 3 to relate `/I/`s to time), and the y-axis is BER. Figure 5a clearly illustrates that the larger ε, the smaller BER. In particular, modulating 128 `/I/`s (=102.4 ns) is enough to achieve BER=7.7% with (1518B, 1Gbps) (filled in round dots). All the other cases also achieve the goal BER except (64B, 6Gbps) and (64B, 3Gbps). Recall that Table 2 gives the capacity of the covert channel. The takeaway is that when there is



(a) Without cross traffic over a small network (b) With cross traffic over a small network (c) Over National Lambda Rail

Figure 5: BER of *Chupja* over a small network and NLR. X-Y-Z means that workload of cross traffic is X (H-heavy, M-medium, or L-light), and the size of packet and data rate of overt channel is Y (B-big=1518B or S-small=64B) and Z (1, 3, or 6G).

no cross traffic, modulating small number of `/I/s` (128 `/I/s`, 102.4 ns) is sufficient to create a timing channel. In addition, it is more efficient with large packets.

Now, we evaluate *Chupja* with cross traffic. In order to generate cross traffic, we used four servers (Server1 to 4). Each server has four 1 GbE and two 10 GbE ports. Server1 (Server4) is connected to SW3 (SW4) via three 1 GbE links, and Server2 (Server3) is connected to SW3 via two 10 GbE links. These servers generate traffic across the network with Linux `pktgen` [30]. The bandwidth of cross traffic over each link between switches is illustrated in Figure 4: 1 GbE links were utilized with flows at α Gbps and 10 GbE links at β Gbps. We created three workloads where $(\alpha,\beta) = (0.333,0.333)$, $(0.9,0.9)$, and $(0.9,3.7)$, and we call them Light, Medium and Heavy workloads. Packets of cross traffic were always maximum transmission unit (MTU) sized. Then SoNIC1 generated timing channel packets at 1, 3, and 6 Gbps with 1518 and 64 byte packets. Figure 5b illustrates the result. At a glance, because of the existence of cross traffic, ε must be larger to transmit bits correctly compared to the case without cross traffic. There are a few takeaways. First, regardless of the size of workloads, timing channels with (1518B, 1Gbps) and (1518B, 3Gbps) work quite well, achieving the goal BER of less than 10% with $\varepsilon \geq 1024$. On the other hand, channels at a data rate higher than 6 Gbps are not efficient. In particular, $\varepsilon = 4096$ is not sufficient to achieve the goal BER with (1518B, 6Gbps). Second, creating timing channels with small packets is more difficult. Generally, BER is quite high even with $\varepsilon = 4096$ except H-S-1G case (BER=9%).

4.2.2 National Lambda Rail

In this section, we evaluate *Chupja* in the wild over a real network, National Lambda Rail (NLR). NLR is a wide-area network designed for research and has significant cross traffic [29]. We set up a path from Cornell university to NLR over nine routing hops and 2500 miles

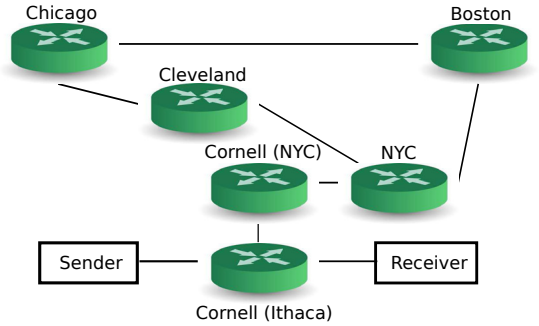


Figure 6: Our path on the National Lambda Rail

one-way (Figure 6). All the routers in NLR are Cisco 6500 routers. We used a SoNIC server to generate and capture *Chupja* packets at each end. The average round trip time of the path was 67.6 ms, and there was always cross traffic. In particular, many links on our path were utilized with 1∼2 Gbps cross traffic during the experiment. Cross traffic was not under our control, however we received regular measurements of traffic on external interfaces of all routers.

Figure 5c illustrates the results. Again, we changed the size and the data rate of overt packets. In NLR, it becomes more difficult to create a timing channel. In particular, only (1518B, 1Gbps) achieved BER less than 10% when ε is larger than 2048 (8.9%). All the other cases have higher BERs than our desired goal, although BERs are less than 30% when ε is 4096. Creating a channel with 64 byte packet is no longer possible in NLR. This was because more than 98% of IPGs were minimum interpacket gaps, i.e. most of bit information was discarded because of packet train effects [15].

We demonstrated in this section (Figure 5c) that we can create an effective covert timing channel with (1518B, 1Gbps), and with large enough ε over the NLR. The capacity of this channel can be as high as 81 kbps (Table 2). In general, large packets at slower data rate is desirable to create a timing channel. In the following sec-

tion, we will investigate why this is possible by closely analyzing the behavior of network devices with respect to IPG modulations, ε.

4.3 Sensitivity Analysis

Network devices change interpacket gaps while forwarding packets; switches add randomness to interpacket gaps. In this section, we discuss how *Chupja* can deliver secret messages via a PHY timing channel in spite of the randomness added from a network. In particular, we discuss the following observations.

- A single switch does not add significant perturbations to IPDs when there is no cross traffic.
- A single switch treats IPDs of a timing channel's encoded 'zero' bit and those of an encoded 'one' bit as uncorrelated distributions; ultimately, allowing a PHY timing channel receiver to distinguish an encoded 'zero' from an encoded 'one'.
- The first and second observations above hold for multiple switches and cross traffic.

In other words, we demonstrate that timing channels can encode bits by modulating IPGs by a small number of `/I/` characters (hundreds of nanoseconds) and these small modulations can be effectively delivered to a receiver over multiple routing hops with cross traffic.

In order to understand and appreciate these observations, we must first define a few terms. We denote the interpacket delay between packet i and $i+1$ with the random variable D_i. We use superscript on the variables to denote the number of routers. For example, D_i^1 is the interpacket delay between packet i and $i+1$ after processed by one router, and D_i^0 is the interpacket delay before packet i and $i+1$ are processed by any routers. Given a distribution of D, and the average interpacket delay μ, we define I_{90} as the smallest ε that satisfies $P(\mu-\varepsilon \le D \le \mu+\varepsilon) \ge 0.90$. In other words, I_{90} is the interval from the average interpacket delay, μ, which contains 90% of D (i.e. at least 90% of distribution D is within $\mu \pm I_{90}$). For example, I_{90} of a homogeneous stream (a delta function, which has no variance) that leaves the sender and enters the first router is zero; i.e. D^0 has $I_{90}=0$ and $P(\mu-0 \le D \le \mu+0)=1$ since there is no variance in IPD of a homogeneous stream. We will use I_{90} in this section to quantify perturbations added by a network device or a network itself. Recall that the goal of *Chupja* is to achieve a BER less than 10%, and, as a result, we are interested in the range where 90% of D observed by a timing channel receiver is contained.

First, *switches do not add significant perturbations to IPDs when there is no cross traffic.* In particular, when a homogeneous packet stream is processed by a switch, I_{90} is always less than a few hundreds of nanoseconds, i.e. 90% of the received IPDs are within a few hundreds of nanoseconds from the IPD originally sent. Figure 7 displays the received IPD distribution measured after packets were processed and forwarded by one switch: The x-axis is the received IPD and the y-axis is the cumulative distribution function (CDF). Further, different lines represent different switches from Table 1. The characteristics of the original sender's homogeneous packet stream was a data rate of 1 Gbps with 1518B size packets, or (1518B, 1Gbps) for short, which resulted in an average IPD of 12.2 us (i.e. $\mu = 12.2$ us). We can see in Figure 7 that when μ is 12.2 us and ε is 0.1 us, $P(12.2-0.1 < D < 12.2+0.1) \ge 0.9$ is true for all switches. In general, the range of received IPDs was always bounded by a few hundreds of nanoseconds from the original IPD, regardless of the type of switch.

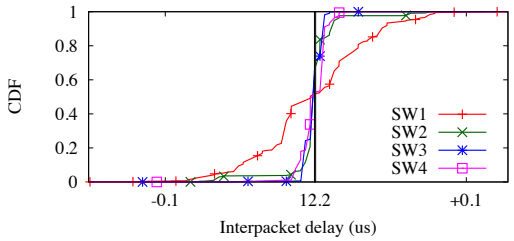


Figure 7: Comparison of IPDs after switches process a homogeneous packet stream (1518B, 1Gbps)

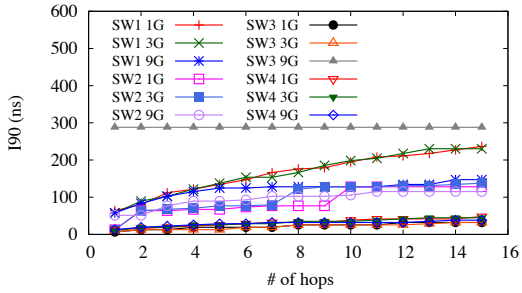
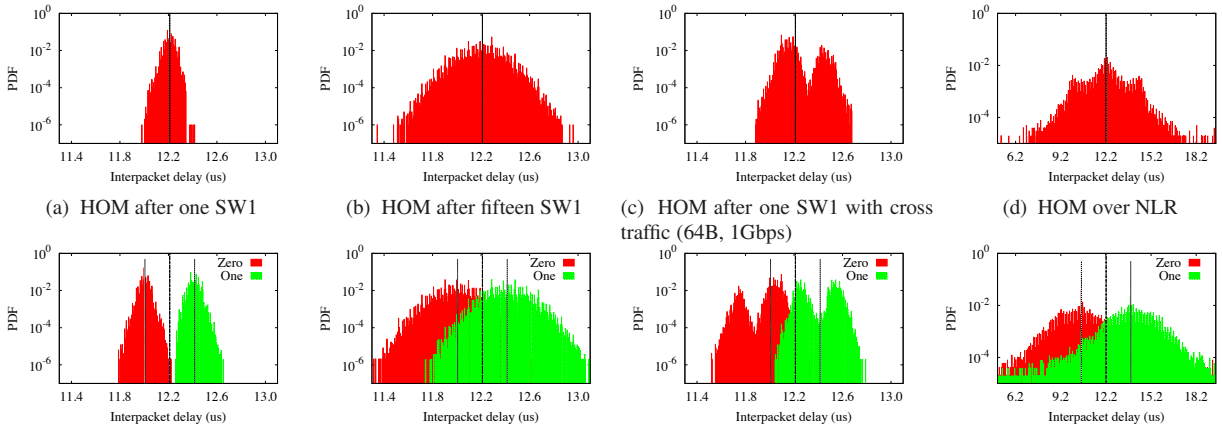


Figure 8: I_{90} comparison between various switches

Moreover, when a packet stream is processed by the same switch type, but for multiple hops, I_{90} increases linearly. Each packet suffers some perturbation, but the range of perturbation is roughly constant at every hop over different packet sizes [23] resulting in a linear increase in I_{90}. In Figure 8, we illustrate I_{90} for different switches, at different data rates (1, 3, and 9G), and as we increase the number of hops: The x-axis is the number of routing hops, y-axis is measured I_{90}, and each line is a different type of switch with a different packet stream data rate. Packet size was 1518B for all test configurations. One important takeaway from the graph is that I_{90} for the same switch shows similar patterns regardless of data rates, except SW3 9 Gbps. In particular, the degree of perturbation added by a switch is not related to the data rate (or the average interpacket delay). Instead, IPD perturbation is related to the number of hops, and the size of packet. Further, a second takeaway is that I_{90}



(a) HOM after one SW1 (b) HOM after fifteen SW1 (c) HOM after one SW1 with cross traffic (64B, 1Gbps) (d) HOM over NLR

(e) $\varepsilon = 256$ after one SW1 (f) $\varepsilon = 256$ after fifteen SW1 (g) $\varepsilon = 256$ after one SW1 with cross traffic (64B, 1Gbps) (h) $\varepsilon = 2048$ over NLR

Figure 9: Comparison of homogeneous streams and covert channel streams of (1518B, 1Gbps)

values after one hop are all less than 100 ns, except SW3 9 Gbps, and still less than 300 ns after fifteen hops. 300 ns is slightly greater than 256 /I/s (Table 3). Unfortunately, we do not have a definitive explanation on why I_{90} of SW3 9 Gbps is larger than any other case, but it is likely related to how SW3 handles high data rates.

Second, *switches treat IPDs of an encoded 'zero' bit and those of an encoded 'one' bit as uncorrelated distributions.* After encoding bits in a timing channel, there will be only two distinctive IPD values leaving the sender: $\mu + \varepsilon$ for 'one', and $\mu - \varepsilon$ for 'zero'. Let a random variable $D+$ be the IPDs of signal 'one', and $D-$ be those of signal 'zero'. We observed that the encoded distributions after one routing hop, D^1+ and D^1-, looked similar to the unencoded distribution after one routing hop, D^1. The similarity is likely due to the fact that at the sender the encoded distributions, D^0+ and D^0-, are each homogeneous packet streams (i.e. D^0, D^0+, and D^0- are all delta functions, which have no variance). For instance, using switch SW1 from Table 1, Figure 9a illustrates D^1 (the unencoded distribution of IPDs after one routing hop) and Figure 9e illustrates D^1+ and D^1- (the encoded distribution after one routing hop). The data rate and packet size was 1Gbps and 1518B, respectively, with $\varepsilon = 256$ /I/s for the encoded packet stream. We encoded the same number of 'zeros' and 'ones' randomly into the packet stream. Note the similarity in distributions between D^1 in Figure 9a and D^1+ and D^1- in Figure 9e. We observed a similarity in distributions among D^1, D^1+, and D^1- throughout different data rates and switches. We can conjecture that $D+$ and $D-$ are uncorrelated because the computed correlation coefficient between $D+$ and $D-$ is always very close to zero.

Because the distributions of D^1+ and D^1- are uncorrelated, we can effectively deliver bit information with appropriate ε values for one hop. If ε is greater than

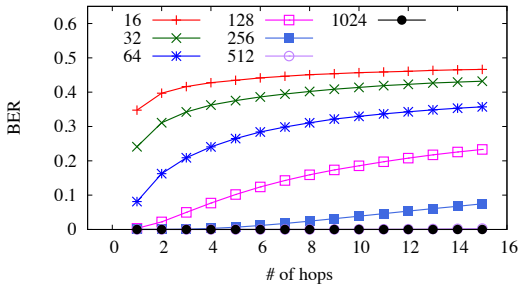


Figure 10: BER over multiple hops of SW1 with various ε values with (1518B, 1Gbps)

I_{90} of D^1, then 90% of IPDs of D^1+ and D^1- will not overlap. For example, when I_{90} is 64 ns, and ε is 256 /I/s (=204.8 ns), two distributions of D^1+ and D^1- are clearly separated from the original IPD (Figure 9e). On the other hand, if ε is less than I_{90} of D^1, then many IPDs will overlap, and thus the BER increases. For instance, Table 4 summarizes how BER of the timing channel varies with different ε values. From Table 4, we can see that when ε is greater than 64 /I/s, BER is always less than 10%. The key takeaway is that BER is *not* related with the data rate of the overt channel, rather it is related to I_{90}.

ε (/I/s)		16	32	64	128	256	512	1024
BER	1G	0.35	0.24	0.08	0.003	10^{-6}	0	0
	3G	0.37	0.25	0.10	0.005	10^{-5}	0	0
	6G	0.35	0.24	0.08	0.005	0.8×10^{-6}	0	0
	9G	0.34	0.24	0.07	0.005	0.0005	0.0004	0.0005

Table 4: ε and associated BER with (1518B, 1Gbps)

Third, *switches treat IPDs of an encoded 'zero' bit and those of an encoded 'one' bit as uncorrelated distributions over multiple switches and with cross traffic.* In particular, distributions D^n+ and D^n- are uncorrelated regardless of the number of hops and the existence of

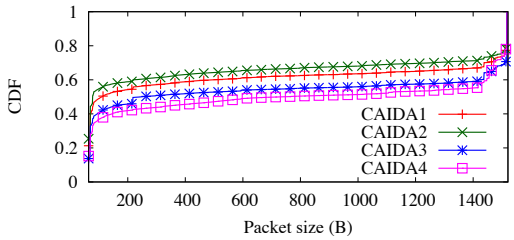


Figure 11: Packet size distributions of CAIDA traces

cross traffic. However, I_{90} becomes larger as packets go through multiple routers with cross traffic. Figures 9b and 9f show the distributions of D^{15}, $D^{15}+$, and $D^{15}-$ without cross traffic (Note that the y-axis is log-scale). The data rate and packet size was 1 Gbps and 1518B, respectively, with $\varepsilon = 256$ `/I/`s for the encoded packet stream. We conjecture that the distributions are still uncorrelated without cross traffic and after multiple hops of routers: The computed correlation coefficient was close to zero. Further, the same observation is true with the other switches from Table 1. Figure 10 shows BER over multiple hops of SW1. When ε is greater than 256 `/I/`s (=204.8 ns), BER is always less than 10% after fifteen hops. Recall that I_{90} of D after 15 hops of SW1 is 236 ns (Figure 8).

Figures 9c and 9g show the distributions after one routing hop when there was cross traffic: Distributions D^1, D^1+, and D^1- using overt data rate and packet size 1 Gbps and 1518B, respectively. The cross traffic was (64B, 1Gbps). We can see in the figures that there is still a similarity in D^1+ and D^1- even with cross traffic. However, I_{90} becomes larger due to cross traffic when compared to without cross traffic. Table 6 summarizes how I_{90} changes with cross traffic. We used five different patterns of cross traffic for this evaluation: 10-clustered (10C), 100-clustered (100C), one homogeneous stream (HOM), two homogeneous streams (HOM2), and random IPD stream (RND). A N-clustered packet stream consists of multiple clusters of N packets with the minimum interpacket gap (96 bits, which is 12 `/I/` characters) allowed by the standard [3] and a large gap between clusters. Note that a larger N means the cross traffic is *bursty*. For the RND stream, we used a geometric distribution for IPDs to create bursty traffic. In addition, in order to understand how the distribution of packet sizes affect I_{90}, we used four CAIDA traces [8] at different data rates to generate cross traffic (Table 5). With packet size and timestamp information from the traces, we reconstructed a packet stream for cross traffic with SoNIC. In the CAIDA traces, the distribution of packet sizes is normally a bimodal distribution with a peak at the lowest packet size and a peak at the highest packet size (Figure 11).

	Data Rate [Gbps]	Packet Rate [pps]	I_{90} [ns]	BER $\varepsilon = 512$	BER $\varepsilon = 1024$
CAIDA1	2.11	418k	736.0	0.10	0.041
CAIDA2	3.29	724k	848.1	0.148	0.055
CAIDA3	4.27	723k	912.1	0.184	0.071
CAIDA4	5.12	798k	934.4	0.21	0.08

Table 5: Characteristics of CAIDA traces, and measured I_{90} and BER

We observe that I_{90} increases with cross traffic (Table 6). In particular, bursty cross traffic at higher data rates can significantly impact I_{90}, although they are still less than one microsecond except 100C case. The same observation is also true using the CAIDA traces with different data rates (Table 5). As a result, in order to send encoded timing channel bits effectively, ε must increase as well. Figure 9d and 9h show the distributions of IPDs over the NLR. It demonstrates that with external traffic and over multiple routing hops, sufficiently large ε can create a timing channel.

Data Rate [Gbps]	Packet Size [Byte]	I_{90} 10C	100C	HOM	HOM2	RND
0.5	64	79.9	76.8	166.45	185.6	76.8
	512	79.9	79.9	83.2	121.6	86.3
	1024	76.8	76.8	80.1	115.2	76.8
	1518	111.9	76.8	128.0	604.7	83.2
1	64	111.9	108.8	236.8	211.2	99.3
	512	115.2	934.4	140.8	172.8	188.9
	1024	111.9	713.5	124.9	207.9	329.5
	1518	688.1	1321.5	64.0	67.1	963.3

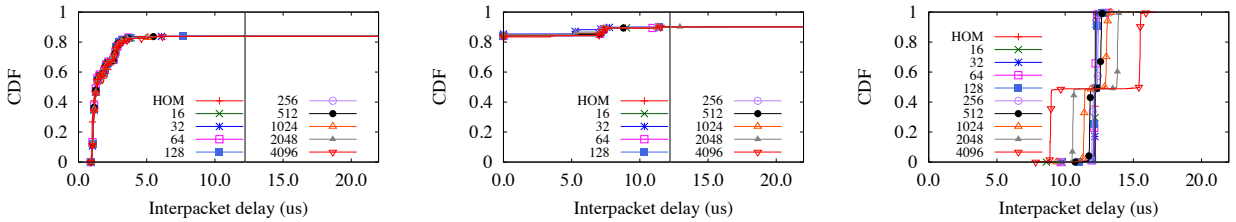
Table 6: I_{90} values in nanosecond with cross traffic.

Summarizing our sensitivity analysis results, I_{90} is determined by the characteristics of switches, cross traffic, and the number of routing hops. Further, I_{90} can be used to create a PHY timing channel like *Chupja*. In particular, we can refine the relation between I_{90} and ε^* (the minimum ε measured to achieve a BER of less than 10%). Let I_{90}^+ be the minimum ε that satisfies $P(D > \mu - \varepsilon) \geq 0.90$ and let I_{90}^- be the minimum ε that satisfies $P(D < \mu + \varepsilon) \geq 0.90$ given the average interpacket delay μ. Table 7 summarizes this relationship between $\varepsilon*$, I_{90} and $\max(I_{90}^+, I_{90}^-)$ over the NLR (Figure 6) and our small network (Figure 4).

Network	Workload	I_{90}	$\max(I_{90}^+, I_{90}^-)$	ε^* (ns)
Small network	Light	1065.7	755.2	819.2
Small network	Medium	1241.6	1046.3	1638.4
Small network	Heavy	1824.0	1443.1	1638.4
NLR		2240.0	1843.2	1638.4

Table 7: Relation between ε, I_{90}, and $\max(I_{90}^+, I_{90}^-)$ over different networks with (1518B, 1Gbps). Values are in nanosecond.

In our small network, BER is always less than 10% when ε is greater than $\max(I_{90}^+, I_{90}^-)$. On the other hand, we were able to achieve our goal BER over the NLR when ε^* is slightly less than $\max(I_{90}^+, I_{90}^-)$. Because we



(a) Kernel timestamping. (b) Zero-copy timestamping. (c) Hardware timestamping.

Figure 12: Comparison of various timestampings. Each line is a covert channel stream of (1518B, 1Gbps) with a different ε value.

do not have control over cross traffic in the NLR, I_{90} varied across our experiments.

4.4 Detection

In order to detect timing channels, applying statistical tests to captured IPDs is widely used. For example, the adversary can use regularity, similarity, shape test, and entropy test of IPDs in order to detect potential timing channels [11, 12, 16, 32]. The same strategies can be applied to *Chupja*. Since our traffic is very *regular*, those algorithms could be easily applied to detect *Chupja*. However, we argue that none of these algorithms will work if the IPG modulations cannot be observed at all. In particular, endhost timestamping is too inaccurate to observe fine-grained IPG modulations whereas *Chupja* modulates IPGs in hundreds of nanoseconds to create a timing channel. In fact, the accuracy of endhost timestamping is at most microsecond resolution. Specialized NICs can provide a few hundreds of nanosecond resolution. In this section, we demonstrate that endhost or hardware timestamping is not sufficient to detect *Chupja* timing channels. We focus on measuring and comparing an endhost's ability to accurately timestamp arrival times (i.e. accurately measure IPDs) since the ability to detect a PHY timing channel is dependent upon the ability to accurately timestamp the arrival time of packets. As a result, we do not discuss statistical approaches further.

There are mainly three places where packets can be timestamped at an endhost: Kernel, userspace, and hardware (NIC). Kernel network stacks record arrival times of packets upon receiving them from a NIC. Since so many factors are involved during the delivery of a packet to kernel space, such as DMA transaction, interrupt routines, and scheduler, kernel timestamping can be inaccurate in a high-speed network. As a result, userspace timestamping will also be inaccurate because of delays added due to transactions between kernel and userspace. To reduce the overhead of a network stack between kernel and userspace and between hardware and kernel, a technique called zero-copy can be employed to improve the performance of userspace network applications. An example of a zero-copy implementation is Netmap [33]. In Netmap, packets are delivered from a NIC directly to a memory region which is shared by a userspace application. This zero-copy removes expensive memory operations and bypasses the kernel network stack. As a result, Netmap is able to inject and capture packets at line speed in a 10 GbE network with a single CPU. Therefore, detection algorithms can exploit a platform similar to Netmap to improve the performance of network monitoring applications. We call this *zero-copy timestamping*. In hardware timestamping, a NIC uses an external clock to timestamp incoming packets at a very early stage to achieve better precision. The accuracy of timestamping is determined by the frequency of an external clock. Unfortunately, hardware timestamping is not often available with commodity NICs. However, we did include in our evaluation a specialized NIC, the Sniffer 10G [5], which can provide 500 ns resolution for timestamping.

In order to compare *kernel*, *zero-copy*, and *hardware* timestamping, we connected a SoNIC server and a network monitor directly via an optical fiber cable, generated and transmitted timing channel packets to a NIC installed in the network monitor, and collected IPDs using different timestampings. The network monitor is a passive adversary built from a commodity server. Further, we installed Netmap in the network monitor. Netmap originally used the `do_gettimeofday` for timestamping packets in kernel space, which provides only microsecond resolution. We modified the Netmap driver to support nanosecond resolution instead. For this evaluation, we always generated ten thousand packets for comparison because some of the approaches discarded packets when more than ten thousand packets were delivered at high data rates.

Figure 12 illustrates the results. Figure 12a demonstrates the effectiveness of *kernel timestamping* of a timing channel with various IPG modulation (ε) values. The data rate of the overt channel was 1 Gbps and the packet size was 1518 bytes. The x-axis is interpacket delays (IPDs) in microsecond and y-axis is a cumulative distribution function (CDF). The vertical line in the middle is the original IPD (=12.2 us) of *Chupja*. In order to detect *Chupja*, the timestamping CDF would be cen-

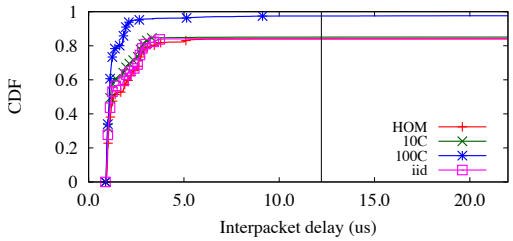


Figure 13: Kernel timestamping with (1518B, 1Gbps).

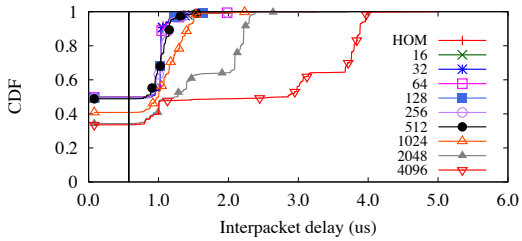


Figure 14: Hardware timestamping with (64B, 1Gbps)

tered around the vertical line at ≈12.2 us. Instead, as can be seen from the graph, all measured kernel timestamps were nowhere near the vertical line regardless of ε values (ε varied between ε=0 [HOM] to ε=4096 `/I/s`). As a result, kernel timestamping cannot distinguish a PHY covert channel like *Chupja*. In fact, even an i.i.d random packet stream is inseparable from other streams (Figure 13). Unfortunately, *zero-copy timestamping* does not help the situation either (Figure 12b). Netmap does not timestamp every packet, but assigns the same timestamp value to packets that are delivered in the one DMA transaction (or polling). This is why there are packets with zero IPD. Nonetheless, Netmap still depends on underlying system's timestamping capability, which is not capable.

On the other hand, *hardware timestamping* using the Sniffer 10G demonstrates enough fidelity to detect *Chupja* when modulation (ε) values are larger than 128 `/I/s` (Figure 12c). However, hardware timestamping still cannot detect smaller changes in IPDs (i.e. modulation, ε, values smaller than 128 `/I/s`), which is clear with a timing channel with smaller packets. A timing channel with 64 byte packets at 1 Gbps is not detectable by hardware timestamping (Figure 14). This is because packets arrive much faster with smaller packets making IPGs too small for the resolution of hardware to accurately detect small IPG modulations.

The takeaway is that to improve the possibility of detecting *Chupja*, which modulates IPGs in a few hundreds of nanoseconds, a network monitor (passive adversary) must employ hardware timestamping for analysis. However, using better hardware (more expensive and sophisticated NICs) still may not be sufficient; i.e. for much finer timing channels. Therefore, we can conclude that a PHY timing channel such as *Chupja* is invisible to a software endhost. However, a hardware based solutions with fine-grained capability [1] may be able to detect *Chupja*.

5 Countermeasures

So far, we demonstrated that covert timing channels implemented in the physical layer can leak secret information without being detected. Such channels are great threats to a system's security, and should be prevented or detected. However, as we discussed, detecting a PHY timing channel is not easy with commodity components. As a result, a network administrator who is worried about information leaks from the network must employ *capable* network appliances for prevention or detection: *PHY*-enhanced network jammers or monitoring appliances could potentially prevent or detect the existence of a covert channel.

6 Conclusion

In this paper, we presented *Chupja*, a PHY covert timing channel that is high-bandwidth, robust and undetectable. The covert timing channel embeds secret messages into interpacket gaps in the physical layer by modulating interpacket gaps at sub-microsecond scale. We empirically demonstrated that our channel can effectively deliver 81 kilobits per second over nine routing hops and thousands miles over the Internet, with a BER less than 10%. As a result, a *Chupja* timing channel works in practice and is undetectable by software endhosts since they are not capable of detecting such small modulations in interpacket gaps employed by *Chupja*. Now that we have demonstrated that a PHY covert timing channel is a security risk, future directions include efficient methods to prevent or detect such covert channels.

7 Availability

The *Chupja* and SoNIC source code is published under a BSD license and is freely available for download at `http://sonic.cs.cornell.edu`

8 Acknowledgements

This work was partially funded and supported by an IBM Faculty Award received by Hakim Weatherspoon, DARPA (No. D11AP00266), DARPA MRC, NSF CAREER (No. 1053757), NSF TRUST (No. 0424422), NSF FIA (No. 1040689), NSF CiC (No. 1047540), and NSF EAGER (No. 1151268). We would like to thank our shepherd, Shyamnath Gollakota, and the anonymous reviewers for their comments. We further recognize the engineers who helped establish and maintain the network infrastructure on which we performed these experiments: Eric Cronise (Cornell Information Technologies); Scott Yoest, and his team (Cornell Computing and Information Science Technical Staff). We also thank Danial Freedman, Ethan Kao, Erluo Li, and Chiun Lin Lim for reviewing and comments.

References

[1] Endace dag network cards. http://www.endace.com/endace-dag-high-speed-packet-capture-cards.html.

[2] Endaceprobes. http://www.endace.com/endace-high-speed-packet-capture-probes.html.

[3] IEEE Standard 802.3-2008. http://standards.ieee.org/about/get/802/802.3.html.

[4] Intel Westmere. http://ark.intel.com/products/codename/33174/Westmere-EP.

[5] Myricom Sniffer10G. http://www.myricom.com/sniffer.html.

[6] Napatech. http://www.endace.com/endace-high-speed-packet-capture-probes.html.

[7] NAPI. http://linuxfoundation.org/en/Net:NAPI.

[8] The CAIDA UCSD Anonymized Internet Traces. http://www.caida.org/datasets/.

[9] Wildpackets. http://www.wildpackets.com/products/network_recorders.

[10] Trusted computer system evaluation criteria. Tech. Rep. DOD 5200.28-STD, National Computer Security Center, December 1985.

[11] BERK, V., GIANI, A., AND CYBENKO, G. Detection of covert channel encoding in network packet delays. Tech. Rep. TR2005-536, Department of Computer Science, Dartmouth College, November 2005.

[12] CABUK, S., BRODLEY, C. E., AND SHIELDS, C. IP Covert Timing Channels: Design and Detection. In *Proceedings of the 11th ACM conference on Computer and Communications Security* (2004).

[13] 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).

[14] FISK, G., FISK, M., PAPADOPOULOS, C., AND NEIL, J. Eliminating steganography in internet traffic with active wardens. In *Revised Papers from the 5th International Workshop on Information Hiding* (2003).

[15] FREEDMAN, D. A., MARIAN, T., LEE, J. H., BIRMAN, K., WEATHERSPOON, H., AND XU, C. Exact temporal characterization of 10 gbps optical wide-area network. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (2010).

[16] GIANVECCHIO, S., AND WANG, H. Detecting covert timing channels: an entropy-based approach. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (2007).

[17] GILES, J., AND HAJEK, B. An information-theoretic and game-theoretic study of timing channels. *IEEE Transactions on Information Theory 48*, 9 (Sept. 2002), 2455–2477.

[18] HAN, S., JANG, K., PARK, K., AND MOON, S. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference* (2010).

[19] HANDLEY, M., KREIBICH, C., AND PAXSON, V. Network intrusion detection: Evasion, traffic normalization. In *Proceedings of the 10th USENIX Security Symposium* (2001).

[20] KUNDUR, D., AND AHSAN, K. Practical internet steganography: Data hiding in ip. In *Proceedings of Texas workshop on Security of Information Systems* (2003).

[21] LAMPSON, B. W. A note on the confinement problem. *Communications of the ACM 16*, 10 (October 1973), 613–615.

[22] LEE, K. S., WANG, H., AND WEATHERSPOON, H. SoNIC: Precise Realtime Software Access and Control of Wired Networks. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (2013).

[23] LIM, C. L., LEE, K. S., WANG, H., WEATHERSPOON, H., AND TANG, A. Packet clustering introduced by routers: Modeling, analysis and experiments. In *Proceedings of the 48th Annual Conference on Information Sciences and Systems* (2014).

[24] LIU, Y., GHOSAL, D., ARMKNECHT, F., SADEGHI, A.-R., SCHULZ, S., AND KATZENBEISSER, S. Hide and Seek in Time: Robust Covert Timing Channels. In *Proceedings of the 14th European Conference on Research in Computer Security* (2009).

[25] LIU, Y., GHOSAL, D., ARMKNECHT, F., SADEGHI, A.-R., SCHULZ, S., AND KATZENBEISSER, S. Robust and Undetectable Steganographic Timing Channels for i.i.d. Traffic. In *Proceedings of the 12th International Conference on Information Hiding* (2010).

[26] MALAN, G. R., WATSON, D., JAHANIAN, F., AND HOWELL, P. Transport and application protocol scrubbing. In *Proceedings of IEEE Conference on Computer Communications* (2000).

[27] MARIAN, T., LEE, K. S., AND WEATHERSPOON, H. Netslices: Scalable multi-core packet processing in user-space. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2012).

[28] MURDOCH, S. J., AND LEWIS, S. Embedding covert channels into tcp/ip. In *Proceedings of 7th Information Hiding workshop* (2005).

[29] NLR. National Lambda Rail. http://www.nlr.net/.

[30] OLSSON, R. pktgen the linux packet generator. In *Proceeding of the Linux symposium* (2005).

[31] PADLIPSKY, M. A., SNOW, D. W., AND KARGER, P. A. Limitations of end-to-end encryption in secure computer networks. Tech. Rep. ESD-TR-78-158, Mitre Corporation, August 1978.

[32] PENG, P., NING, P., AND REEVES, D. S. On the secrecy of timing-based active watermarking trace-back techniques. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

[33] RIZZO, L. Netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (2012).

[34] ROWLAND, C. H. Covert channels in the tcp/ip protocol suite. *First Monday* (July 1997).

[35] RUTKOWSKA, J. The implementation of passive covert channels in the linux kernel. In *Proceedings of Chaos Communication Congress* (2004).

[36] SHAH, G., MOLINA, A., AND BLAZE, M. Keyboards and covert channels. In *Proceedings of the 15th conference on USENIX Security Symposium* (2006).

[37] ZANDER, S., ARMITAGE, G., AND BRANSH, P. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys and Tutorials 9*, 3 (October 2007), 44–57.

cTPM: A Cloud TPM for Cross-Device Trusted Applications

Chen Chen†, Himanshu Raj, Stefan Saroiu, and Alec Wolman
Microsoft Research and †CMU

Abstract:

Current Trusted Platform Modules (TPMs) are ill-suited for cross-device scenarios in trusted mobile applications because they hinder the seamless sharing of data across multiple devices. This paper presents cTPM, an extension of the TPM's design that adds an additional root key to the TPM and shares that root key with the cloud. As a result, the cloud can create and share TPM-protected keys and data across multiple devices owned by one user. Further, the additional key lets the cTPM allocate cloud-backed remote storage so that each TPM can benefit from a trusted real-time clock and high-performance, non-volatile storage.

This paper shows that cTPM is practical, versatile, and easily applicable to trusted mobile applications. Our simple change to the TPM specification is viable because its fundamental concepts – a primary root key and off-chip, NV storage – are already found in the current specification, TPM 2.0. By avoiding a clean-slate redesign, we sidestep the difficult challenge of re-verifying the security properties of a new TPM design. We demonstrate cTPM's versatility with two case studies: extending Pasture with additional functionality, and re-implementing TrInc without the need for extra hardware.

1 Introduction

People are increasingly relying on more than one mobile device. Recent news reports estimate that: the average US consumer owns 1.57 mobile devices [8]; Singapore has 7.8 million mobile devices, which translates to 150% mobile penetration [36]; and the average Australian will own five mobile devices by 2040 [37]. Given this trend, mobile platforms are recognizing the need for "cross-device" functionality that automatically synchronizes photos, videos, apps, data, and even games across all devices owned by a single user.

Simultaneously, laptops, smartphones, and tablets are increasingly incorporating trusted computing hardware. For example, Google's Chromebooks use TPMs to prevent firmware rollbacks and to store and attest user's data encryption keys [11]. Windows 8 (on tablets and phones) offers BitLocker full-disk encryption [21] and virtual smart cards [23] using TPMs. Recent research leverages TPMs to build new trusted mobile services [30, 32, 9, 17, 14], new trusted cloud services [31], and new operating systems [33].

Unfortunately, these two trends may be at odds: trusted hardware, such as the trusted platform module (TPM), does not provide good support for cross-device functionality. By design, TPMs offer a hardware root-of-trust bound to a single, standalone device. TPMs come equipped with encryption keys whose private parts never leave the TPM hardware chip, reducing the possibility those keys may be compromised. The tension between single-device TPM guarantees and the need for cross-device sharing makes it difficult for trusted applications to cope with multi-device scenarios. For example, Pasture [14], a TPM-based secure offline data access system that can be used for movie rentals, limits all its guarantees to one device due to TPM limitations. Similarly, Windows TPM-based virtual smart cards are single-device only – users have to provision and renew their credentials separately on each device they own.

Support for cross-device sharing requires altering the TPM design, which raises the following question: Can a small-scale TPM design change overcome these limitations? While a clean-slate TPM re-design could provide a variety of additional security properties, there are two pragmatic reasons why a smaller change is preferable. First, TPMs have undergone a decade of API and implementation revisions to reduce the likelihood of vulnerabilities. A clean-slate re-design would demand considerable time and effort to provide a mature codebase. Second, TPM manufacturers would more willingly adopt smaller and simpler changes.

This paper proposes a single, simple design change to the TPM, called cTPM, that overcomes the limitations noted above by equipping the TPM with one additional root key that is shared with the cloud. This key lets trusted applications overcome their cross-device limitations with the cloud's assistance. It ensures that the cloud can control only a portion of TPM resources: those encrypted with the shared key. The cloud remains restricted from accessing the TPM resources protected by all other device-local, TPM root keys. We verified the security of the communication protocol between the TPM and the cloud using a protocol verifier [3].

The new key also lets cTPM allocate non-volatile (NV) storage in the cloud. The cTPM's remote storage enables the cloud to provide a trusted, synchronized and highly accurate source of time by periodically recording the time to remote storage. Today's TPMs lack a trusted source of time (i.e., a trusted real-time clock). Although the TPM provides an internal trusted timer, this timer

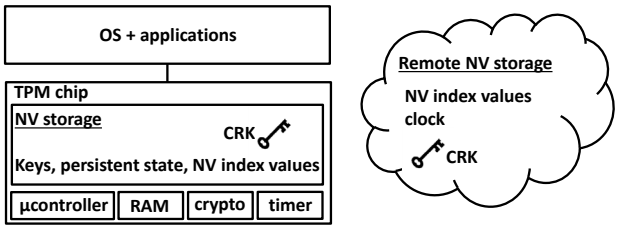


Figure 1. **Diagram of cTPM architecture.**

alone is insufficient to build a trusted real-time clock. Finally, the cTPM's remote cloud storage offers TPM applications large amounts of NV storage and lets them perform frequent NV writes. In contrast, TPM chips cannot offer such resources because they suffer from serious resource and performance limitations. These limitations drastically reduce the use cases for TPMs in both mobile and server scenarios, and have led researchers investigate alternatives to TPMs such as trusted devices whose storage offers increased performance [16]. Figure 1 shows a diagram of cTPM's architecture.

We demonstrate the benefits of cTPM by presenting two case studies. First, the cloud's ability to manage a portion of the TPM's state provides Pasture [14] with additional functionality. With cTPM, Pasture can extend its guarantees across all devices owned by a single user and can support server-side revocation, an operation not offered by the original Pasture protocol. Further, cTPM's trusted clock enables Pasture to grant data access at a specific time in the future (e.g., *make this movie available on Friday at midnight*). Second, the high-performance nature of cTPM's remote storage improves the performance of applications that require frequent writes. We re-implement TrInc [16] without the need of extra hardware (TrInc requires a smartcard).

2 Background

TPM Primer. At manufacturing time, TPM chips are provisioned with a couple of public/private key-pairs for cryptography (i.e., digital signatures and asymmetric encryption). The TPM design guarantees that the private keys of these *root* key-pairs never leave the TPM, thereby reducing the possibility of compromise. TPMs can also generate public/private key-pairs with private keys stored in the TPM's NV storage. However, TPMs have limited NV storage and thus cannot store many such key-pairs.

TPMs are equipped with a set of platform configuration registers (PCRs) guaranteed to be reset upon a computer reboot. PCRs are primarily used to store fingerprints of a portion of the software booting on a computer (e.g., the BIOS, firmware, and OS bootloader); Chromebooks [11] and BitLocker [21] use PCRs in this way.

TPMs can perform cryptographic algorithms for encrypting, authenticating, and attesting data. Implementing functionality beyond that offered by TPMs in a trustworthy manner can be done using secure execution mode, a form of hardware protection offered by x86 CPUs. Intel's secure execution architecture, called Trusted Execution Technology (TXT), offers a runtime environment strongly isolated from other software running on the computer.

The TPM spec does not provide minimum performance requirements, and, as a result, today's commodity TPMs are slow and inefficient [19, 14]. TPM vendors have little incentive to use faster but more expensive internal parts when building their TPM chips. This performance handicap has limited the use of TPMs to scenarios that do not require fast or frequent operations. However, no technological constraints prevents a hardware vendor from building a high-performance TPM.

Describing the full functionality of a TPM is beyond the scope of this paper. Ryan [29] and Challener et al. [5] provide good overviews of how TPMs work, although the TPM specs [39] remain the authoritative source for a full description of TPM functionality.

TPM 2.0. The Trusted Computing Group (TCG) is currently defining the specification for TPM version 2.0 [38], the next version of the TPM. TPM 2.0 offers several improvements, including a more complete set of cryptographic algorithms, i.e., SHA-2 and elliptic curve cryptography (ECC) in addition to SHA-1 and RSA offered by TPM 1.2. TPM 2.0 also provides more PCRs and supports more flexible authorization policies that control access to TPM-protected data. Finally, TPM 2.0 provides a reference implementation, while TPM 1.2 provides only an open-source implementation developed by a third party [10]. A complete list of differences between the two versions is provided by the TCG [38].

In TPM 2.0, three entities can control the TPM's resources: the platform manufacturer, the owner, and the privacy administrator. The TPM 2.0 spec *control domain* refers to the specific resources that each entity controls. The platform firmware control domain overseen by the platform manufacturer updates the TPM firmware as needed. The owner control domain protects keys and data on behalf of users and applications. The privacy administrator control domain safeguards privacy-sensitive TPM data. This role can be played by anyone; for example, in an enterprise the IT department acts as the privacy administrator for all its machines' TPMs.

Each TPM 2.0 control domain has a primary seed, which is a large, random value permanently stored in the TPM. Primary seeds are used to generate symmetric/asymmetric keys and proofs for each control domain.

3 Motivation

This section first describes how the additional TPM functionality can be implemented at present and why this

approach is problematic. We then discuss specific feature limitations of existing TPMs for cross-device sharing, trusted clock, and NV storage. Finally, we describe how cTPM addresses each limitation.

3.1 Secure Execution Mode Limitations

Extending the TPM functionality can be done at present by leveraging its extensibility mechanism, which is a *secure execution mode* integrated into the system CPU. Both Intel's TXT and AMD's SEM are extensibility mechanisms for the TPM – they enable the development of trusted computing features not easily achieved solely through the built-in TPM commands. Unfortunately, major stumbling blocks prevent a secure execution mode from providing needed features. The first stumbling block is performance; to use the secure execution mode, CPU interrupts must be disabled, and, in a multiprocessor system, only one CPU core can be enabled. Entering this mode requires the OS to save its state and suspend execution, operations that are relatively heavyweight. Second, none of today's smartphones and tablets includes a CPU that supports TXT or SEM; these features are provided only on laptops and desktops sold to enterprises. Third, even if they did, using secure execution mode is remarkably difficult. It requires support from the motherboard chipset, BIOS , and, in the case of Intel's TXT, an additional chipset-specific authenticated code module. Also, once in secure execution mode, the code only has access to a "barebones" machine without any I/O, OS, or library support. Building such support without relying on interrupts may be challenging. We know of no production software that uses secure execution mode[1].

Section 6 will describe how cTPM solves this problem in a simpler way that does not require the use of secure execution mode. Despite cTPM's benefits, however, changing the TPM design raises a legitimate concern: Does the verification cost of introducing TPM changes outweigh the additional benefits of the new design? Although we cannot provide a reliable estimate of these costs, we deliberately kept our design changes minimal. The cTPM design affects only the TPM commands that access NV data (to indicate whether operations are local or remote) and adds three TPM commands that synchronize data between the device TPM and the cloud TPM. The vast majority of the TPM logic remains the same.

3.2 Limitation 1: Cross-Device Data Sharing

Current TPM abstractions offer guarantees about one single computer, and the TPM's hardware protection mechanisms do not extend across devices. For example, the TPM owner domain provides an isolation mechanism for only a single TPM. When a new owner takes ownership of the TPM, they cannot access the previous owner's TPM-protected secrets. When the same user owns two different TPMs (on two different devices), the owner domains of each TPM remain isolated and cannot jointly offer hardware-based protection of the user's keys and data. Thus, mobile services cannot rely on TPMs alone to enable secure data sharing across devices.

3.2.1 Secure Key Exchange

To better illustrate these challenges, we now describe in-depth how to perform secure key exchange between two TPM-equipped mobile devices, a critical building block in enabling secure data sharing across devices. Key exchange is a common bootstrapping step used in security protocols that provide authentication and encryption, such as SSL, SSH, VPNs, etc. The TPM offers hardware-protection for cryptographic keys. Thus, even if a system were compromised, the key itself would remain protected. A desirable property for secure key exchange between two TPM-equipped devices is the establishment of a secure communication channel even when both are infected by malware. This requires TPMs to perform the cryptographic steps for key exchange without leaking the key to the malware.

Unfortunately exchanging a key securely between two parties is notoriously challenging in practice because of the *identity problem* – one party needs to verify the correct identity of the other. One way to do this is to use a public key infrastructure (PKI) where each party applies to a certificate authority for a digital certificate, which serves for others as a non-tamperable authentication of identity.

Thus, the TPM specification does not directly provide an implementation of any secure key exchange protocol. Because TPMs lack the functionality of a key exchange protocol (e.g., Diffie-Hellman), two TPMs can exchange keys only by performing a *one-time key migration* from one device's TPM to another in the *absence of malware*. Without either of these properties, malware could either migrate the key to a malicious device or obtain a copy of it during migration.

Figure 2 shows the pseudo-code a device must execute to generate a key that can be shared with another device. This code requires use of the secure execution mode (i.e., Intel's TXT or AMD's SEM) to reduce its TCB and thereby reduce the likelihood of the presence of malware. To ensure that key migration is a one-time only operation, the code assigns a migration policy to the key based on a secret (denoted by S in the pseudo-code). Once S is destroyed, the key can no longer be migrated.

[1] See the Flicker [19] Web page for details on the difficulty of finding the appropriate hardware and software to use SEM (https://sparrow.ece.cmu.edu/group/flicker.html).

```
//Reduce the likelihood of malware.
1. Enter secure execution mode

//Create a shared key K. K is migrateable
//only by knowing a secret S.
2. Create symmetric key K
3. Create secret S
4. Set migration policy of K to a secret S

//Secure identity verification of device 2.
5. Verify identity of device 2

//Encrypt K with device 2's public key. Secret S
//is needed for this operation.
6. Using S, create migrateable copy of K for device 2

//Permanently disable K's ability to migrate.
7. Permanently destroy S

//Exit SEM and send encrypted K to device 2.
8. Leave secure execution mode
9. Send migrateable copy of K to device 2
```

Figure 2. **Pseudo-code for secure key exchange.**

3.3 Limitation 2: Trusted Clock

Today's TPMs do not offer a trusted real-time clock. Instead, the TPM combines a trusted timer with a secure, non-volatile counter. For every tick received, the TPM increments the value of a counter stored in memory. For every n increments of this counter, the counter value is persisted to the TPM's NV storage. The TPM has an estimate of the timer's frequency and thus has an approximate notion of time. However, this mechanism can keep track of time only when the TPM is running (and *not* when the platform is powered off). Because the counter value is persisted only every n increments, this mechanism does not even provide a guarantee of monotonicity. Upon a reboot, the timer is rolled back to the last persisted counter value violating monotonicity. The TPM's timer mechanism solely guarantees that as long as the platform does not reboot, the timer will move forward. As such, it can provide an approximate time-since-boot.

This mechanism is inadequate for offering real-time guarantees that would be useful for offline content access. For example, movie studios already charge a premium to make a movie available on home theaters on the day of release. Although TPMs can provide offline access securely, they cannot offer *make the following movie available for watching next Friday at midnight*.

3.4 Limitation 3: NV Storage

The TPM's NV storage is inadequate for applications that require frequent writes or require large amounts of trusted storage. For example, previous work [16] has shown that a trusted module offering a monotonic counter and a key solves several problems in distributed systems that stem from participants' ability to equivocate. Unfortunately, even though TPMs offer this functionality, their implementation of NV storage cannot meet the write frequency requirements of distributed systems protocols. The TPM specification dictates the inclusion of monotonic counters, but the spec requires only the ability to increment these counters at a very slow place (e.g., once every five seconds), which is insufficient for high-event applications such as networked games [16]. Similarly, although the TPM specification mandates access-controlled, non-volatile storage, most implementations provide only 1,280 bytes of NVRAM [26]. These limitations have led researchers to seek alternative designs for trusted devices [16].

3.5 How cTPM Overcomes These Limitations

To address these limitations, we propose cTPM, a modification to the TPM design that includes an additional cloud control domain. This domain offers the same functionality as the owner domain except that its primary seed is also shared with the cloud. Sharing the seed with the cloud allows both cTPM and the cloud to generate the same cloud root key (see Section 5.3 for details). Combining the cloud root key with remote storage lets cTPM: 1) better share data via the cloud, 2) have access to a trusted real-time clock, and 3) have access to remote NV storage that supports a large quantity of storage, and high frequency writes.

cTPM's design facilitates data sharing. The pre-shared primary seed lets the cloud effectively act as a PKI. The cloud and the device's TPM can use this shared secret to encrypt and authenticate their messages to each other. The identity problem has now been "pushed" to ensuring that the cloud primary seed is shared securely between cTPM and the cloud. This initial sharing step should be done at cTPM manufacturing time when the cTPM's three other primary seeds are provisioned.

The cloud domain also equips cTPM with a trusted clock using a protocol similar to the Time Protocol described in RFC 868 [27]. Once the clock value is obtained from the cloud, cTPM uses its local timer to advance the clock. It has a global variable that dictates how often it should re-synchronize the clock; the TPM owner sets this variable whose value default is one day.

Finally, cTPM uses the cloud for additional NV storage to overcome TPM NV storage limitations. There are no limits on how much additional NV storage the cloud can provide to a single cTPM. A portion of the physical cTPM chip's RAM is thus allocated as a local cache for the cloud-backed NV storage. The performance of cTPM cloud-backed NV storage exceeds that of the TPM because TPM NV accesses are no longer needed.

4 Trust Assumptions and Threat Model

4.1 Trusting the Cloud

All the new cTPM functionality associated with the cloud domain assumes the cloud is trustworthy and

not compromised by malware. While everyone may not agree with this assumption, cloud providers have more incentives and resources to monitor and eliminate malware than average users. Security-conscious cloud providers could use secure hypervisors with a small TCB [18], narrow interfaces [24], or increased protection against cloud administrators [40, 28].

Whether using a TPM or not, a cloud compromise would already affect the security of a mobile service relying on the cloud for its functionality. However, even if the cloud were compromised, all secrets protected by the TPM-specific control domains other than the cloud domain would remain secure. For example, all device-specific secrets protected in the owner's control domain (i.e., using TPM's SRK) would remain uncompromised.

In the event that the cloud were compromised, cTPM could no longer offer its security guarantees. To recover would require changing the cloud seed (rekeying). To do so, we see only two options: issue a new device to the user, or implement a secure rekeying mechanism by visiting an authorized store (e.g., a mobile operator store such as AT&T) where the staff has specialized hardware to perform a secure rekey. A rekey would also be required whenever devices change ownership. While cTPM lets the owner *clear* the device (i.e., erase its cloud seed and all secrets protected by it), the new owner would need to physically visit a store to obtain a new seed.

One alternative to the current cTPM design is to have a trusted 3rd party offer the remote cTPM functionality rather than the cloud. For example, the cTPM could be offered by a TPM manufacturer rather than by the cloud. However, we have not fully pursued such an alternative cTPM design.

4.2 Threat Model

Our threat model resembles that of traditional TPMs: all software attacks are in scope (including side-channel attacks) because cTPM is isolated from the host platform and can therefore provide its security guarantees even if the host were compromised (e.g., infected with malware). However, physical attacks are out of scope. Such attacks include decapsulation, microprobing, or focused ion beaming the TPM chip [34], monitoring its internal buses [35], or inserting traffic on the bus between the CPU and the TPM. Furthermore, DoS attacks in which the (untrusted) operating system or applications deny access to the cTPM or to the cloud are out of scope. For example, a TPM can be put in lockout mode if an application attempts to "guess" an authorization value (e.g., a "password") to a secret it protects. During the lockout, the TPM refuses to serve any requests to protected secrets made by any application. Once the lockout timeout expires, the TPM exits lockout and can receive additional requests. TPMs today are thus susceptible to DoS attacks by applications that repeatedly attempt to guess the wrong authorization values until the TPM enters lockout and refuses to answer additional requests.

Another class of attacks specific to the cTPM stems from our use of remote cloud storage. The (untrusted) operating system could drop, corrupt, or re-order messages from the cloud. Even worse, it could delay messages from the cloud in an effort to serve stale data to the TPM. All such attacks are in scope and addressed by cTPM; for example, to ensure freshness, cTPM uses a local timer to timeout any pending requests not yet serviced.

cTPM has a dual relationship with the cloud. On one hand, it trusts the cloud with any keys and data the cloud stores in the cloud-backed NV storage. The cloud must offer increased assurance that these keys are not compromised; for example, cloud-stored keys should be protected against malware, malicious administrators [31], and side-channel attacks [41]. On the other hand, cTPM has additional local NV storage that protects its own secrets from the cloud, as needed. We believe that this dual relationship helps mobile services share data across devices, yet does not place unlimited trust in the cloud. The owner or privacy administrator can always use their own control domain to protect secrets from the cloud.

5 cTPM High-Level Design

The cTPM design extends the TPM 2.0 by adding: the ability to share a primary seed with the cloud, and the ability to access cloud-hosted non-volatile (NV) storage. This section describes the high-level design and the challenges we encountered when implementing these features. While our description is TPM 2.0-specific, our changes could be equally applied to TPM 1.2.

5.1 Cross-Device Usage Model

Each device has a unique cTPM with a unique primary seed shared with the cloud and used to derive additional keys (Section 5.3 describes the derived keys in more depth). All devices registered with the same owner have their keys tied to the owner's credentials. The cloud could then offer cTPM services that create a shared key across all devices owned by the same user. For example, when "bob@hotmail.com" calls this service, a shared key is automatically provisioned to the cTPM on each of Bob's devices. This shared key can bootstrap the data sharing scenarios described by this paper.

5.2 Architecture

cTPM consists of two different components, one running on the device and the other in the cloud. Both components implement the full TPM 2.0 software stack with the additional cTPM features. This ensures that all cloud

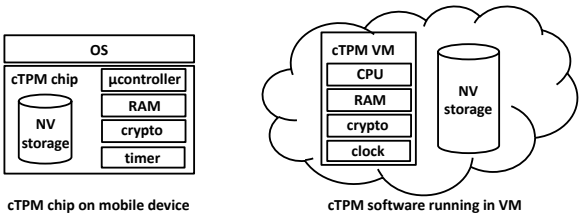


Figure 3. **cTPM High-level Architecture**.

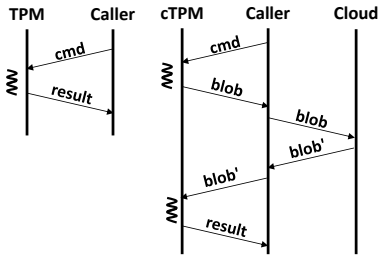


Figure 4. **The sequence of steps for issuing a synchronous command (left) versus an asynchronous command (right).** The cTPM remains responsive to other commands while the caller relays the blob to the cloud.

operations made to the cTPM strictly follow TPM semantics, and thus we do not need to re-verify their security properties. On the device-side, the cTPM software stack runs in the TPM chip, whereas the cloud runs the cTPM software inside a VM. On the cloud-side, the NV storage is regular cloud storage, and the timer offers a real-time clock function. The cloud-side cTPM software reads the local time upon every initialization and uses NTP to synchronize with a reference clock. When running in the cloud, cTPM resources (e.g., storage, clock) need not be encapsulated in hardware because the OS running in the VM is assumed to be trusted. In contrast, the device's OS is untrusted, and thus the cTPM chip itself must be able to offer these resources in isolation from the OS. Figure 3 illustrates the high-level architecture of the cTPM.

5.3 Shared Cloud Primary Seed

Upon starting, the local cTPM checks whether a shared cloud primary seed is present. If not, it disables its cloud control domain and all commands associated with it. A cTPM is provisioned with a cloud primary seed via a proprietary interface available only to the device manufacturer.

The cTPM uses the cloud primary seed to generate an asymmetric storage root key, called the *cloud root key* (CRK), and a symmetric communication key, called the *cloud communication key* (CCK). Both keys are derived from the cloud primary seed. These key derivations occur twice: once on the device-side and once on the cloud-side of the cTPM. Because the key derivations are deterministic, both the device and the cloud end up with identical key copies. The CRK's semantics are identical to those of the *storage root key* (SRK) controlled by the TPM's owner domain. The CRK encrypts all objects protected within the cloud control domain (similar to how SRK encrypts all objects within the owner domain). The CCK is specific to the cloud domain, and it protects all data exchanged with the cloud.

The cTPM uses the same mechanism to generate keys as TPM 2.0. In particular, the generation of a primary key from a seed is based on use of an approved key derivation function (KDF). TPM 2.0 uses the KDF from SP800-108 [25] in its specification.

We now examine the design challenges associated with exchanging data between the local cTPM and the cloud cTPM.

The Need for Secure Asynchronous Communication. cTPM cannot directly communicate with the cloud. Instead, it must rely on the OS for all its communication needs. Since the OS is untrusted, cTPM must protect the integrity and confidentiality of all data exchanged between the cTPM and the cloud-backed storage, as well as protect against rollback attacks. The OS is regarded merely as an insecure channel that forwards information to and from the cloud.

In addition to ensuring security, cTPM must support asynchronous communication between the local cTPM and the cloud. Today, the TPM is single-threaded, and all TPM commands are synchronous. When a command arrives, the caller blocks and the TPM cannot process any other commands until the command terminates. Making cTPM cloud communication synchronous would lead to unacceptable performance. For example, consider issuing a cTPM command that increments a counter in cloud-backed NV storage. This command would make the TPM unresponsive and block until the increment update propagates all the way to the cloud and the response returns to the local device.

Instead, we chose to make cloud communication asynchronous. Whenever a command that needs access to remote NV is received, cTPM returns to the caller an encrypted blob that needs to be sent remotely. The caller must send this blob to the cloud; if the cloud accepts the blob, it returns another encrypted blob reply to the caller. The caller then passes this reply to the cTPM, at which point the command completes. cTPM remains responsive to all other commands during this asynchronous communication with the cloud. Figure 4 illustrates these steps and contrasts them with a traditional simple TPM command. All cTPM commands that do not require access to remote NV storage remain synchronous, similar to TPMs today.

Dealing with Connectivity Loss. Loss of connectivity is transparent to the cTPM because all network signaling and communication is done by the operating system. However, the two-step nature of asynchronous commands requires the cTPM to maintain in-memory state between the steps. This introduces another potential resource allocation denial-of-service attack: a malicious OS could issue many asynchronous commands that cause the cTPM to fill up its RAM. Also, as mentioned in our threat model, an attacker could launch a staleness attack whereby artificial delays are introduced in the communication with the cloud.

To protect against these attacks, cTPM maintains a *global route timeout* (GRT) value. Whenever an asynchronous request is issued, cTPM starts a timer set to the GRT. Additionally, to free up RAM, cTPM scans all outstanding asynchronous commands and discards those whose timers have expired. The GRT can be set by the cTPM's owner and has a default value of 5 minutes.

5.4 Cloud-backed NV Storage

The TPM uses a special data structure, called an NV index, to store data values persistently to NV storage. When a persistent object is referenced in a TPM command, the TPM loads the object into its RAM. When allocating a new index, an application must specify its access control (read-only or read-write), its type, and size. There are four possible types of NV indexes: (1) *ordinary*, for storing data blobs, (2) *counters*, for storing secure monotonic counters, (3) *bit-fields*, which can be set individually, and (4) *extend*, which can be modified only by using an extend operation similar to PCRs.

At a high level, the cloud-backed NV storage is just a key-value store whose keys are NV indices. Accessing the remote NV index entries requires the OS to assist with the communication between the cTPM and the cloud. These operations are thus asynchronous and follow the same two-step model described in Figure 4. However, the remote nature of these NV indices raises additional design challenges.

Local NV Storage Cache. Remote NV entries can be cached locally in the cTPM's RAM. To do so, we add a *time-to-live* (TTL) to remote NV entries. The TTL specifies how long (in seconds) the cTPM can cache an NV entry in its local RAM. Once the TTL expires, the NV index is deleted from RAM and must be re-loaded from the remote cloud NV storage with a fresh, up-to-date copy. The local storage cache is *not persistent* – it is fully erased each time the computer reboots. We also add a *synchronization timestamp* (ST) set to the time the entry was last cached locally. If there is no in-memory cached entry of the NV index, this timestamp is null.

Caching's main benefits are performance and availability; remote NV read operations may not require a round-trip to the cloud if they can be read from the local cache. This enables the reading of NV storage entries even when the device is disconnected as long as their TTL has not expired. The trade-off is that locally cached entries could be stale. Cloud updates to a cloud-backed NV entry are reflected locally only after the TTL expires. The TTL controls the trade-off between performance and staleness for each NV index entry.

For writes, the local cache's policy is write back, and it relies on the caller to propagate the write to the cloud NV storage. A cTPM NV write command updates the cache first and returns an error code that indicates the write back to the NV storage is pending. The caller must initiate a write protocol to the cloud NV. If the caller fails to complete the write back, the write remains volatile, and the cTPM makes no guarantees about its persistence.

Trusted Clock. In cTPM, the trusted clock is an NV entry (with a pre-assigned NV index) that only the cloud can update. The local device can read the trusted clock simply by issuing an NV read command for this remote entry. Reading the entry is subject to a timeout much stricter than the regular global route timeout (GRT), called the *global clock timeout* (GCT). The trusted clock NV entry is cached in the on-chip RAM. In this way, the cTPM always has access to the current time by adding the current timer tick count to the synchronization timestamp (ST) of the clock NV entry.

$$maxClockError \leq TTL \times drift + GCT \quad (1)$$

Equation (1) describes the upper bound of the local clock's accuracy as a function of TTL, drift and GCT. By default, the TTL is set to 1 day and the global clock timeout (GCT) to 1 second. A low GCT improves local clock accuracy, but may lead to unavailability if the device-to-cloud communication has high latency. We find that these values are sufficiently accurate for our mobile scenario (i.e., the release of movies on Fridays at midnight). However, setting the GCT even lower can further improve accuracy, while setting the TTL lower reduces the effect of drift.

5.5 Islanded Devices

Although connectivity loss is masked by the OS, devices could be offline for long periods of time. We refer to such devices as islanded devices. Islanded devices do not raise additional security concerns, even when they are out of sync with the cloud. Instead, when long periods of disconnection occur the cTPM functionality slowly degrades as entries in the local NV cache become stale. When devices reconnect, they need to re-sync their cloud-based cTPM state. However, we believe that most

```
NV_Read(NVIndex idx) {

// Garbage collect all local cache
foreach nvIdx in LocalCache
  if LocalCache[nvIdx].TTL Is Expired
    delete nvIdx from LocalCache
  endif
endforeach

// return NV entry if present
if idx in LocalCache return LocalCache[idx]

// return not found in cache
return ErrorCode.NotFoundInCache
}
```

Figure 5. **Reading NV entry from local cache.**

```
NV_Write(NVEntry entry) {

//Garbage collect all local cache
foreach nvIdx in LocalCache
  if LocalCache[nvIdx].TTL Is Expired
    delete nvIdx from LocalCache
  endif
endforeach

//Insert the entry in the cache
idx = LocalCache.Append(entry)

//Set the entry's TTL
LocalCache[idx].TTL = DefaultTTL
}
```

Figure 6. **Writing NV entry to local cache.**

mobile devices become islanded only when left unused. When used regularly, devices have ample opportunity to connect to the Internet and sync their cTPM state.

6 Detailed Design and Implementation

This section provides more detail on the cTPM's design and implementation. We describe how the cTPM shares TPM-protected keys between the cloud and the device, and we present the changes made to support NV reads and writes. We also describe the cloud/device synchronization protocol, and the three new TPM commands we added to implement synchronization.

6.1 Sharing TPM-protected Keys

The TPM 2.0 API facilitates the sharing of TPM-protected keys by decoupling key creation from key usage. **TPM2_Create()**, a TPM 2.0 command, creates a symmetric key or asymmetric key-pair. The TPM creates the key internally and encrypts any private (or symmetric) keys with its storage key before returning them to the caller. To use the key, the caller must issue a **TPM2_Load()** command, which passes in the public storage key and the encrypted private (or symmetric) key. The TPM decrypts the private key and loads it in RAM. The TPM can then begin to encrypt or decrypt using the key.

The separation between create and load is needed due to the limited RAM available on the TPM chip. It lets callers create many keys without having to load them all into RAM. As long as the storage root key (SRK) never leaves the chip, encrypting the new keys' private parts with the SRK guarantees their confidentiality.

This separation lets cTPM use cloud-created keys on the local device to gain two benefits. First, key sharing between devices becomes trivial. The cloud can perform the key sharing protocol between two cTPM VMs, as described earlier in Figure 2. Unlike TPM 2.0, this protocol does not need to use a PKI, nor does it need to run in a SEM. Once a shared key is created between two cloud cTPM VMs, both mobile devices can load the key in their chips separately by issuing **TPM2_Load()** commands. Second, key creation can be performed even when the mobile device is offline. This makes it simple for users to create shared keys across all their devices without having to ensure those devices are online first. We illustrate both these benefits in our extension of Pasture in Section 7.

6.2 Accessing Cloud NV Storage

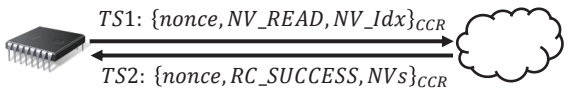
The cTPM maintains a local cache of all reads and writes made to the cloud NV storage. A read returns a cache entry, and a write updates a cache entry only. The cTPM does not itself update remote cloud NV storage; instead the caller must synchronize the on-chip RAM cache with the cloud NV storage. This is done using a synchronization protocol.

Read Cloud NV. Upon an NV read command, the corresponding NV entry is returned from the local cache. If not found, cTPM returns an error code. The caller must now check the remote NV; to do so, it needs to initiate a *pull* synchronization operation (described in Section 6.3) to update the local cache. After synchronization completes, the caller must reissue the read TPM command, which will now be answered successfully from the cache. Figure 5 shows the pseudo-code for reading a remote NV entry from the local cache.

Write Cloud NV. An NV write command first updates the cache and returns an error code that indicates the write back to the remote NV storage is pending. The caller must initiate a *push* synchronization operation to the cloud NV (see Section 6.3). If the caller fails to complete the write back, the write remains volatile, and cTPM makes no guarantees about its persistence. Figure 6 shows the pseudo-code for writing an NV entry to the local cache.

6.3 Synchronization Protocol

The synchronization protocol serves to: (1) update the local cache with entries from the cloud-backed NV storage for NV reads) and (2) write updated cache entries back to the cloud-backed NV storage (for NV writes). On the device side, the caller performs the protocol using two new commands, **TPM2_Sync_Begin()** and



If $TS_2 - TS_1 >$ GRT , read is not fresh.

Figure 7. **Synchronization protocol: pull NV entry from cloud-backed NV storage.**

TPM2_Sync_End(). These commands take a parameter called *direction*, which can be set to either a *pull* or *push* to distinguish between reads and writes. All messages are encrypted with the cloud communication key (CCK), a symmetric key.

Pull from Cloud-backed NV Storage. The cTPM first records the value of its internal timer and sends a message that includes the requested NV index and a nonce. The nonce checks for freshness of the response and protect against replay attacks. Upon receipt, the cloud decrypts the message and checks its integrity. In response, the cloud sends back the nonce together with the value corresponding to the NV index requested. The cTPM decrypts the message, checks its integrity, and verifies the nonce. If these checks are successful, cTPM performs one last check to verify that the response's delay did not exceed its global read timeout (GRT) value. If all checks pass, cTPM processes the read successfully. Figure 7 shows the precise messages exchanged between the cTPM and the cloud to read the remote NV.

Push to Cloud-backed NV Storage. The protocol for writing back an NV entry is more complex because it must also handle the possibility that an attacker may try to reorder write operations. For example, a malicious OS or application can save an older write and attempt to reapply it later, effectively overwriting the up-to-date value. To overcome this, the protocol relies on a secure monotonic counter maintained by the cloud. Each write operation must present the current value of the counter to be applied; thus, stale writes cannot be replayed. cTPM can read the current value of the secure counter using the previously described pull protocol. Figure 8 shows the precise messages exchanged between the cTPM and the cloud to write a remote NV entry. Note that reading the secure counter need not be done on each write because the local cTPM caches the up-to-date value in RAM.

When the cloud receives an NV entry through the push synchronization protocol, it must update its NV storage. To do so, we equipped the cTPM with a third command, called **TPM2_Sync_Proc()** (for *process*). This command can be issued only by the cloud; the cloud takes the message received from the local device and calls sync process with it. The cloud cTPM decrypts the message and applies the NV update.

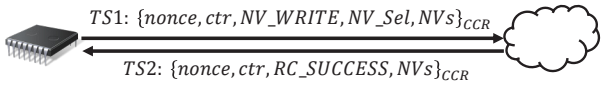


Figure 8. **Synchronization protocol: push NV entry to cloud-backed NV storage.**

6.4 Implementation

We implemented cTPM by modifying the TPM 2.0 (release 0.96) codebase; this codebase serves as both the TPM specification and a reference implementation. The original codebase was 23,163 lines of code; for cTPM, we added 1,304 lines of code, for a total of 24,467 lines of code. The bulk of this code implements the three new cTPM commands – sync begin, end, and process. We also made minor changes to the commands that update the TPM NV. These commands indicate whether NV access should be to the local on-chip NV or the cloud-backed NV. If the latter, the command can return additional error codes depending on whether the NV entry is found in the local cache (for reads) or whether the update must be written back to the cloud NV.

The TPM 2.0 codebase does not include a cryptography library. This is deliberate in order to reduce the hurdle of porting it to different OEM hardware environments. For example, one hardware OEM might want to use its own in-house crypto library, whereas another might want to use OpenSSL. The TPM 2.0 codebase just defines a crypto API. We used a Microsoft internal cryptography library for the TPM 2.0 needs.

TPM 2.0 also does not include platform resources, such as how to obtain entropy, how to receive a power-on/power-off signal, or how to access the underlying NV storage. For all platform needs, we used a library that provides the TPM platform resources from the underlying OS (Windows). Our platform implementation receives TPM commands via a network socket using a home-baked command/response protocol.

All our testing code and applications, such as Pasture and TrInc, were implemented in C#. All TPM commands are relayed via the network socket to and from cTPM.

7 Case Studies

This section presents two case studies on using cTPM to build trusted mobile services. In each case, we describe these services' current limitations and show how cTPM addresses them by improving performance or adding functionality.

7.1 Case Study 1: Pasture

Pasture [14] is a TPM-based protocol for secure offline data access. Using Pasture, the content receiver creates a TPM-bound encryption key, called a *bound key*,

with a usage policy dictating that the TPM can use the key for decrypting content only when a certain PCR register contains a specified value. This policy provides *access undeniability* – once the key is used for decryption, the user cannot lie about its usage, and *verifiable revocation* – the user cannot use the key once revoked. Issuing a TPM extend operation on the PCR register to a predetermined value R represents the receiver's decision to consume the content. If the receiver decides to revoke the content instead, it extends the PCR to a different value. In this case, the PCR cannot be extended to R any longer. Over time, the PCR value represents a chain of decisions about whether to watch or revoke a sequence of movies.

Upon receiving a bound key, the content server checks that the key is bound to a correct usage policy, encrypts the content using it, and sends the encrypted content to the receiver. At any point in the future, the receiver can choose whether or not to decrypt the content; this choice can occur even in disconnected mode. Once made however, this choice cannot be undone.

The Pasture protocol also addresses computer reboots. An adversary could try to use such reboots to reset the PCR register, which opens the door to rollback attacks. This part of the Pasture protocol requires secure execution mode (SEM). In our cross-device Pasture implementation, we eliminated the need for SEM by leveraging a new TPM 2.0 type of NV index that has behavior similar to a PCR and is modified using *TPM2_NV_Extend()*.

Limitation 1: Lack of Sharing. The lack of sharing primitives in TPM prevents extending Pasture to a set of devices owned by a single user. Instead, each device must run its own version of Pasture, creating a TPM-specific bound key and uploading it to the server. The server then runs a Pasture session with each individual device. All devices must act separately despite being owned by the same user.

With cTPM, the cloud performs the Pasture protocol on behalf of all devices owned by a single user. A single bound key is shared by all devices, and a single copy of the content is encrypted with this bound key. The cloud can write this bound key directly to the cloud-backed NV storage and encrypt the content even when the client is disconnected. When the receiver connects to the cloud, it can then re-sync its nonvolatile state to receive a copy of the bound key and to start downloading encrypted content. As with the original Pasture, the policy specifies the PCR value necessary to use the key bound to it.

This multi-device version of Pasture complicates the process of accepting and revoking content. If any device accepts content and starts decrypting it, then the content can no longer be revoked. Thus, the content server accepts a revoke decision only when *all* of a user's devices have decided to revoke.

```
CreateBoundKey(hM):

//For each device, read current PCR and
// future PCR if decision is accept.
foreach dev in Owner.AllDevices() do
  R_t^dev ← TPM_Read(PCR_APP^dev)
  R_{t+1}^dev ← SHA2(R_t^dev||hM)
endfor

//Create bound key with a disjunction of commit values.
E←TPM_CreateWrapKey({
  {
    PCR_APP = R_{t+1}^dev1 ||
    PCR_APP = R_{t+1}^dev2 ||
    ...
    PCR_APP = R_{t+1}^devN
  }
  &&
  PCR_SEM = SemHappy
})

//Create proof for the bound key.
EP← ( ``CreateBoundKey'', hM,
  R_t^dev1, R_{t+1}^dev1, ..., R_t^devN, R_{t+1}^devN, E, α)
```

Figure 9. **Create bound key in multi-device Pasture.**

Another interesting challenge of multi-device Pasture is when one user makes conflicting decisions on different devices. This causes different values to be stored in their PCR registers. One solution is to insist that all devices owned by the same user share the same log of decisions about accepting or revoking the content. This ensures that the PCR registers on each device share the same value and work in sync. However, it is very difficult to enforce this coordination across devices when some are offline.

An alternate approach lets different devices maintain their own per-device log of decisions. This more flexible solution lets a user make different decisions for different devices without having to reconcile them. Because the per-device decisions can differ, the content server must ensure that a content revocation occurs only when all devices revoke. This approach requires the bound key to be attached to a policy that specifies a set of possible PCR values corresponding to each separate device.

We implemented this latter approach using the *TPM2_PolicyOR()* command, which creates a single policy as a disjunction of individual conditions (in our case, each condition corresponds to one PCR value). As long as a device's PCR value matches one condition, the bound key can decrypt the content. Note that extending the accept decision from one value to multiple values does not reduce protocol security even though it increases the chances of a hash collision. If hash collisions ever become a cause of concern, TPM 2.0 (and thus cTPM) permits the use of stronger hash functions (e.g., 192-bit SHA384). Figure 9 shows the multi-device CreateBoundKey implementation (CreateBoundKey in the original Pasture is shown in Figure 3 of [14]).

```
Attest(i, c', h, n):

1. Assert NV_Read(i) is a remote NV of type counter.
2. Let c be the value of that counter.
3. Assert no roll-over: c ≤ c'.
4. a ←< i, c, c', h >TRINC_PRIV.
5. Insert a into Q, ejecting the oldest value.
6. NV_Write(i, c').
7. Return a.
```

Figure 10. **Attest in cTPM TrInc [16].**

Limitation 2: Lack of Server-side Revocation. The original Pasture protocol lets a receiver revoke access to content in a verifiable manner. Once revoked, the receiver cannot further access the content. However, Pasture does not support server-side revocation. A Pasture movie server could use revocation to deny access to malicious clients, such as clients that paid using stolen credit cards.

Implementing server-side revocation in Pasture would prove very challenging because the client would have to agree to run code that would unload the bound key from the TPM. The client could always refuse to run such code and prevent a server from revoking the bound key.

With cTPM, the content server could simply ask the cloud to delete the bound key from the cloud-backed cTPM NV storage. This would not necessarily cause an immediate revocation because the device could be offline and store a cached copy of the key. However, the cached copy would eventually expire (based on its TTL), at which point the key is guaranteed to be revoked.

Limitation 3: Lack of Trusted Clock Guarantees. Because TPMs lack a trusted source of time, a Pasture movie server cannot offer time-based guarantees for its content (e.g., *make the following movie available for watching next Friday at midnight*). This scenario is quite attractive to consumers: a startup is currently selling proprietary technology for watching movies at home the same day they arrive in theaters. This hardware costs $35,000, and each movie release costs $500 [6].

With cTPM's trusted clock, a bound key could incorporate a clause specifying a future timestamp as an additional condition for using the key for decryption. The bound key could be revoked (either client-side or server-side) at any time; however its usage for decryption would remain restricted to only a future, pre-specified time value.

7.2 Case Study 2: TrInc

TrInc [16] is a trusted incrementer used to combat "equivocation", i.e., making conflicting statements to others in a distributed system. It uses a secure counter and a key, and was implemented on a smartcard due to its poor performance on TPMs.

TrInc's main API, a function called *Attest*, produces an attestation that the secure counter has been incremented from the current value c to a value c' not smaller than c. Each attestation covers the secure counter's interval $(c, c']$. TrInc uses these attestations to prove statements that prevent nodes from equivocating without being detected. In BitTorrent, for instance, the counter represents the number of blocks a peer has received, a value which is naturally monotonically increasing. Figure 10 illustrates our implementation of the Attest function using cTPM (the original Attest implementation is described in Section 3.5.1 of [16]).

TPM 2.0	TPM 1.2
TPM2_NV_Write()	TPM_NV_Write()
TPM2_NV_Read()	TPM_NV_Read()
TPM2_NV_Read (Counter)()	TPM_ReadCounter()
TPM2_PCR_Read()	TPM_PCRRead()
TPM2_PCR_Extend()	TPM_PCRWrite()
TPM2_Create()	TPM_Create()
TPM2_Load()	TPM_Load()
TPM2_Unseal(Unbind)()	TPM_Unbind()
TPM2_Sign()	TPM_Sign()
TPM2_Quote()	TPM_Quote()
TPM2_CertifyCreation()	TPM_Sign()
TPM2_Sync_Begin()	TPM_Unbind()
TPM2_Sync_End()	TPM_Unbind()

Table 1. **Mapping TPM 2.0 commands to their TPM 1.2 counterparts.**

8 Evaluation

8.1 Protocol Verification

We verified the correctness of our protocols using an automated theorem prover, ProVerif [3], which supports the specification of security protocols for distributed systems in concurrent process calculus (pi-calculus). We specified the synchronization protocol used by our system, both pull and push, in 98 lines of pi-calculus code. ProVerif verified the security of our protocols in the presence of an attacker with unrestricted access to the OS, applications, or network. The attacker could intercept, modify, replay and inject new messages into the network (similar to the Dolev-Yao model).

8.2 Performance Evaluation

Our main challenge in evaluating the performance of cTPM was the unavailability of a hardware TPM 2.0 chip. The TPM 2.0 specification, currently released for public review, is not yet available off-the-shelf. Through private conversations with TPM manufacturers, we learned that they are already porting the TPM 2.0 specification to their hardware, and that the hardware performance profile for TPM 2.0 will be similar to that of TPM 1.2. As a result, we used a TPM 1.2 chip to emulate the hardware performance of a future TPM 2.0 chip. To do so, we mapped TPM 2.0 commands used in our cTPM implementation to their equivalent TPM 1.2 counterparts, as shown in Table 1.

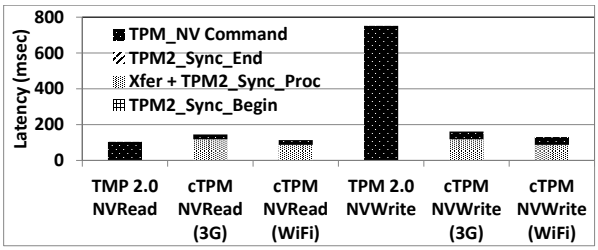


Figure 11. **cTPM NV performance:** 640 bytes.

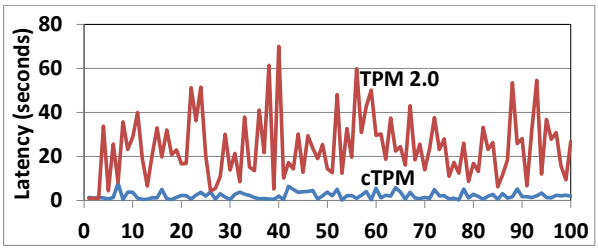


Figure 12. **Latency of TPM RSA 2048 key creation**.

Our experiments used the following setup. All applications that ran on the local cTPM used Windows 7 running on a PC with a 2 GHz Quad Intel Core i7. When a local cTPM command was issued, it was translated according to the map in Table 1 and executed against an Infineon TPM 1.2 chip. In the cloud, we ran the cTPM software in a Windows 7 VM (to emulate cloud behavior). By design, the cloud component of cTPM did not need to interact with a TPM chip.

Benchmarking NV Storage Performance. cTPM trades off the accessing of limited local TPM NV storage for the accessing of cloud-backed storage. While cloud-backed storage is very fast, it introduces latency between the device and the cloud. To evaluate this trade-off, we measured the latency of NV read and write operations for both TPM 2.0 and cTPM. We emulated Internet latencies using a standard network emulator [22] primed with 3G/4G and Wi-Fi Internet latency distributions from a recent measurement project [13].

We repeated our experiments with differently sized objects accessed in NV storage; sizes ranged from 256 bytes (corresponding to the size of a regular NV counter) to the maximum size allowed by the hardware TPM. Unfortunately, TPMs have low NV storage capacities: the largest write allowed by our TPM was only 640 bytes (whereas cTPM had no restrictions on the maximum size of its NV data). We present results using only 640-byte data objects; the results for the lower-sized objects are similar.

Figure 11 shows the access latencies for 640-byte NV objects. The local TPM 2.0 latencies are all due to the running of TPM NV commands. In contrast, cTPM latencies are the combination of four steps: (1) issuing a TPM2_Sync_Begin() command, (2) transferring the data to and from the cloud (labeled *Xfer*) and issuing a TPM2_Sync_Proc() command in the cloud, (3) issuing a TPM2_Sync_End() command, and (4) issuing a TPM NV command to access the data in memory. For NV reads, Internet latencies make the cTPM commands slightly slower in the case of 3G latencies and slightly faster in the case of Wi-Fi latencies. Note, however, that NV reads become much faster once cached locally.

Benchmarking Key Creation. When creating a key, the TPM uses local on-chip entropy, whereas cTPM can use any entropy source available to the cloud, such as a high performance hardware entropy source. For example, a company in Japan sells hardware able to produce 550 MBytes/sec of entropy for less than US$10K [15]. Our experiments used local hardware entropy found on commodity PCs. Even this entropy source was much faster than that found in TPMs.

Figure 12 shows the latencies of running 100 consecutive TPM commands to create an RSA key. The latencies of TPM commands are highly variable because the TPM blocks incoming commands to wait for the entropy source to generate random bits. The same is true for the hardware source of entropy on our PC, but this source is much faster, i.e., an average of an order of magnitude (factor of 12) faster than the TPM chip.

9 Related Work

cTPM draws inspiration from previous work on commodity trusted hardware and trusted applications.

Commodity Trusted Hardware Other than TPMs. The ARM architecture's solution for trusted computing is known as the ARM TrustZone [1]. ARM TrustZone provides a trusted execution environment on CPU cores, with hardware support for memory protection of the trusted environment, flexible control over interrupt delivery to the trusted environment, and the full power of the CPU for cryptographic operations. One could equip an ARM device with a TPM (or a cTPM) by running the TPM software stack inside the TrustZone.

Recent work from Intel has described Secure Guard Extensions (SGX) [20, 2, 12], a set of new instructions and architectures that support the concept of *enclaves*, which are isolated runtime environments similar to ARM TrustZone. Intel has shown the possibility of running secure applications inside of an enclave, such as a password manager, an enterprise rights management solution, and secure video conferencing [12]. It appears feasible for the TPM and cTPM software stacks to run inside an enclave, as well.

Trusted Applications. In addition to Pasture [14] and TrInc [16], several previous works have proposed the use

of TPMs for building trusted mobile services. TruWallet described a TPM-based authentication tool for Web password protection [7]. It offered password sharing across devices owned by the same user (called *secure migration*). However, TruWallet needed to assume that the GUI and kernel were both trusted. Another project implemented a credentials manager in secure execution mode [4]. It encountered many of the performance challenges associated with this mode; for example, the network driver froze when running in SEM for more than 8 seconds. More recently, Windows 8 provided virtual smart cards, a way to use TPMs for remote authentication with server-side support [23]; these cards, bound to a single TPM, cannot migrate. For all these applications, cTPM would greatly ease sharing across devices.

Memoir describes a technique to protect the state of a trusted application while minimizing the number of NVRAM write operations [26]. With cTPM, applications could write their state to the cloud-backed NV storage and rely on Memoir-like techniques only when operating in disconnected mode.

10 Conclusions

This paper introduced cTPM, a cloud-enhanced design change to a traditional TPM that enables the simple sharing of keys and data across a user's many devices. We demonstrated cTPM's versatility by: 1) extending Pasture to support offline data access across multiple devices, server-side revocation, and real-time based guarantees for content availability, and 2) re-implementing TrInc without the need for extra hardware. We verified the protocols used to synchronize the cTPM's remote cloud storage and showed that cTPM's performance meets or exceeds that of a traditional TPM.

Acknowledgments: We are grateful to Ron Aigner, Ramakrishna Kotla, Jay Lorch, Bryan Parno, and Scott Shell for their feedback on this work and on the paper. We would like to thank Jonathan Smith and the anonymous reviewers for their feedback.

References

[1] ARM Security Technology – Building a Secure System using TrustZone Technology. ARM Technical White Paper, 2005-2009.

[2] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proc. of Workshop on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel, 2013.

[3] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. of 14th IEEE Computer Security Foundations Workshop (CSFW)*, Cape Breton, NS, 2001.

[4] S. Bugiel and J.-E. Ekberg. Implementing an Application-Specific Credential Platform Using Late-Launched Mobile Trusted Module. In *Proc. of 5th Annual Workshop on Scalable Trusted Computing*, Chicago, IL, 2010.

[5] D. Challener, K. Yoder, R. Catherman, D. Safford, , and L. V. Doorn. *A Practical Guide to Trusted Computing*. IBM Press, 2007.

[6] Digital Trends. Prima Cinema brings movies to home theaters on the day of the release for $500 a pop. http://www.digitaltrends.com/home-theater/prima-cinema-brings-movies-to-the-home-on-the-day-of-the-release/.

[7] S. Gajek, H. Lohr, and A.-R. Sadeghi. TruWallet: Trustworthy and Migratable Wallet-Based Web Authentication. In *Proc. of 4th Annual Workshop on Scalable Trusted Computing*, Chicago, IL, 2009.

[8] GigaOM. The average US subscriber owns 1.57 mobile devices. http://gigaom.com/2012/10/22/the-average-us-subscriber-owns-1-57-mobile-devices/.

[9] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharkey, A. Sheth, and L. P. Cox. YouProve: Authenticity and Fideltiy in Mobile Sensing. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Lake District, UK, 2012.

[10] K. A. Goldman. IBM's Software Trusted Platform Module. http://sourceforge.net/projects/ibmswtpm/.

[11] Google. The Chromium Projects. http://www.chromium.org/developers/design-documents/tpm-usage.

[12] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proc. of Workshop on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel, 2013.

[13] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance Power Characteristics of 4G LTE Networks. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Lake District, UK, 2012.

[14] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester. Pasture: Secure Offline Data Access Using Commodity Trusted Hardware. In *Proc. of 10th USENIX Symposium on Operating Systems*

Design and Implementation (OSDI), Hollywoood, CA, 2012.

[15] LETech. The Fastest True Random Number Generator with a real-time self-test function. http://www.letech.jpn.com/rng/grang_24ch_e.html.

[16] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proc. of 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, 2009.

[17] H. Liu, S. Saroiu, A. Wolman, and H. Raj. Software Abstractions for Trusted Sensors. In *Proc. of 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Lake District, UK, 2012.

[18] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2010.

[19] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, Glasgow, UK, 2008.

[20] F. Mckeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proc. of Workshop on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel, 2013.

[21] Microsoft. Help protect your files with BitLocker Driver Encryption. http://windows.microsoft.com/en-us/windows-8/using-bitlocker-drive-encryption.

[22] Microsoft. Standalone Network Emulator Tool. http://blogs.technet.com/b/juanand/archive/2010/03/05/standalone-network-emulator-tool.aspx.

[23] Microsoft. Understanding and Evaluating Virtual Smart Cards. http://www.microsoft.com/en-us/download/details.aspx?id=29076.

[24] A. Nguyen, H. Raj, S. Rayanchu, S. Saroiu, and A. Wolman. Delusional Boot: Securing Cloud Hypervisors without Massive Re-engineering. In *Proc. of the European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, April 2012.

[25] NIST. Recommendation for Key Derivation Using Pseudorandom Functions. http://csrc.nist.gov/publications/nistpubs/800-108/sp800-108.pdf.

[26] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical State Continuity for Protected Modules. In *Proc. of IEEE Symposium on Security and Privacy*, Oakland, CA, 2011.

[27] J. Postel and K. Harrenstien. Time Protocol. http://tools.ietf.org/html/rfc868.

[28] H. Raj, D. Robinson, T. Tariq, P. England, S. Saroiu, and A. Wolman. Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor. Technical Report MSR-TR-2011-130, Microsoft Research, 2011.

[29] M. Ryan. Introduction to the TPM 1.2. www.cs.bham.ac.uk/~mdr/research/papers/pdf/08-intro-TPM.pdf.

[30] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proc. of 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, Phoenix, AZ, 2011.

[31] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proc. of the 21st USENIX Security Symposium*, Bellevue, WA, 2012.

[32] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus Authorization Logic (NAL): Design Rationale and Applications. *ACM Transactions on Information and System Security*, 14(1), 2011.

[33] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, , and F. B. Schneider. Logical Attestation: An Authorization Architecture For Trustworthy Computing. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[34] S. Skorobogatov. Physical Attacks on Tamper Resistance: Progress and Lessons. In *Proc. of 2nd ARO Special Workshop on Hardware Assurance*, Washington, DC, 2011.

[35] C. Tarnovsky. Semiconductor Security Awareness, Today & Yesterday. BlackHat 2010 – http://www.youtube.com/watch?v=YzejlrGcnY8.

[36] The Economic Times (indiatimes). Singapore leads the world on mobile take up and marketing. http://articles.economictimes.indiatimes.com/2012-12-19/news/35912444_1_mobile-app-mobile-sales-singaporeans.

[37] The Register. Five mobile devices per person for 2040? http://www.theregister.co.uk/2012/07/18/acma_05_mobile_numbers/.

[38] Trusted Computing Group. TPM 2.0 Library Specification FAQ. http://www.trustedcomputinggroup.org/resources/tpm_20_library_specification_faq.

[39] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2, Revision 116. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[40] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proc. of Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[41] Y. Zhang, A. Juels, M. Reiter, and T. Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proc. of the 19th ACM Conference on Computer and Communications Security*, Raleigh, NC, 2012.

Network Virtualization in Multi-tenant Datacenters

Teemu Koponen*, Keith Amidon*, Peter Balland*, Martín Casado*, Anupam Chanda*, Bryan Fulton*, Igor Ganichev*, Jesse Gross*, Natasha Gude*, Paul Ingram*, Ethan Jackson*, Andrew Lambeth*, Romain Lenglet*, Shih-Hao Li*, Amar Padmanabhan*, Justin Pettit*, Ben Pfaff*, Rajiv Ramanathan*, Scott Shenker†, Alan Shieh*, Jeremy Stribling*, Pankaj Thakkar*, Dan Wendlandt*, Alexander Yip*, Ronghua Zhang*

*VMware, Inc. †UC Berkeley and ICSI

Operational Systems Track

ABSTRACT

Multi-tenant datacenters represent an extremely challenging networking environment. Tenants want the ability to migrate unmodified workloads from their enterprise networks to service provider datacenters, retaining the same networking configurations of their home network. The service providers must meet these needs without operator intervention while preserving their own operational flexibility and efficiency. Traditional networking approaches have failed to meet these tenant and provider requirements. Responding to this need, we present the design and implementation of a network virtualization solution for multi-tenant datacenters.

1 Introduction

Managing computational resources used to be a time-consuming task requiring the acquisition and configuration of physical machines. However, with server virtualization – that is, exposing the software abstraction of a server to users – provisioning can be done in the time it takes to load bytes from disk. In the past fifteen years server virtualization has become the dominant approach for managing computational infrastructures, with the number of virtual servers exceeding the number of physical servers globally [2, 18].

However, the promise of seamless management through server virtualization is only partially realized in practice. In most practical environments, deploying a new application or development environment requires an associated change in the network. This is for two reasons:

Topology: Different workloads require different network topologies and services. Traditional enterprise workloads using service discovery protocols often require flat L2, large analytics workloads require L3, and web services often require multiple tiers. Further, many applications depend on different L4-L7 services. Today, it is difficult for a single physical topology to support the configuration requirements of all of the workloads of an organization, and as a result, the organization must build multiple physical networks, each addressing a particular common topology.

Address space: Virtualized workloads today operate in the same address space as the physical network.[1] That is, the VMs get an IP from the subnet of the first L3 router to which they are attached. This creates a number of problems:

- Operators cannot move VMs to arbitrary locations.
- Operators cannot allow VMs to run their own IP Address Management (IPAM) schemes. This is a common requirement in datacenters.
- Operators cannot change the addressing type. For example, if the physical network is IPv4, they cannot run IPv6 to the VMs.

Ideally, the networking layer would support similar properties as the compute layer, in which arbitrary network topologies and addressing architectures could be overlayed onto the same physical network. Whether hosting applications, developer environments, or actual tenants, this desire is often referred to as shared multi-tenancy; throughout the rest of this paper we refer to this as a *multi-tenant datacenter* (MTD).

Unfortunately, constructing an MTD is difficult because while computation is virtualized, the network is not. This may seem strange, because networking has long had a number of virtualization primitives such as VLAN (virtualized L2 domain), VRFs (virtualized L3 FIB), NAT (virtualized IP address space), and MPLS (virtualized path). However, these are traditionally configured on a box-by-box basis, with no single unifying abstraction that can be invoked in a more global manner. As a result, making the network changes needed to support server virtualization requires operators to configure many boxes individually, and update these configurations in response to changes or failures in the network. The result is excessive operator overhead and the constant risk of misconfiguration and error, which has led to painstaking change log systems used as best practice in most environments. It is our experience in numerous customer environments that while compute provisioning is generally on the order of minutes, network provisioning can take months. Our experience is commonly echoed in analyst reports [7, 29].

[1] This is true even with VMware VDS and Cisco Nexus 1k.

Academia (as discussed in Section 7) and industry have responded by introducing the notion of *network virtualization*. While we are not aware of a formal definition, the general consensus appears to be that a network virtualization layer allows for the creation of virtual networks, each with independent service models, topologies, and addressing architectures, over the same physical network. Further, the creation, configuration and management of these virtual networks is done through global abstractions rather than pieced together through box-by-box configuration.

And while the idea of network virtualization is not new, little has been written about how these systems are implemented and deployed in practice, and their impact on operations.

In this paper we present NVP, a network virtualization platform that has been deployed in dozens of production environments over the last few years and has hosted tens of thousands of virtual networks and virtual machines. The target environment for NVP is enterprise datacenters, rather than mega-datacenters in which virtualization is often done at a higher level, such as the application.

2 System Design

MTDs have a set of hosts connected by a physical network. Each host has multiple VMs supported by the host's hypervisor. Each host hypervisor has an internal software virtual switch that accepts packets from these local VMs and forwards them either to another local VM or over the physical network to another host hypervisor.

Just as the hypervisor on a host provides the right virtualization abstractions to VMs, we build our architecture around *a network hypervisor* that provides the right network virtualization abstractions. In this section we describe the network hypervisor and its abstractions.

2.1 Abstractions

A tenant interacts with a network in two ways: the tenant's VMs send packets and the tenant configures the network elements forwarding these packets. In configuring, tenants can access tenant- and element-specific control planes that take switch, routing, and security configurations similar to modern switches and routers, translating them into low-level packet forwarding instructions. A service provider's network consists of a physical forwarding infrastructure and the system that manages and extends this physical infrastructure, which is the focus of this paper.

The network hypervisor is a software layer interposed between the provider's physical forwarding infrastructure and the tenant control planes, as depicted in Figure 1. Its purpose is to provide the proper abstractions both to tenant's control planes and endpoints; we describe these abstractions below:

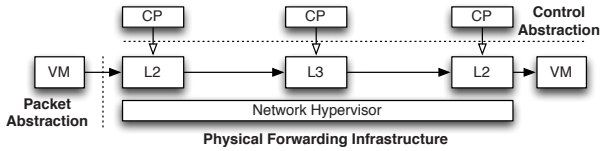


Figure 1: A network hypervisor sits on top of the service provider infrastructure and provides the tenant control planes with a control abstraction and VMs with a packet abstraction.

Control abstraction. This abstraction must allow tenants to define a set of logical network elements (or, as we will call them, logical datapaths) that they can configure (through their control planes) as they would physical network elements. While conceptually each tenant has its own control planes, the network hypervisor provides the control plane *implementations* for the defined logical network elements.[2] Each logical datapath is defined by a packet forwarding pipeline interface that, similar to modern forwarding ASICs, contains a sequence of lookup tables, each capable of matching over packet headers and metadata established by earlier pipeline stages. At each stage, packet headers can be modified or the packet can be dropped altogether. The pipeline results in a forwarding decision, which is saved to the packet's metadata, and the packet is then sent out the appropriate port. Since our logical datapaths are implemented in software virtual switches, we have more flexibility than ASIC implementations; datapaths need not hardcode the type or number of lookup tables and the lookup tables can match over arbitrary packet header fields.

Packet abstraction. This abstraction must enable packets sent by endpoints in the MTD to be given the same switching, routing and filtering service they would have in the tenant's home network. This can be accomplished within the packet forwarding pipeline model described above. For instance, the control plane might want to provide basic L2 forwarding semantics in the form of a *logical switch*, which connects some set of tenant VMs (each of which has its own MAC address and is represented by a *logical port* on the switch). To achieve this, the control plane could populate a single logical forwarding table with entries explicitly matching on destination MAC addresses and sending the matching packets to ports connected to the corresponding VMs. Alternatively, the control plane could install a special learning flow that forwards packets to ports where traffic from the destination MAC address was last received (which will time out in the absence of new traffic) and simply flood unknown packets. Similarly, it could broadcast destination addresses with a flow entry that sends packets to all logical ports (excluding the port on which the packet was received) on the logical switch.

[2] In other words, the network hypervisor does not run third-party control plane binaries but the functionality is part of the hypervisor itself. While running a third-party control plane stack would be feasible, we have had no use case for it yet.

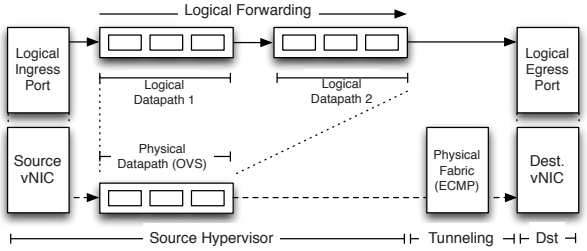


Figure 2: The virtual switch of the originating host hypervisor implements logical forwarding. After the packet has traversed the logical datapaths and their tables, the host tunnels it across the physical network to the receiving host hypervisor for delivery to the destination VM.

2.2 Virtualization Architecture

The network hypervisor supports these abstractions by implementing tenant-specific logical datapaths on top of the provider's physical forwarding infrastructure, and these logical datapaths provide the appropriate control and packet abstractions to each tenant.

In our NVP design, we implement the logical datapaths in the software virtual switches on each host, leveraging a set of tunnels between every pair of host-hypervisors (so the physical network sees nothing other than what appears to be ordinary IP traffic between the physical hosts). The logical datapath is almost entirely implemented on the virtual switch where the originating VM resides; after the logical datapath reaches a forwarding decision, the virtual switch tunnels it over the physical network to the receiving host hypervisor, which decapsulates the packet and sends it to the destination VM (see Figure 2). A centralized SDN controller cluster is responsible for configuring virtual switches with the appropriate logical forwarding rules as tenants show up in the network.[3]

While tunnels can efficiently implement logical point-to-point communication, additional support is needed for logical broadcast or multicast services. For packet replication, NVP constructs a simple multicast overlay using additional physical forwarding elements (x86-based hosts running virtual switching software) called *service nodes*. Once a logical forwarding decision results in the need for packet replication, the host tunnels the packet to a service node, which then replicates the packet to all host hypervisors that need to deliver a copy to their local VMs. For deployments not concerned about the broadcast traffic volume, NVP supports configurations without service nodes: the sending host-hypervisor sends a copy of the packet directly to each host hypervisor needing one.

In addition, some tenants want to interconnect their logical network with their existing physical one. This is done via *gateway* appliances (again, x86-based hosts running virtual switching software); all traffic from the physical network goes to the host hypervisor through this gateway appliance, and then can be controlled by NVP (and vice versa for the reverse direction). Gateway appliances can be either within the MTD or at the tenant's remote site. Figure 3 depicts the resulting arrangement of host hypervisors, service nodes and gateways, which we collectively refer to as *transport nodes*.

[3]NVP does not control physical switches, and thus does not control how traffic between hypervisors is routed. Instead, it is assumed the physical network provides uniform capacity across the servers, building on ECMP-based load-balancing.

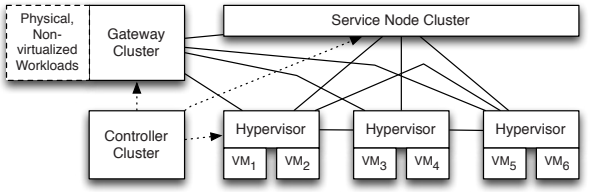


Figure 3: In NVP, controllers manage the forwarding state at all transport nodes (hypervisors, gateways, service nodes). Transport nodes are fully meshed over IP tunnels (solid lines). Gateways connect the logical networks with workloads on non-virtualized servers, and service nodes provide replication for logical multicast/broadcast.

2.3 Design Challenges

This brief overview of NVP hides many design challenges, three of which we focus on in this paper.

Datapath design and acceleration. NVP relies on software switching. In Section 3 we describe the datapath and the substantial modifications needed to support high-speed x86 encapsulation.

Declarative programming. The controller cluster is responsible for computing all forwarding state and then disseminating it to the virtual switches. To minimize the cost of recomputation, ensure consistency in the face of varying event orders, and promptly handle network changes, we developed a declarative domain-specific language for the controller that we discuss in Section 4.

Scaling the computation. In Section 5 we discuss the issues associated with scaling the controller cluster.

After we discuss these design issues, we evaluate the performance of NVP in Section 6, discuss related work in Section 7, and then conclude in Sections 8 and 9.

3 Virtualization Support at the Edge

The endpoints of the tunnels created and managed by NVP are in the virtual switches that run on host hypervisors, gateways and service nodes. We refer to this collection of virtual switches as the *network edge*. This section describes how NVP implements logical datapaths at the network edge, and how it achieves sufficient data plane performance on standard x86 hardware.

3.1 Implementing the Logical Datapath

NVP uses Open vSwitch (OVS) [32] in all transport nodes (host hypervisors, service nodes, and gateway nodes) to forward packets. OVS is remotely configurable by the

NVP controller cluster via two protocols: one that can inspect and modify a set of flow tables (analogous to flow tables in physical switches),[4] and one that allows the controller to create and manage overlay tunnels and to discover which VMs are hosted at a hypervisor [31].

The controller cluster uses these protocols to implement packet forwarding for logical datapaths. Each logical datapath consists of a series (*pipeline*) of logical flow tables, each with its own globally-unique identifier. The tables consist of a set of *flow entries* that specify expressions to match against the header of a packet, and actions to take on the packet when a given expression is satisfied. Possible actions include modifying a packet, dropping it, sending it to a given egress port on the logical datapath, and modifying in-memory metadata (analogous to registers on physical switches) associated with the packet and resubmitting it back to the datapath for further processing. A flow expression can match against this metadata, in addition to the packet's header. NVP writes the flow entries for each logical datapath to a single OVS flow table at each virtual switch that participates in the logical datapath. We emphasize that this model of a logical table pipeline (as opposed to a single table) is the key to allowing tenants to use existing forwarding policies with little or no change: with a table pipeline available to the control plane, tenants can be exposed to features and configuration models similar to ASIC-based switches and routers, and therefore the tenants can continue to use a familiar pipeline-based mental model.

Any packet entering OVS – either from a virtual network interface card (vNIC) attached to a VM, an overlay tunnel from a different transport node, or a physical network interface card (NIC) – must be sent through the logical pipeline corresponding to the logical datapath to which the packet belongs. For vNIC and NIC traffic, the service provider tells the controller cluster which ports on the transport node (vNICs or NICs) correspond to which logical datapath (see Section 5); for overlay traffic, the tunnel header of the incoming packet contains this information. Then, the virtual switch connects each packet to its logical pipeline by pre-computed flows that NVP writes into the OVS flow table, which match a packet based on its ingress port and add to the packet's metadata an identifier for the first logical flow table of the packet's logical datapath. As its action, this flow entry resubmits the packet back to the OVS flow table to begin its traversal of the logical pipeline.

The control plane abstraction NVP provides internally for programming the tables of the logical pipelines is largely the same as the interface to OVS's flow table and NVP writes logical flow entries directly to OVS, with two important differences:

- *Matches.* Before each logical flow entry is written to OVS, NVP augments it to include a match over the packet's metadata for the logical table's identifier. This enforces isolation from other logical datapaths and places the lookup entry at the proper stage of the logical pipeline. In addition to this forced match, the control plane can program entries that match over arbitrary logical packet headers, and can use priorities to implement longest-prefix matching as well as complex ACL rules.
- *Actions.* NVP modifies each logical action sequence of a flow entry to write the identifier of the next logical flow table to the packet's metadata and to resubmit the packet back to the OVS flow table. This creates the logical pipeline, and also prevents the logical control plane from creating a flow entry that forwards a packet to a different logical datapath.

At the end of the packet's traversal of the logical pipeline it is expected that a forwarding decision for that packet has been made: either drop the packet, or forward it to one or more logical egress ports. In the latter case, NVP uses a special action to save this forwarding decision in the packet's metadata. (Dropping translates to simply not resubmitting a packet to the next logical table.) After the logical pipeline, the packet is then matched against egress flow entries written by the controller cluster according to their logical destination. For packets destined for logical endpoints hosted on other hypervisors (or for physical networks not controlled by NVP), the action encapsulates the packet with a tunnel header that includes the logical forwarding decision, and outputs the packet to a tunnel port. This tunnel port leads to another hypervisor for unicast traffic to another VM, a service node in the case of broadcast and multicast traffic, or a gateway node for physical network destinations. If the endpoint happens to be hosted on the same hypervisor, it can be output directly to the logical endpoint's vNIC port on the virtual switch.[5]

At a receiving hypervisor, NVP has placed flow entries that match over both the physical ingress port for that end of the tunnel and the logical forwarding decision present in the tunnel header. The flow entry then outputs the packet to the corresponding local vNIC. A similar pattern applies to traffic received by service and gateway nodes.

The above discussion centers on a single L2 datapath, but generalizes to full logical topologies consisting of several L2 datapaths interconnected by L3 router

[4] We use OpenFlow [27] for this protocol, though any flow management protocol with sufficient flexibility would work.

[5] For brevity, we don't discuss logical MAC learning or stateful matching operations, but in short, the logical control plane can provide actions that create new lookup entries in the logical tables, based on incoming packets. These primitives allow the control plane to implement L2 learning and stateful ACLs, in a manner similar to advanced physical forwarding ASICs.

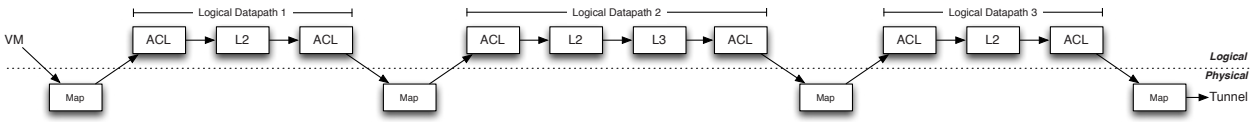


Figure 4: Processing steps of a packet traversing through two logical switches interconnected by a logical router (in the middle). Physical flows prepare for the logical traversal by loading metadata registers: first, the tunnel header or source VM identity is *mapped* to the first logical datapath. After each logical datapath, the logical forwarding decision is mapped to the next logical hop. The last logical decision is mapped to tunnel headers.

datapaths. In this case, the OVS flow table would hold flow entries for all interconnected logical datapaths and the packet would traverse each logical datapath by the same principles as it traverses the pipeline of a single logical datapath: instead of encapsulating the packet and sending it over a tunnel, the final action of a logical pipeline submits the packet to the first table of the next logical datapath. Figure 4 depicts how a packet originating at a source VM first traverses through a logical switch (with ACLs) to a logical router before being forwarded by a logical switch attached to the destination VM (on the other side of the tunnel). This is a simplified example: we omit the steps required for failover, multicast/broadcast, ARP, and QoS, for instance.

As an optimization, we constrain the logical topology such that logical L2 destinations can only be present at its edge.[6] This restriction means that the OVS flow table of a sending hypervisor needs only to have flows for logical datapaths to which its local VMs are attached as well as those of the L3 routers of the logical topology; the receiving hypervisor is determined by the logical IP destination address, leaving the last logical L2 hop to be executed at the receiving hypervisor. Thus, in Figure 4, if the sending hypervisor does not host any VMs attached to the third logical datapath, then the third logical datapath runs at the receiving hypervisor and there is a tunnel between the second and third logical datapaths instead.

3.2 Forwarding Performance

OVS, as a virtual switch, must classify each incoming packet against its entire flow table in software. However, flow entries written by NVP can contain wildcards for any irrelevant parts of a packet header. Traditional physical switches generally classify packets against wildcard flows using TCAMs, which are not available on the standard x86 hardware where OVS runs, and so OVS must use a different technique to classify packets quickly.[7]

To achieve efficient flow lookups on x86, OVS exploits *traffic locality*: the fact that all of the packets belonging to a single flow of traffic (*e.g.,* one of a VM's TCP connections) will traverse exactly the same set of flow entries. OVS consists of a kernel module and a userspace program; the kernel module sends the first packet of each new flow into userspace, where it is matched against the full flow table, including wildcards, as many times as the logical datapath traversal requires. Then, the userspace program installs *exact-match flows* into a flow table in the kernel, which contain a match for every part of the flow (L2-L4 headers). Future packets in this same flow can then be matched entirely by the kernel. Existing work considers flow caching in more detail [5, 22].

While exact-match kernel flows alleviate the challenges of flow classification on x86, NVP's encapsulation of all traffic can introduce significant overhead. This overhead does not tend to be due to tunnel header insertion, but to the operating system's inability to enable standard NIC hardware offloading mechanisms for encapsulated traffic.

There are two standard offload mechanisms relevant to this discussion. TCP Segmentation Offload (TSO) allows the operating system to send TCP packets larger than the physical MTU to a NIC, which then splits them into MSS-sized packets and computes the TCP checksums for each packet on behalf of the OS. Large Receive Offload (LRO) does the opposite and collects multiple incoming packets into a single large TCP packet and, after verifying the checksum, hands it to the OS. The combination of these mechanisms provides a significant reduction in CPU usage for high-volume TCP transfers. Similar mechanisms exist for UDP traffic; the generalization of TSO is called Generic Segmentation Offload (GSO).

Current Ethernet NICs do not support offloading in the presence of any IP encapsulation in the packet. That is, even if a VM's operating system would have enabled TSO (or GSO) and handed over a large frame to the virtual NIC, the virtual switch of the underlying hypervisor would have to break up the packets into standard MTU-sized packets and compute their checksums before encapsulating them and passing them to the NIC; today's NICs are simply not capable of seeing into the encapsulated packet.

To overcome this limitation and re-enable hardware offloading for encapsulated traffic with existing NICs, NVP uses an encapsulation method called STT [8].[8] STT places a standard, but fake, TCP header after the physical IP header. After this, there is the actual encapsulation header including contextual information that specifies, among other things, the logical destination of the packet. The actual logical packet (starting with its Ethernet header) follows. As a NIC processes an STT packet,

[6] We have found little value in supporting logical routers interconnected through logical switches without tenant VMs.

[7] There is much previous work on the problem of packet classification without TCAMs. See for instance [15, 37].

[8] NVP also supports other tunnel types, such as GRE [9] and VXLAN [26] for reasons discussed shortly.

it will first encounter this fake TCP header, and consider everything after that to be part of the TCP payload; thus, the NIC can employ its standard offloading mechanisms.

Although on the wire the STT packet looks like standard TCP packet, the STT protocol is stateless and requires no TCP handshake procedure between the tunnel endpoints. VMs can run TCP over the logical packets exchanged over the encapsulation.

Placing contextual information into the encapsulation header, at the start of the fake TCP payload, allows for a second optimization: this information is not transferred in every physical packet, but only once for each large packet sent to the NIC. Therefore, the cost of this context information is amortized over all the segments produced out of the original packet and additional information (*e.g.,* for debugging) can be included as well.

Using hardware offloading in this way comes with a significant downside: gaining access to the logical traffic and contextual information requires reassembling the segments, unlike with traditional encapsulation protocols in which every datagram seen on wire has all headers in place. This limitation makes it difficult, if not impossible, for the high-speed forwarding ASICs used in hardware switch appliances to inspect encapsulated logical traffic; however, we have found such appliances to be rare in NVP production deployments. Another complication is that STT may confuse middleboxes on the path. STT uses its own TCP transport port in the fake TCP header, however, and to date administrators have been successful in punching any necessary holes in middleboxes in the physical network. For environments where compliance is more important than efficiency, NVP supports other, more standard IP encapsulation protocols.

3.3 Fast Failovers

Providing highly-available dataplane connectivity is a priority for NVP. Logical traffic between VMs flowing over a direct hypervisor-to-hypervisor tunnel clearly cannot survive the failure of either hypervisor, and must rely on path redundancy provided by the physical network to survive the failure of any physical network elements. However, the failure of any of the new appliances that NVP introduces – service and gateway nodes – must cause only minimal, if any, dataplane outage.

For this reason, NVP deployments have multiple service nodes, to ensure that any one service node failure does not disrupt logical broadcast and multicast traffic. The controller cluster instructs hypervisors to load-balance their packet replication traffic across a bundle of service node tunnels by using flow hashing algorithms similar to ECMP [16]. The hypervisor monitors these tunnels using BFD [21]. If the hypervisor fails to receive heartbeats from a service node for a configurable period of time, it removes (without involving the controller cluster) the failed service node from the load-balancing tunnel bundle and continues to use the remaining service nodes.

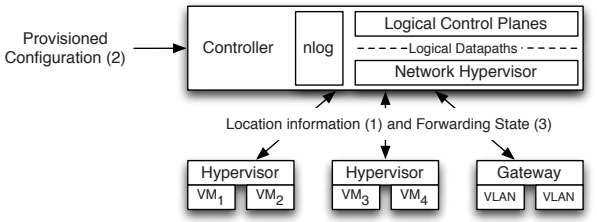


Figure 5: Inputs and outputs to the forwarding state computation process which uses nlog, as discussed in §4.3.

As discussed in Section 2, gateway nodes bridge logical networks and physical networks. For the reasons listed above, NVP deployments typically involve multiple gateway nodes for each bridged physical network. Hypervisors monitor their gateway tunnels, and fail over to backups, in the same way they do for service tunnels.[9] However, having multiple points of contact with a particular physical network presents a problem: NVP must ensure that no loops between the logical and physical networks are possible. If a gateway blindly forwarded logical traffic to the physical network, and vice versa, any traffic sent by a hypervisor over a gateway tunnel could wind up coming back into the logical network via another gateway attached to the same network, due to MAC learning algorithms running in the physical network.

NVP solves this by having each cluster of gateway nodes (those bridging the same physical network) elect a leader among themselves. Any gateway node that is not currently the leader will disable its hypervisor tunnels and will not bridge traffic between the two networks, eliminating the possibility of a loop. Gateways bridging a physical L2 network use a lightweight leader election protocol: each gateway broadcasts CFM packets [19] onto that L2 network, and listens for broadcasts from all other known gateways. Each gateway runs a deterministic algorithm to pick the leader, and if it fails to hear broadcasts from that node for a configurable period of time, it picks a new leader.[10] Broadcasts from an unexpected gateway cause all gateways to disable their tunnels to prevent possible loops.

4 Forwarding State Computation

In this section, we describe how NVP computes the forwarding state for the virtual switches. We focus on a single controller and defer discussion about distributing the computation over a cluster to the following section.

4.1 Computational Structure of Controller

The controller inputs and outputs are structured as depicted in Figure 5. First, hypervisors and gateways

[9] Gateway and service nodes do not monitor hypervisors, and thus, they have little per tunnel state to maintain.

[10] L3 gateways can use ECMP for active-active scale-out instead.

provide the controller with location information for vNICs over the OVS configuration protocol [31] (1), updating this information as virtual machines migrate. Hypervisors also provide the MAC address for each vNIC.[11] Second, service providers configure the system through the NVP API (see the following section) (2). This configuration state changes as new tenants enter the system, as logical network configuration for these tenants change, and when the physical configuration of the overall system (*e.g.,* the set of managed transport nodes) changes.

Based on these inputs, the logical control plane computes the logical lookup tables, which the network hypervisor augments and transforms into physical forwarding state (realized as logical datapaths with given logical lookup entries, as discussed in the previous section). The forwarding state is then pushed to transport nodes via OpenFlow and the OVS configuration protocol (3). OpenFlow flow entries model the full logical packet forwarding pipeline, whereas OVS configuration database entries are responsible for the tunnels connecting hypervisors, gateways and service nodes, as well as any local queues and scheduling policies.[12]

The above implies the computational model is entirely proactive: the controllers push *all* the necessary forwarding state down and do not process any packets. The rationale behind this design is twofold. First, it simplifies the scaling of the controller cluster because infrequently pushing updates to forwarding instructions to the switch, instead of continuously punting packets to controllers, is a more effective use of resources. Second, and more importantly, failure isolation is critical in that the managed transport nodes and their data planes must remain operational even if connectivity to the controller cluster is transiently lost.

4.2 Computational Challenge

The input and output domains of the controller logic are complex: in total, the controller uses 123 types of input to generate 81 types of output. A single input type corresponds to a single configured logical feature or physical property; for instance, a particular type of logical ACL may be a single logical input type, whereas the location of a vNIC may be a single physical input information type. Similarly, each output type corresponds to a single type of attribute being configured over OpenFlow or the OVS configuration protocol; for example, a tunnel parameter and particular type of ACL flow entries are both examples of individual output types.

The total amount of input state is also large, being proportional to the size of the MTD, and the state changes frequently as VMs migrate and tenants join, leave, and reconfigure their logical networks. The controller needs to react quickly to the input changes. Given the large total input size and frequent, localized input changes, a naïve implementation that reruns the full input-to-output translation on every change would be computationally inefficient. Incremental computation allows us to recompute only the affected state and push the delta down to the network edge. We first used a hand-written state machine to compute and update the forwarding state incrementally in response to input change events; however, we found this approach to be impractical due to the number of event types that need to be handled as well as their arbitrary interleavings. Event handling logic must account for dependencies on previous or subsequent events, deferring work or rewriting previously generated outputs as needed. In many languages, such code degenerates to a reactive, asynchronous style that is difficult to write, comprehend, and especially test.

4.3 Incremental State Computation with nlog

To overcome this problem, we implemented a domain-specific, declarative language called $nlog$ for computing the network forwarding state. It allows us to separate logic specification from the state machine that implements the logic. The logic is written in a declarative manner that specifies a *function* mapping the controller input to output, without worrying about state transitions and input event ordering. The state transitions are handled by a compiler that generates the event processing code and by a runtime that is responsible for consuming the input change events and recomputing all affected outputs. Note that nlog is not used by NVP's users, only internally by its developers; users interact with NVP via the API (see §5.3).

nlog declarations are Datalog queries: a single declaration is a join over a number of tables that produces immutable tuples for a *head table*. Any change in the joined tables results in (incremental) re-evaluation of the join and possibly in adding tuples to, or removing tuples from, this head table. Joined tables may be either *input tables* representing external changes (input types) or *internal tables* holding only results computed by declarations. Head tables may be internal tables or *output tables* (output types), which cause changes external to the nlog runtime engine when tuples are added to or removed from the table. nlog does not currently support recursive declarations or negation.[13] In total, NVP has about 1200 declarations and 900 tables (of all three types).

[11] The service provider's cloud management system can provision this information directly, if available.

[12] One can argue for a single flow protocol to program the entire switch but in our experience trying to fold everything into a single flow protocol only complicates the design.

[13] The lack of negation has had little impact on development but the inability to recurse complicates computations where the number of iterations is unknown at compile time. For example, traversing a graph can only be done up to maximum diameter.

```
# 1. Determine tunnel from a source hypervisor
#    to a remote, destination logical port.
tunnel(dst_lport_id, src_hv_id, encap, dst_ip) :-
   # Pick logical ports & chosen encap of a datapath.
   log_port(src_lport_id, log_datapath_id),
   log_port(dst_lport_id, log_datapath_id),
   log_datapath_encap(log_datapath_id, encap),

   # Determine current port locations (hypervisors).
   log_port_presence(src_lport_id, src_hv_id),
   log_port_presence(dst_lport_id, dst_hv_id),

   # Map dst hypervisor to IP and omit local tunnels.
   hypervisor_locator(dst_hv_id, dst_ip),
   not_equal(src_hv_id, dst_hv_id);

# 2. Establish tunnel via OVS db. Assigned port # will
#    be in input table ovsdb_tport. Ignore first column.
ovsdb_tunnel(src_hv_id, encap, dst_ip) :-
   tunnel(_, src_hv_id, encap, dst_ip);

# 3. Construct the flow entry feeding traffic to tunnel.
#    Before resubmitting packet to this stage, reg1 is
#    loaded with 'stage id' corresponding to log port.
ovs_flow(src_hv_id, of_expr, of_actions) :-
   tunnel(dst_lport_id, src_hv_id, encap, dst_ip),
   lport_stage_id(dst_lport_id, processing_stage_id),
   flow_expr_match_reg1(processing_stage_id, of_expr),
   # OF output action needs the assigned tunnel port #.
   ovsdb_tport(src_hv_id, encap, dst_ip, port_no),
   flow_output_action(port_no, of_actions);
```

Figure 6: Steps to establish a tunnel: 1) determining the tunnels, 2) creating OVS db entries, and 3) creating OF flows to output packets into tunnels.

The code snippet in Figure 6 has simplified nlog declarations for creating OVS configuration database tunnel entries as well as OpenFlow flow entries feeding packets to tunnels. The tunnels depend on API-provided information, such as the logical datapath configuration and the tunnel encapsulation type, as well as the location of vNICs. The computed flow entries are a part of the overall packet processing pipeline, and thus, they use a controller-assigned stage identifier to match with the packets sent to this stage by the previous processing stage.

The above declaration updates the head table *tunnel* for all pairs of logical ports in the logical datapath identified by *log_datapath_id*. The head table is an internal table consisting of rows each with four data columns; a single row corresponds to a tunnel to a logical port *dst_lport_id* on a remote hypervisor *dst_hv_id* (reachable at *dst_ip*) on a hypervisor identified by *src_hv_id* for a specific encapsulation type (*encap*) configured for the logical datapath. We use a function *not_equal* to exclude tunnels between logical ports on a single hypervisor. We will return to functions shortly.

In the next two declarations, the internal *tunnel* table is used to derive both the OVS database entries and OpenFlow flows to output tables *ovsdb_tunnel* and *ovs_flow*. The declaration computing the flows uses functions *flow_expr_reg1* and *flow_output_action* to compute the corresponding OpenFlow expression (matching over register 1) and actions (sending to a port assigned for the tunnel). As VMs migrate, the *log_port_presence* input table is updated to reflect the new locations of each *log_port_id*, which in turn causes corresponding changes to *tunnel*. This will result in re-evaluation of the second and third declaration, which will result in OVS configuration database changes that create or remove tunnels on the corresponding hypervisors, as well as OpenFlow entries being inserted or removed. Similarly, as tunnel or logical datapath configuration changes, the declarations will be incrementally re-evaluated.

Even though the incremental update model allows quick convergence after changes, it is not intended for reacting to dataplane failures at dataplane time scales. For this reason, NVP precomputes any state necessary for dataplane failure recovery. For instance, the forwarding state computed for tunnels includes any necessary backup paths to allow the virtual switch running on a transport node to react independently to network failures (see §3.3).

Language extensions. Datalog joins can only rearrange existing column data. Because most non-trivial programs must also transform column data, nlog provides extension mechanisms for specifying transformations in C++.

First, a developer can implement a *function table*, which is a virtual table where certain columns of a row are a stateless function of others. For example, a function table could compute the sum of two integer columns and place it in a third column, or create OpenFlow match expressions or actions (like in the example above). The base language provides various functions for primitive column types (*e.g.,* integers, UUIDs). NVP extends these with functions operating over flow and action types, which are used to construct the complex match expressions and action sequences that constitute the logical datapath flow entries. Finally, the developer is provided with *not_equal* to express inequality between two columns.

Second, if developers require more complicated transformations, they can hook an output and an input table together through arbitrary C++ code. Declarations produce tuples into the output table, which transforms them into C++ and feeds them to the output table C++ implementation. After processing, the C++ code transforms them back into tuples and passes them to nlog through the input table. For instance, we use this technique to implement hysteresis that dampens external events such as a network port status flapping.

5 Controller Cluster

In this section we discuss the design of the controller cluster: the distribution of physical forwarding state computation to implement the logical datapaths, the auxiliary distributed services that the distribution of the computation requires, and finally the implementation of the API provided for the service provider.

5.1 Scaling and Availability of Computation

Scaling. The forwarding state computation is easily parallelizable and NVP divides computation into a loosely-

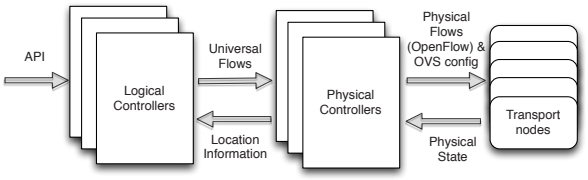


Figure 7: NVP controllers arrange themselves into two layers.

coupled two-layer hierarchy, with each layer consisting of a cluster of processes running on multiple controllers. We implement all of this computation in nlog, as discussed in the previous section.

Figure 7 illustrates NVP's two-layer distributed controller cluster. The top layer consists of *logical controllers*. NVP assigns the computation for each logical datapath to a particular live controller using its identifier as a sharding key, parallelizing the computation workload.

Logical controllers compute the flows and tunnels needed to implement logical datapaths, as discussed in Section 3. They encode all computed flow entries, including the logical datapath lookup tables provided by the logical control planes and instructions to create tunnels and queues for the logical datapath, as *universal flows*, an intermediate representation similar to OpenFlow but which abstracts out all transport-node-specific details such as ingress, egress or tunnel port numbers, replacing them with abstract identifiers. The universal flows are published over RPC to the bottom layer consisting of *physical controllers*.

Physical controllers are responsible for communicating with hypervisors, gateways and service nodes. They translate the location-independent portions of universal flows using node- and location-specific state, such as IP addresses and physical interface port numbers (which they learn from attached transport nodes), as well as create the necessary configuration protocol instructions to establish tunnels and queue configuration. The controllers then push the resulting *physical flows* (which are now valid OpenFlow instructions) and configuration protocol updates down to the transport nodes. Because the universal-to-physical translation can be executed independently for every transport node, NVP shards this responsibility for the managed transport nodes among the physical controllers.

This arrangement reduces the computational complexity of the forwarding state computation. By avoiding the location-specific details, the logical controller layer can compute one "image" for a single ideal transport node participating in a given logical datapath (having $O(N)$ tunnels to remote transport nodes), without considering the tunnel mesh between all transport nodes in its full $O(N^2)$ complexity. Each physical controller can then translate that image into something specific for each of the transport nodes under its responsibility.

Availability. To provide failover within the cluster, NVP provisions hot standbys at both the logical and physical controller layers by exploiting the sharding mechanism. One controller, acting as a sharding coordinator, ensures that every shard is assigned one master controller and one or more other controllers acting as hot standbys. On detecting the failure of the master of a shard, the sharding coordinator promotes the standby for the shard to master, and assigns a new controller instance as the standby for the shard. On detecting the failure of the standby for a shard, the sharding coordinator assigns a new standby for the shard. The coordinator itself is a highly-available service that can run on any controller and will migrate as needed when the current coordinator fails.

Because of their large population, transport nodes do not participate in the cluster coordination. Instead, OVS instances are configured by the physical controllers to connect to both the master and the standby physical controllers for their shard, though their master controller will be the only one sending them updates. Upon master failure, the newly-assigned master will begin sending updates via the already-established connection.

5.2 Distributed Services

NVP is built on the Onix controller platform [23] and thus has access to the elementary distributed services Onix provides. To this end, NVP uses the Onix replicated transactional database to persist the configuration state provided through API, but it also implements two additional distributed services.

Leader election. Each controller must know which shard it manages, and must also know when to take over responsibility of slices managed by a controller that has disconnected. Consistent hashing [20] is one possible approach, but it tends to be most useful in very large clusters; with only tens of controllers, NVP simply elects a sharding coordinator using Zookeeper [17]. This approach makes it easier to implement sophisticated assignment algorithms that can ensure, for instance, that each controller has equal load and that assignment churn is minimized as the cluster membership changes.

Label allocation. A network packet encapsulated in a tunnel must carry a label that denotes the logical egress port to which the packet is destined, so the receiving hypervisor can properly process it. This identifier must be globally unique at any point in time in the network, to ensure data isolation between different logical datapaths. Because encapsulation rules for different logical datapaths may be calculated by different NVP controllers, the controllers need a mechanism to pick unique labels, and ensure they will stay unique in the face of controller failures. Furthermore, the identifiers must be relatively compact to minimize packet overhead. We use Zookeeper to implement a label allocator that ensures labels will not

be reused until NVP deletes the corresponding datapath. The logical controllers use this label allocation service to assign logical egress port labels at the time of logical datapath creation, and then disseminate the labels to the physical controllers via universal flows.

5.3 API for Service Providers

To support integrating with a service provider's existing cloud management system, NVP exposes an HTTP-based REST API in which network elements, physical or logical, are presented as objects. Examples of physical network elements include transport nodes, while logical switches, ports, and routers are logical network elements. Logical controllers react to changes to these logical elements, enabling or disabling features on the corresponding logical control plane accordingly. The cloud management system uses these APIs to provision tenant workloads, and a command-line or a graphical shell implementation could map these APIs to a human-friendly interface for service provider administrators and/or their customers.

A single API request can require state from multiple transport nodes, or both logical and physical information. Thus, API operations generally merge information from multiple controllers. Depending on the operation, NVP may retrieve information on-demand in response to a specific API request, or proactively, by continuously collecting the necessary state.

6 Evaluation

In this section, we present measurements both for the controller cluster and the edge datapath implementation.

6.1 Controller Cluster

Setup. The configuration in the following tests has 3,000 simulated hypervisors, each with 21 vNICs for a total of 63,000 logical ports. In total, there are 7000 logical datapaths, each coupled with a logical control plane modeling a logical switch. The average size of a logical datapath is 9 ports, but the size of each logical datapath varies from 2 to 64. The test configures the logical control planes to use port ACLs on 49,188 of the logical ports and generic ACLs for 1,553 of the logical switches.[14]

The test control cluster has three nodes. Each controller is a bare-metal Intel Xeon 2.4GHz server with 12 cores, 96GB of memory, and 400GB hard disk. The logical and physical computation load is distributed evenly among the controllers, with one master and one standby per shard. The physical network is a dedicated switched network.

Each simulated hypervisor is a Linux VM that contains an OVS instance with a TUN device simulating each virtual interface on the hypervisor. The simulated hypervisors run within XenServer 5.6 physical hypervisors, and are connected via Xen bridges to the physical network. We test four types of conditions the cluster may face.

[14]This serves as our base validation test; other tests stress the system further both in scale and in complexity of configurations.

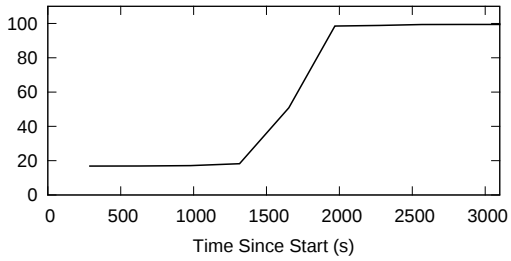


Figure 8: Cold start connectivity as a percentage of all pairs connected.

Cold start. The cold start test simulates bringing the entire system back online after a major datacenter disaster in which all servers crash and all volatile memory is lost. In particular, the test starts with a fully configured system in a steady state, shuts down all controllers, clears the flows on all OVS instances, and restarts everything.

Restore. The restore test simulates a milder scenario where the whole control cluster crashes and loses all volatile state but the dataplane remains intact.

Failover. The failover test simulates a failure of a single controller within a cluster.

Steady state. In the steady state test, we start with a converged idle system. We then add 10 logical ports to existing switches through API calls, wait for connectivity correctness on these new ports, and then delete them. This simulates a typical usage of NVP, as the service provider provisions logical network changes to the controller as they arrive from the tenant.

In each of the tests, we send a set of pings between logical endpoints and check that each ping either succeeds if the ping is supposed to succeed, or fails if the ping is supposed to fail (*e.g.,* when a security policy configuration exists to reject that ping). The pings are grouped into rounds, where each round measures a sampling of logical port pairs. We continue to perform ping rounds until all pings have the desired outcome and the controllers finish processing their pending work. The time between the rounds of pings is 5-6 minutes in our tests.

While the tests are running, we monitor the sizes of all the nlog tables; from this, we can deduce the number of flows computed by nlog, since these are stored in a single table. Because nlog is running in a dedicated thread, we measure the time this thread was running and sleeping to get the load for nlog computation.

Finally, we note that we do not consider routing convergence of any kind in the tests. Physical routing protocols handle any failures in the connectivity between the nodes, and thus, aside from tunnel failovers, the network hypervisor can remain unaware of such events.

Results. Figure 8 shows the percentage of correct pings over time for the cold start test, beginning at time 0. It

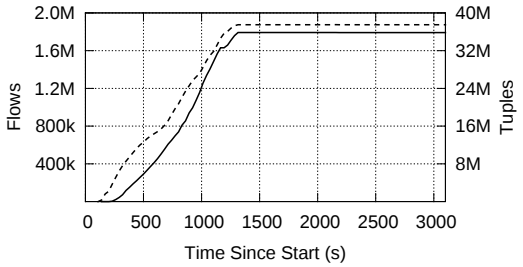


Figure 9: Total physical flows (solid line) and nlog tuples (dashed line) in one controller after a cold start.

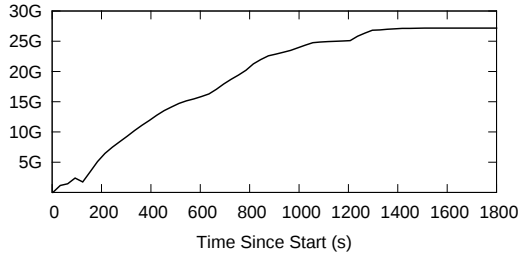


Figure 10: Memory used by a controller after a cold start.

starts at 17% because 17% of the pings are expected to fail, which they do in the absence of any flows pushed to the datapath. Note that, unlike typical OpenFlow systems, NVP does not send packets for unclassified flows to the controller cluster; instead, NVP precomputes all necessary flow changes after each configuration change. Thus, cold start represents a worst-case scenario for NVP: the controller cluster must compute all state and send it to the transport nodes before connectivity can be fully established. Although it takes NVP nearly an hour to achieve full connectivity in this extreme case, the precomputed flows greatly improve dataplane performance at steady state. While the cold-start time is long, it is relevant only in catastrophic outage conditions and thus considered reasonable: after all, if hypervisors remain powered on, the data plane will also remain functional even though the controllers have to go through cold-start (as in the restore test below).

The connectivity correctness is not linear for two reasons. First, NVP does not compute flows for one logical datapath at a time, but does so in parallel for all of them; this is due to an implementation artifact stemming from arbitrary evaluation order in nlog. Second, for a single ping to start working, the correct flows need to be set up on all the transport nodes on the path of the ping (and ARP request/response, if any).

We do not include a graph for connectivity correctness during the restore or failover cases, but merely note that connectivity correctness remains at 100% during these tests. The connectivity is equally well-maintained in the case of adding or removing controllers to the cluster, but again we do not include a graph here for brevity.

Figure 9 shows the total number of tuples, as well as the total number of flows, produced by nlog on a single controller over time during the cold start test. The graphs show that nlog is able to compute about 1.8M flows in about 20 minutes, involving about 37M tuples in total across all nlog tables. This means that to produce 1 final flow, we have an average of 20 intermediary tuples, which points to the complexity of incorporating all of the possible factors that can affect a flow. After converging, the measured controller uses approximately 27G of memory, as shown in Figure 10.

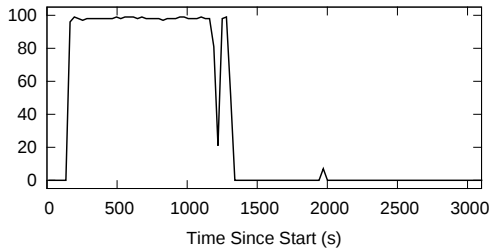


Figure 11: nlog load during cold start.

Since our test cluster has 3 controllers, 1.8M flows is 2/3 of all the flows in the system, because this one controller is the master for 1/3 of the flows and standby for 1/3 of the flows. Additionally, in this test nlog produces about 1.9M tuples per minute on average. At peak performance, it produces up to 10M tuples per minute.

Figure 11 shows nlog load during the cold start test. nlog is almost 100% busy for 20 minutes. This shows that controller can read its database and connect to the switches (thereby populating nlog input tables) faster than nlog can process it. Thus, nlog is the bottleneck during this part of the test. During the remaining time, NVP sends the computed state to each hypervisor.

A similar load graph for the steady state test is not included but we merely report the numeric results, highlighting nlog's ability to process incremental changes to inputs: the addition of 10 logical ports (to the existing 63,000) results in less than 0.5% load for a few seconds. Deleting these ports results in similar load. This test represents the usual state of a real deployment – constantly changing configuration at a modest rate.

6.2 Transport Nodes

Tunnel performance. Table 1 shows the throughput and CPU overhead of using non-tunneled, STT, and GRE to connect two hypervisors. We measured throughput using Netperf's TCP_STREAM test. Tests ran on two Intel Xeon 2.0GHz servers with 8 cores, 32GB of memory, and Intel 10Gb NICs, running Ubuntu 12.04 and KVM. The CPU load represents the percentage of a single CPU core used, which is why the result may be higher than 100%. All the results only take into account the CPU used to switch traffic in the hypervisor, and not the CPU used by the VMs. The test sends a single flow between two VMs on the different hypervisors.

We see that the throughput of GRE is much lower

	No encap	STT	GRE
TX CPU load	49%	49%	85%
RX CPU load	72%	119%	183%
Throughput	9.3Gbps	9.3Gbps	2.4Gbps

Table 1: Non-tunneled, STT, and GRE performance.

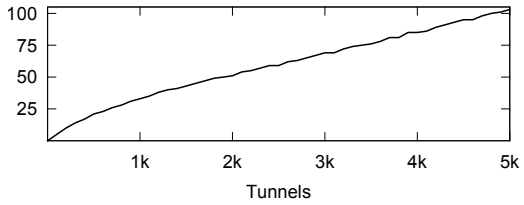


Figure 12: Tunnel management CPU load as a % of a single core.

and requires more CPU than either of the other methods due to its inability to use hardware offloading. However, STT's use of the NIC's TCP Segmentation Offload (TSO) engine makes its throughput performance comparable to non-tunneled traffic between the VMs. STT uses more CPU on the receiving side of the tunnel because, although it is able to use LRO to coalesce incoming segments, LRO does not always wait for all packet segments constituting a single STT frame before passing the result of coalescing down to OS. After all, for NIC the TCP payload is a byte stream and not a single jumbo frame spanning multiple datagrams on the wire; therefore, if there is enough time between two wire datagrams, the NIC may decide to pass the current result of the coalescing to the OS, just to avoid introducing excessive extra latency. STT requires the full set of segments before it can remove the encapsulation header within the TCP payload and deliver the original logical packet, and so on these occasions it must perform the remaining coalescing in software.

Connection set up. OVS connection setup performance has been explored in the literature (see *e.g.,* [32–34]) and we have no new results to report here, though we return to the topic shortly in Section 8.

Tunnel scale. Figure 12 shows the keepalive message processing cost as the number of tunnels increases. This test is relevant for our gateways and service nodes, which have tunnels to potentially large numbers of hypervisors and must respond to keepalives on all of these tunnels. The test sends heartbeats at intervals of 500ms, and the results indicate a single CPU core can process and respond to them in a timely manner for up to 5000 tunnels.

7 Related Work

NVP borrows from recent advances in datacenter network design (*e.g.,* [1, 12, 30]), software forwarding, programming languages, and software defined networking, and thus the scope of related work is vast. Due to limited space, we only touch on topics where we feel it useful to distinguish our work from previous efforts. While NVP relies on SDN [3, 4, 13, 14, 23, 27] in the form of an OpenFlow forwarding model and a control plane managed by a controller, NVP requires significant extensions.

Virtualization of the network forwarding plane was first described in [6]; NVP develops this concept further and provides a detailed design of an edge-based implementation. However, network virtualization as a general concept has existed since the invention of VLANs that slice Ethernet networks. Slicing as a mechanism to share resources is available at various layers: IP routers are capable of running multiple control planes over one physical forwarding plane [35], and FlowVisor introduced the concept of slicing to OpenFlow and SDN [36]. However, while slicing provides isolation, it does not provide either the packet or control abstractions that enable tenants to live within a faithful logical network. VMs were proposed as a way to virtualize routers [38] but this is not a scalable solution for MTDs.

NVP uses a domain-specific declarative language for efficient, incremental computation of all forwarding state. Expressing distributed (routing) algorithms in datalog [24, 25] is the most closely related work, but it focuses on concise, intuitive modeling of distributed algorithms. Since the early versions of NVP, our focus has been on structuring the computation within a single node to allow efficient incremental computation. Frenetic [10, 11] and Pyretic [28] have argued for reactive functional programming to simplify the implementation of packet forwarding decisions, but they focused on reactive packet processing rather than the proactive computations considered here. Similarly to NVP (and [6] before it), Pyretic [28] identifies the value of an abstract topology and uses it to support composing modular control logic.

8 Discussion

After having presented the basic design and its performance, we now return to discuss which aspects of the design were most critical to NVP's success.

8.1 Seeds of NVP's Success

Basing NVP on a familiar abstraction. While one could debate which abstraction best facilitates the management of tenant networks, the key design decision (which looks far more inevitable now than four years ago when we began this design) was to make logical networks look *exactly* like current network configurations. Even though current network control planes have many flaws, they represent a large installed base; NVP enables tenants to use their current network policies *without modification* in the cloud, which greatly facilitates adoption of both NVP and MTDs themselves.

Declarative state computation. Early versions of NVP used manually designed state machines to compute

forwarding state; these rapidly became unwieldy as additional features were added, and the correctness of the resulting computations was hard to ensure because of their dependency on event orderings. By moving to nlog, we not only ensured correctness independent of ordering, but also reduced development time significantly.

Leveraging the flexibility of software switching. Innovation in networking has traditionally moved at a glacial pace, with ASIC development times competing with the IETF standardization process for which is slower. On the forwarding plane, NVP is built around Open vSwitch (OVS); OVS went from a crazy idea to a widely-used component in SDN designs in a few short years, with no haggling over standards, low barriers to deployment (since it is merely a software upgrade), and a diverse developer community. Moreover, because it is a software switch, we could add new functionality without concerns about artificial limits on packet matches or actions.

8.2 Lessons Learned

Growth. With network virtualization, spinning up a new environment for a workload takes a matter of minutes instead of weeks or months. While deployments often start cautiously with only a few hundred hypervisors, once the tenants have digested the new operational model and its capabilities their deployments typically witness rapid growth resulting in a few thousand hypervisors.

The story is similar for logical networks. Initial workloads require only a single logical switch connecting a few tens of VMs, but as the deployments mature, tenants migrate more complicated workloads. At that point, logical networks with hundreds of VMs attached to a small number of logical switches interconnected by one or two logical routers, with ACLs, become more typical. The overall trends are clear: in our customers' deployments, both the number of hypervisors as well as the complexity and size of logical networks tend to grow steadily.

Scalability. In hindsight, the use of OpenFlow has been a major source of complications, and here we mention two issues in particular. First, the overhead OpenFlow introduces within the physical controller layer became the limiting factor in scaling the system; unlike the logical controller which has computational complexity of $O(N)$, the need to tailor flows for each hypervisor (as required by OpenFlow) requires $O(N^2)$ operations. Second, as the deployments grow and clusters operate closer to their memory limits, handling transient conditions such as controller failovers requires careful coordination.

Earlier in the product lifecycle, customers were not willing to offload much computation into the hypervisors. While still a concern, the available CPU and memory resources have grown enough over the years that in the coming versions of the product, we can finally run the physical controllers within the hypervisors without concern. This has little impact to the overall system design but moving the physical controllers down to the hypervisors reduces the cluster requirements by an order of magnitude. Interestingly, this also makes OpenFlow a local protocol within the hypervisor, which limits its impact on the rest of the system.

Failure isolation. While the controller cluster provides high-availability, the non-transactional nature of OpenFlow results in situations where switches operate over inconsistent and possibly incomplete forwarding state due to a controller crash or connectivity failure between the cluster and hypervisor. While a transient condition, customers expect better consistency between the switches and controllers. To this end, the next versions of NVP make all declarative computation and communication channels "transactional": given a set of changes in the configuration, all related incremental updates are computed and pushed to the hypervisors as a batch which is then applied atomically at the switch.

Forwarding performance. Exact match flow caching works well for typical workloads where the bulk of the traffic is due to long-lived connections; however, there are workloads where short-lived connections dominate. In these environments, exact match caching turned out to be insufficient: even if the packet forwarding rates were sufficiently high, the extra CPU load introduced was deemed unacceptable by our customers.

As a remedy, OVS replaced the exact match flow cache with *megaflows*. In short, unlike exact match flow cache, megaflows caches *wildcarded* forwarding decisions matching over larger traffic aggregates than a single transport connection. The next step is to re-introduce the exact match flow cache and as a result there will be three layers of packet processing: exact match cache handling packets after the first packets of transport connections (one hash lookup), megaflows that handle most of the first packets of transport connections (a single flow classification) and a slow path finally handling the rest (a sequence of flow classifications).

9 Conclusion

Network virtualization has seen a lot of discussion and popularity in academia and industry, although little has been written about practical network virtualization systems, or how they are implemented and deployed. In this paper, we described the design and implementation of NVP, a network virtualization platform, that has been deployed in production environments for last few years.

Acknowledgments. We would like to thank our shepherd, Ratul Mahajan, and the reviewers for their valuable comments.

10 References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of SIGCOMM*, August 2008.

[2] T. J. Bittman, G. J. Weiss, M. A. Margevicius, and P. Dawson. Magic Quadrant for x86 Server Virtualization Infrastructure. Gartner, June 2013.

[3] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and Implementation of a Routing Control Platform. In *Proc. NSDI*, April 2005.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *Proc. of SIGCOMM*, August 2007.

[5] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In *Proc. of HotNets*, October 2008.

[6] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. In *Proc. of PRESTO*, November 2010.

[7] D. W. Cearley, D. Scott, J. Skorupa, and T. J. Bittman. Top 10 Technology Trends, 2013: Cloud Computing and Hybrid IT Drive Future IT Models. Gartner, February 2013.

[8] B. Davie and J. Gross. A Stateless Transport Tunneling Protocol for Network Virtualization (STT). Internet draft. draft-davie-stt-04.txt, IETF, September 2013.

[9] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Generic Routing Encapsulation (GRE). RFC 2784, IETF, March 2000.

[10] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a Network Programming Language. In *Proc. of SIGPLAN ICFP*, September 2011.

[11] N. Foster, R. Harrison, M. L. Meola, M. J. Freedman, J. Rexford, and D. Walker. Frenetic: A High-Level Language for OpenFlow Networks. In *Proc. of PRESTO*, November 2010.

[12] 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 *Proc. of SIGCOMM*, August 2009.

[13] 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 CCR*, 35(5), 2005.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 38, 2008.

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

[16] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF, November 2000.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for Internet-scale systems. In *Proc. of USENIX ATC*, June 2010.

[18] Server Virtualization Multiclient Study. IDC, January 2012.

[19] IEEE. 802.1ag - Virtual Bridged Local Area Networks Amendment 5: Connectivity Fault Management. Standard, IEEE, December 2007.

[20] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of STOC*, May 1997.

[21] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, IETF, June 2010.

[22] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting Route Caching: The World Should Be Flat. In *Proc. of PAM*, April 2009.

[23] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of OSDI*, October 2010.

[24] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proc. of SOSP*, October 2005.

[25] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proc. of SIGCOMM*, August 2005.

[26] M. Mahalingam et al. VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. Internet draft. draft-mahalingam-dutt-dcops-vxlan-08.txt, IETF, February 2014.

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38(2):69–74, 2008.

[28] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proc. of NSDI*, April 2013.

[29] B. Munch. IT Market Clock for Enterprise Networking Infrastructure, 2013. Gartner, September 2013.

[30] 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 *Proc. of SIGCOMM*, August 2009.

[31] B. Pfaff and B. Davie. The Open vSwitch Database Management Protocol. RFC 7047, IETF, December 2013.

[32] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending Networking into the Virtualization Layer. In *Proc. of HotNets*, October 2009.

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

[34] L. Rizzo, M. Carbone, and G. Catalli. Transparent Acceleration of Software Packet Forwarding Using Netmap. In *Proc. of INFOCOM*, March 2012.

[35] E. Rosen and Y. Rekhter. BGP/MPLS IP Virtual Private Networks. RFC 4364, IETF, February 2006.

[36] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the Production Network Be the Testbed? In *Proc. of OSDI*, October 2010.

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

[38] 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 *Proc. of SIGCOMM*, August 2008.

Operational Experiences with Disk Imaging in a Multi-Tenant Datacenter

USENIX Symposium on Networked Systems Design and Implementation—Operational Systems Track

Kevin Atkinson * Gary Wong Robert Ricci

University of Utah School of Computing
{kevina, gtw, ricci}@cs.utah.edu www.emulab.net

Abstract

Disk images play a critical role in multi-tenant datacenters. In this paper, the first study of its kind, we analyze operational data from the disk imaging system that forms part of the infrastructure of the Emulab facility. This dataset spans four years and more than a quarter-million disk image loads requested by Emulab's users. From our analysis, we draw observations about the nature of the images themselves (for example: how similar are they to each other?) and about usage patterns (what is the statistical distribution of image popularity?). Many of these observations have implications for the design and operation of disk imaging systems, including how images are stored, how caching is employed, the effectiveness of pre-loading, and strategies for network distribution.

1 Introduction

Computers in datacenters are frequently re-allocated from one purpose to another, need to have their software upgraded, or need to be returned to a known "clean" state. This type of re-provisioning is particularly important in *multi-tenant* datacenters [4], which are shared by a large number of applications running on behalf of different clients. Notably, this is the model adopted by "Infrastructure as a Service" (IaaS) clouds such as Amazon EC2 [2], Rackspace [11], and datacenters managed with software such as OpenStack [15]. These facilities provide physical or virtual servers (infrastructure) on which users run their own operating systems and applications [9, 15].

The primary means for initializing user resources is to load them with an initial *disk image*, which is a block-level snapshot of a filesystem containing an installed operating system and set of applications. Typically, a cloud will provide a set of images that any user may install on servers that they provision (*facility images*). Users may also create their own images (*user images*): this is commonly accomplished by loading a facility image, customizing it, and taking a snapshot of the resulting disk.

*Work done at the University of Utah; now at Rice University

Large multi-tenant facilities have hundreds to hundreds of thousands of servers and thousands to millions of users [5]. A busy facility may have many thousands of user images and provision tens of thousands of servers per day. Disk images are commonly written to drives attached to the host; EC2, for example, calls this "instance storage" [1], and it is available on nearly all VM types. Disk imaging is on the critical path for provisioning servers, which cannot be booted until the requested image has been loaded. Images can consume significant resources on the facility, including the space used to store them and the network bandwidth required to distribute them to the hosts on which they are to be used. Thus, understanding disk images and their use is important to the design and operation of multi-tenant datacenters.

In this paper, we study four years' worth of data from the operation of the Emulab testbed [16], a multi-tenant facility with approximately six hundred hosts and over five thousand user accounts. The data we examine covers 279,972 requests for disk images (Section 2) and is, to our knowledge, the only dataset currently available to the public that contains detailed traces of disk imaging in a multi-tenant datacenter. It allows us to study properties of the disk images themselves as well as how they are used by the facility's users, and we draw a number of conclusions that are applicable to the design and operation of imaging systems. Our key findings include:

- **Section 3**: There is substantial block-level similarity between many images, suggesting that deduplicating storage is appropriate. The lifespan of images varies greatly, from days to years, and many images go unused for months at a time, making multi-tier data storage attractive.

- **Section 4**: The working set of images is quite small (mean: 12 per day, 30 per week), making caching of frequently used images potentially effective. However, the makeup of this working set changes frequently, and there are no dominant images. The daily working set size grows linearly with the number of users, but the total number of facility and user

images follow different curves.

- **Section 5**: The popularity of user images follows a heavy-tailed distribution, while the popularity of facility images does not. Most users skew heavily towards using either facility provided images or custom images, not both. While most users do not create their own images, those who do number among the facility's heaviest users.

- **Section 6**: We consider the technique of pre-loading popular facility images, allowing some requests to be satisfied without waiting for the image to load. We find that two factors control the potential benefit from this strategy: (a) the ratio of the working set size to the number of idle disks available for pre-loading, and (b) the ratio of the rate at which the facility can load disks to the arrival rate of requests.

- **Section 7**: Differential loading (pre-loading a base image, then transferring only differing blocks as required) shows potential. In order to be effective, it will require development of sophisticated prediction techniques that take into account both the popularity of images themselves and their block-level similarity to each other.

We conclude in Section 8 with several concrete suggestions regarding the design and operation of disk imaging systems, and point to fertile areas for future work.

2 Dataset

Emulab is a network testbed widely used by the distributed systems and networking communities. An experimenter describes a network in terms of links and hosts. Included in this specification is the disk image to be loaded on each host. Emulab then provisions servers, physical or virtual, loading the requested disk image. This provisioning is done on demand as requests come in, and there is only limited support for ahead-of-time scheduling or batch jobs. The facility provides a number of standard images, including "default" images that are used if the user does not explicitly request an image. Many users create their own images by booting from a facility image, customizing it (for example, by installing software packages or modifying the operating system), and taking a snapshot. This user image can be referenced in future requests, saving the user the effort of re-installing the packages they use, or to scale out to much larger experiments. This basic model of image usage and creation is similar to that used in most IaaS clouds [14].

Emulab is capable of provisioning both physical and virtual machines; physical machines are the most commonly allocated resource. While many IaaS clouds provision solely virtual machines, we believe that this difference does not have a significant impact on conclusions drawn from the dataset: in either case, the user is presented with the abstraction of an PC on which they may load and boot an operating system. While the details of operating systems that run within physical and virtual machines may vary, the quantity and diversity of users' desired images is unlikely to be affected.

Emulab uses block-level disk images and distributes them using the Frisbee [6] disk imaging system. The format uses filesystem-aware compression, meaning that it does not store disk blocks that are not used by the filesytem, and compresses the allocated blocks with `zlib` [7] for efficient storage. Frisbee uses IP multicast to distribute images, and is highly optimized so that the bottleneck in image distribution and installation is the write speed of the target disk. The amount of time required to load a disk image depends on the number of used blocks in the filesystem that it contains, but is typically on the order of a few minutes. Facility images are visible to and may be requested by all users. User images are visible only to their creators unless the creator decides to make the image public, which few do.

2.1 Dataset Details

The dataset that we study covers four years of disk image requests on Emulab, from March 2009 to March 2013. The dataset covers a total of 279,972 requests for 714 unique images. The requests were made by 368 users, at an average rate of 192 disk images loaded per day. The records cover the identity of the image, the user making the request, and the timestamp at which the request was made. Furthermore, the data indicates whether each image was a facility image or a user image, and whether it was requested explicitly by the user or was chosen as a default because the user did not specify an image. To preserve user anonymity, users and user images are assigned random integers as identifiers in this paper. We present the names of facility images using their Emulab-assigned names; user images are presented as $user/image$ pairs.

One of the things we studied was the block-level differences between images. Our primary interest in examining the contents of images is to determine the potential savings from loading a "base" image (usually a facility image), then transferring and writing only the disk blocks required to transform it into a particular "derived" image (usually a user image). We define the difference of two images A and B as:

$$\Delta(A, B) = |\forall i \in b : B[i] \neq A[i]| \quad (1)$$

where b is the set the indices of allocated storage blocks in image B, and $A[i]$ and $B[i]$ are the contents of images A and B, respectively, at index i. This measure directly

captures the numbers of blocks in image B that would need to be written to a disk that already contains image A. We define $\delta(A, B)$ as the fraction of blocks that would need to be written: that is,

$$\delta(A, B) = \frac{\Delta(A, B)}{|b|} \quad (2)$$

The Emulab dataset does not record the provenance of images (that is, which user images where based on which facility images). We assume that each user image U was based on the facility image F for which $\Delta(F, U)$ is minimized. For a particular image U, we refer to this base image as U_B. Emulab allow users to delete their image files: only 37.4% (267) of the images found in the request traces were available for analysis of block-level similarities. Though large in number, the missing images were relatively unpopular, accounting for only 15.8% of all requests. Emulab also allows its users to modify images, so the image files that we analyzed represented a snapshot of image contents at a particular point in time.

2.2 Removing Sources of Bias in the Dataset

We filtered the dataset to remove certain biases. First, we omit all uses of the facility by its operational staff: the maintenance, testing, etc. that they perform is likely to follow different patterns than users of the facility. Second, as a network testbed, Emulab supports a feature known as "delay nodes," [13, 12] which perform a traffic-shaping role that does not represent a function present in most multi-tenant datacenters. Third, Emulab includes some resources that are not the standard PC servers used in clouds and datacenters: these include wireless nodes, programmable network hardware, and sensors. This filtering removed 183,824 of the original 463,796 requests (39.6%), 215 images (23.1%), and 30 users (7.5%), leaving us with the 279,972 requests, 714 images, and 368 users that we studied.

It is worth making special note of Emulab's "default" images. If an Emulab experimenter does not specify a particular disk image in their experiment description, they get a default that is, for historical reasons, quite old. Due to their ages, the default images are not very popular. Most users select the facility image that best meets their needs; as a result, the presence of a default does not have a dominating effect on the way that users select images.

2.3 Users and Projects

For the purposes of this study, we consider users at the level of *organizations*. Emulab groups individual users into "projects." These loosely-defined groups represent research groups, classes, or cross-institution collaborations. Because of this, they are analogous to businesses that purchase time on a cloud such as EC2, or individual business units that share a company-wide datacenter. In the remainder of this paper, we consider all individuals who are part of the same project to be a singe "user" of the facility—when we refer to "users," we are referring to Emulab projects. The number of individuals who requested disk image loads over this time period was 1,301.

2.4 Limitations of This Study

The Emulab dataset is, to our knowledge, the only one of its type currently publicly available. Therefore, we cannot quantitatively assess the degree to which it matches other multi-tenant facilities. We believe our analysis remains valuable nonetheless, for two reasons. First, it is the only analysis to date to apply such a large quantity of real-world data to the problem of improving disk imaging systems. Second, we conjecture that the most *fundamental* findings in our work remain applicable in other environments, even if specifics (such as the λ parameter to the facility image popularity distribution) differ.

Our dataset covers a large number of disk image loads, but comes from a mid-sized facility. We attempt to analyze the effects of facility size in Section 4.3, but application of our conclusions to larger facilities necessarily involves extrapolation. In addition, two features unique to Emulab affected our ability to run certain analyses.

First, the nature of resource allocation in Emulab makes it difficult to study the inter-arrival times of image requests. Emulab's primary unit of resource allocation is the *experiment*: a collections of hosts that together make up a network experiment. In contrast, most IaaS clouds consider only individual servers or "instances," and the cloud has no semantic information about which instances are contributing to the same application. Thus, image requests in Emulab arrive in well-defined bursts that do not have a direct analog in many other datacenters. Deploying an application in a datacenter or cloud does often involve provisioning of multiple machines in a short timeframe; however, we have no data that would allow us to analyze whether experiment sizes in Emulab are representative of burst sizes in other environments. For this reason, we avoid analyzing this aspect of the dataset, and all of our analyses are with respect to individual loads of disk images rather than Emulab experiments.

Second, we chose not to analyze the relative popularity of the operating systems contained in the images (eg. Linux vs. BSD, or the relative popularities of Linux distributions). Emulab's user base is overwhelmingly comprised of academic researchers and students, and their OS preferences may not be representative of a broader population. In particular, while Emulab supports Windows, it constitutes a small fraction of all Emulab use—almost certainly a smaller fraction than would be seen in other

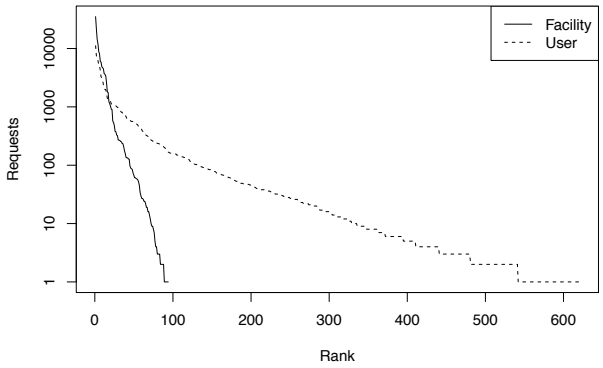


Figure 1: Requests for facility and user images, sorted on the x axis by popularity. Note the log-scale y axis.

settings. We restrict our analysis to the popularity of disk images rather than the operating systems they contain, and it is possible that this distribution is affected by the operating system preferences of Emulab's user base.

3 Storage of Disk Images

We begin our study by examining the basic properties of the images in our dataset, with an eye towards understanding how they should be stored. We pay special attention to the relationship between images that are provided by the facility and images that are created by users; as we will see through further analysis, these images have different characteristics that warrant different treatment.

3.1 Prevalence of User Images

During the 48 months covered by our dataset, there were a total of 368 users. Of these, nearly two thirds (231) used only facility images, and slightly over one third (137, or 37.2%) used at least one user image. This implies that optimizing the provisioning of facility images can improve the experience of a majority of users. For example, if a suitable set of facility images can be identified for preloading on to servers, this could take image loading out of the critical path for creation of those users' instances. We explore this issue further in Section 6.

The number of users who request user images, however, is non-negligible, suggesting that an imaging system should also take their needs into account. In fact, we find that there are more user images in our dataset (619) than facility images (94), meaning that, on average, each user who creates at least one disk image creates 4.5 of them.

3.2 Popularity of User Imags

The top of Table 1 shows the relative popularity of facility and user images. We see that the percentage of requests for user images is over 44%; since only 37.2% of users create their own images, this imples that this set of users are heavier users of the testbed by at least 18%. Table 1 also shows all images that made up at least 1% of the requests. Of these twenty four images, ten are user images. Note that RHL90-STD and FEDORA10-STD each appear twice, because they are both common explicitly requested images and also images loaded by default. The complete image popularity data is plotted in Figure 1. We can see that the number of user images is much larger than the number of facility images, but that the population of user images contains many images that are used few times. Together, the top 17 facility images are more popular than the top 17 user images (the 17th facility image had 1,772 requests, and the 17th user 1,330). From the 18th image onwards, the user images are more popular—the 18th user image had 1,260 requests and the 18th facility image had

	Image name	Requests	%
	All facility images	155,617	55.6%
u	All user images	124,355	44.4%
	RHL90-STD [D]	21,993	7.9%
	FEDORA10-STD	18,042	6.4%
	UBUNTU10-STD	14,402	5.1%
	RHL90-STD	13,182	4.7%
	FC4-UPDATE	12,097	4.3%
u	715/10	11,156	4.0%
	FBSD410-STD	8,916	3.2%
	FEDORA8-STD	8,153	2.9%
u	237/69	7,512	2.7%
u	296/35	7,179	2.6%
u	787/24	6,243	2.2%
	UBUNTU70-STD	6,021	2.2%
	UBUNTU12-64-STD	5,834	2.1%
u	787/14	5,231	1.9%
u	226/44	5,198	1.9%
	FEDORA10-UPDATE	4,861	1.7%
	CENTOS55-64-STD	4,710	1.7%
	FC6-STD	4,455	1.6%
u	762/69	4,213	1.5%
	FC4-WIRELESS	3,700	1.3%
	FC4-STD	3,615	1.3%
	FEDORA10-STD [D]	3,604	1.3%
	UBUNTU11-64-STD	3,383	1.2%
u	624/89	3,277	1.2%
u	238/50	3,113	1.1%
u	226/51	2,899	1.0%

Table 1: Total requests for all user and facility images. Also shown are the number of requests for all images that account for more than 1% of all requests. User images are marked with a 'u' in the left column, and images requested implicitly as defaults are marked with a '[D]'; explicit requests for default images are counted separately.

1,233. Both facility and user images have tails consisting of images that were requested fewer than ten times, but this tail is much more prevalent in the case of user images, where the tail represents nearly half of all user images.

From this data, we can conclude that facility images dominate, but that there are a small number of user images that are as popular as some facility images.

3.3 Image Lifespan

The Emulab dataset does not include explicit creation and deletion dates for images. Thus, we define the lifespan of an image to be the number of days between when the image was first seen in the request stream and when it was last seen. Note that this will tend to underestimate lifespan: some images were likely first used before our dataset begins, and some will continue to be used after the end of the dataset.

A histogram of user image lifespans can be seen in Figure 2. While the majority of images have very short lifespans, there is a long tail: several were used throughout the entire four years covered by the dataset. The observed mean lifespan is 100.4 days.

We found the number of images with short lifespans to be quite surprising, so we examined them in greater detail, and it became clear that a large majority of these short-lifespan images were requested only on a single day: 196 of the 619 user images (31%) fall into this usage pattern. This suggests that a number of users create images for the purposes of running a single experiment, a conclusion borne out by looking at the experiment metadata.

Finally, we looked at how long user images "go idle". We found that it is common for user images to have gaps of months in between requests for them. During this time, there is no need to have the images constantly available; they could be moved to cheaper, but slower, storage. The distribution of the maximum idle periods for the 214 user images with a lifespan of at least 30 days is shown in Figure 3. In total, 162 of the images (76% of long-lived images, and 26% of images overall) had gaps in usage of one month or more. Two images even had gaps of over two years between successive uses.

3.4 Block-Level Differences Between Images

We next examined how much user images differ from the facility images they are based on. We use the definitions of $\Delta(A, B)$, $\delta(A, B)$, and "base" images given in Section 2.1. Figure 4 shows a histogram of similarities between user images and their associated base facility images. From this figure, it is clear that many user images do show significant similarities to their bases—most are more than 50% similar, with a significant peak in the 60%–80% range. This is in line with findings from smaller studies in the past [8]. There is also a significant tail of more than twenty images images with very low similarity (below 10%) to their base images.

Overall, these numbers point to two potential strategies for improving disk imaging systems. First, they suggest that significant storage savings can be had by storing images in a deduplicating storage system [10], which would

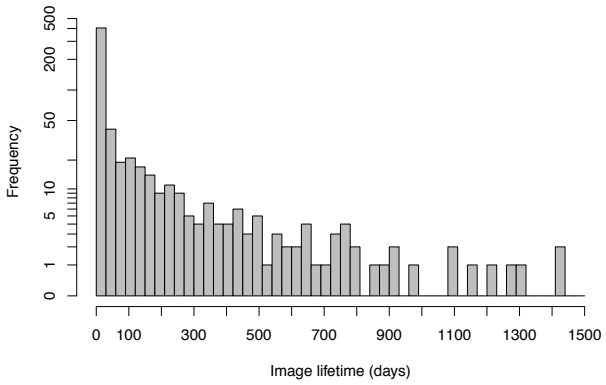


Figure 2: Histogram of the lifespans of user images. Note that the y axis is log-scale.

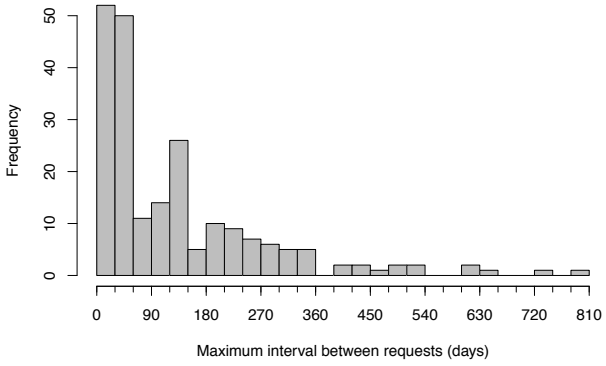


Figure 3: Histogram of usage gaps for user images.

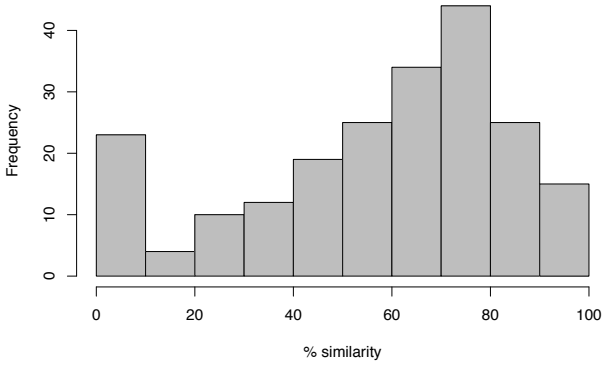


Figure 4: Histogram of similarity ($1 - \delta(U_B, U)$) between user images and their associated base images. Higher percentages indicate more similarity.

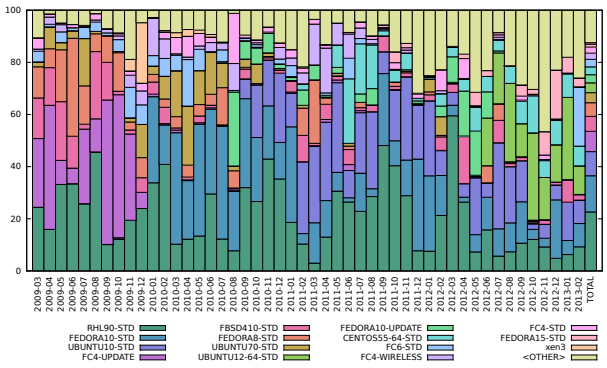


Figure 5: Variation of facility image popularity over time. The fifteen most popular facility images are shown.

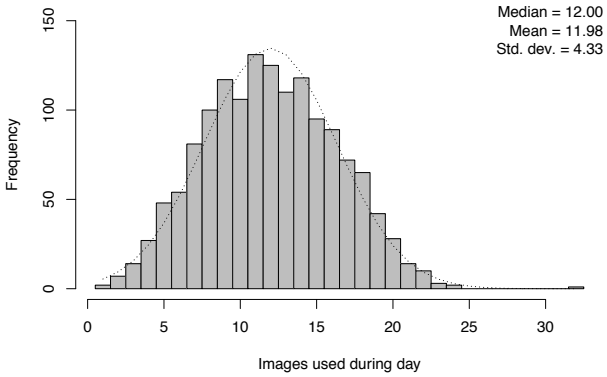


Figure 6: Histogram showing the distribution of the working set size over one-day periods (midnight to midnight).

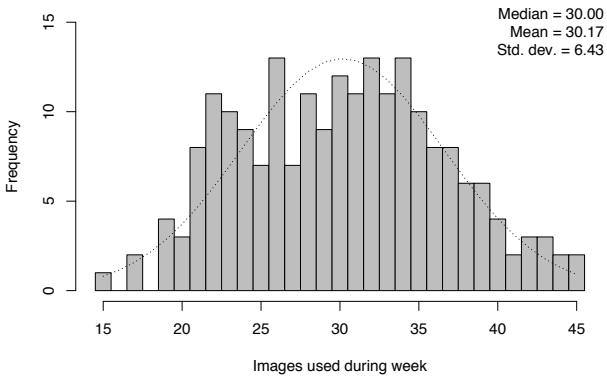


Figure 7: Histogram showing the distribution of the working set size over one week periods (Sunday to Saturday).

store only one copy of the blocks that the base and derived images have in common. Second, they suggest that the technique of differential disk loading, which transforms a base image into a derived image by writing only the blocks that differ, has a potential for reducing the time and bandwith for distributing the user images. We explore the latter in detail in Section 7.

4 Working Set Size and Caching Potential

Having looked at the images themselves, we turn our attention to trends of usage over time, paying particular attention to the working set; understanding the size and composition of the working set is critical to designing strategies for caching and pre-loading.

4.1 No Dominant Images

If a small set of facility images dominates the request stream, it would be possible to design the disk imaging system around that fact. In particular, it would make sense to pre-load most or all idle disks with popular images, allowing user requests to be satisfied without waiting for a disk to load. This is the policy adopted by Emulab: the images labeled '[D]' in Table 1 are loaded as part of the process of freeing machines for the next user.

As we can see in Figure 5, there is no such dominant image. The popularity of all facility images fluctuates wildly from month to month, with new images becoming popular quickly, old images falling out of favor, and some images swinging between popular and unpopular. Even the default images, which remain active throughout the entire time period, sees large changes in popularity. Note that we do not distinguish between explicit and implicit requests for default images as we did in Table 1; for the purposes of disk loading, these two cases are equivalent.

As a result, we conclude that the strategy of pre-loading a single default image is unhelpful. It is, in fact, counterproductive: servers must be taken out of circulation while they are loaded with the default image, and most are re-loaded a second time when requested by a user. If pre-loading strategies are to be useful, they will require more sophisticated methods for predicting future requests.

4.2 Size and Variation of the Working Set

Figure 6 depicts the working set size (number of unique images requested) over one-day periods. The mean working set size is quite small, at a mean of 11.98 images per day—this represents only 1.7% of the total number of images. While there is some variation in the working set size, it is not large: it follows a normal distribution with a standard deviation of 4.33. This result is encouraging from the perspective of caching: it suggests that only a small fraction of images need to be available for quick loading at any point in time, and that others could be stored in cheaper, slower storage systems. Figure 7 shows the distribution over week-long periods. The average working set size is approximately two and a half times larger than the daily average, and again follows a normal distribution with a reasonably small standard deviation.

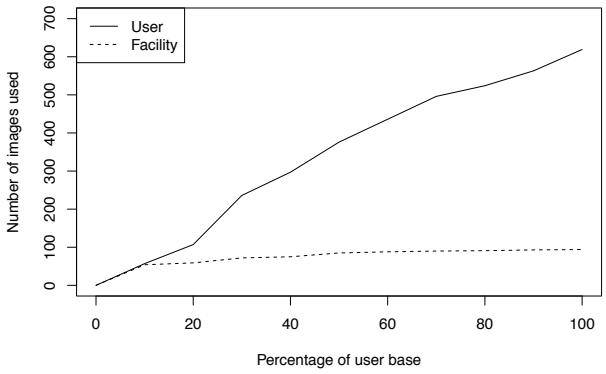


Figure 8: Total number of images used over four years when considering random subsamples of the Emulab userbase.

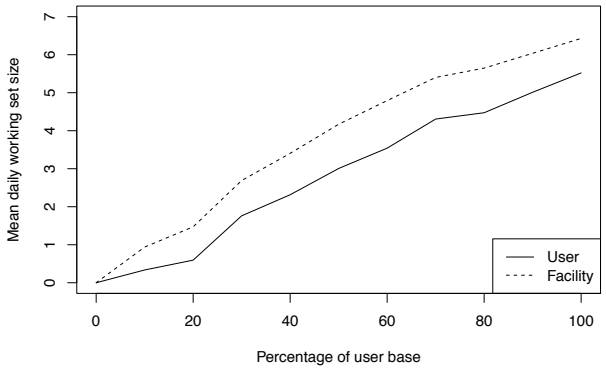


Figure 9: Mean daily working set size when considering random subsamples of the Emulab userbase.

4.3 Scaling of the Working Set

To get a feel for how the size of the working set might vary on facilities larger or smaller than Emulab, we subsampled our data to simulate differently sized userbases. Figure 8 shows the total number of images used over the 4-year period when considering only 10% of the userbase, 20%, etc. The set of facility images quickly reaches saturation (all images are used at least once) and stops growing with additional users. The set of user images, on the other hand, grows linearly with respect to the number of users. This is explained by simple intuition: the set of useful facility images is more a function of the facility than of the userbase, while more users mean more user-created images. Thus, we can expect that a facility with many more users than Emulab will have a greater number of user images in proportion roughly to its greater userbase, but that its set of facility images will not be larger by the same proportion. Indeed, Amazon EC2, which has a userbase that is at least three orders of magnitude larger than Emulab, advertises less than thirty images provided directly by AWS [3] and less than a hundred public images provided by their business partners. In comparison, Emulab has 94 public facility-provided images.

However, this does not quite tell the whole story. Figure 9 shows the same subsampings, but this time looks at the mean daily working set size. Here, we see that the number of images loaded in a typical day increases linearly with the userbase for both facility and user images. Thus, we can expect that facilities much larger than Emulab do exhibit larger working sets. The working set of facility images is capped by the total number of such images, so very large facilities are likely to include most or all of their facility images in the daily working set.

The general trend we can expect, is that for small facilities, the daily image working set size is in direct proportion to the size of the userbase. For large facilities, the working set will contain a relatively small set of facility images, and a very large set of user images; however, we find that the fraction of requests that are for user images stays fairly constant regardless of the size of the userbase, meaning that these requests must necessarily be diverse.

5 Users' Behavior

We now turn our attention to the behavior of individual users; a facility that understands how its users interact with images is in a better position to provide the interfaces and image management tools that they require.

5.1 Distribution of Image Popularity

In distributions with "light" tails, such as the normal distribution, a relatively small subset of the population accounts for most of the popularity. For "heavy tailed" distributions (defined as those whose tail is not bounded by the exponential [17]), this effect is less pronounced, and it takes more of the population to cover the same level of popularity. We compared the popularity distributions of facility and user images separately to exponential distributions chosen to match the sample means. We found that facility images are a reasonably good match for the corresponding exponential distribution (with Kolmogorov-Smirnov statistic $\sqrt{n}D_n = 1.13$), but user images are not ($\sqrt{n}D_n = 5.54$). As can be seen in Figure 10, the tail for user images lies substantially above the exponential.

This is a key finding: user-created images have a significant heavy tail, while facility-provided images do not. The primary consequence of this discrepancy is that strategies that depend on being able satisfy a large number of requests with a relatively small number of images (such as pre-loading, examined in detail in Section 6), will be more effective with facility images than with user images.

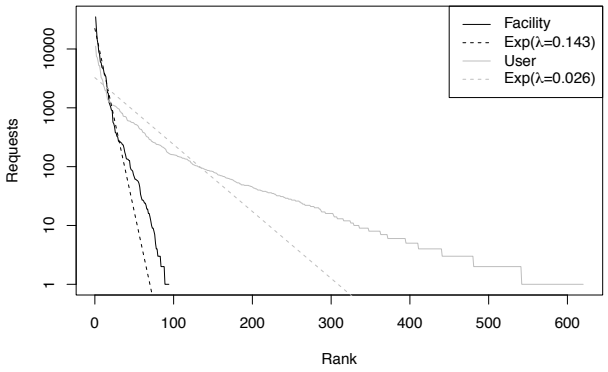


Figure 10: Distribution of image popularity compared to the exponential (shown as dashed lines); note the log-scale y axis.

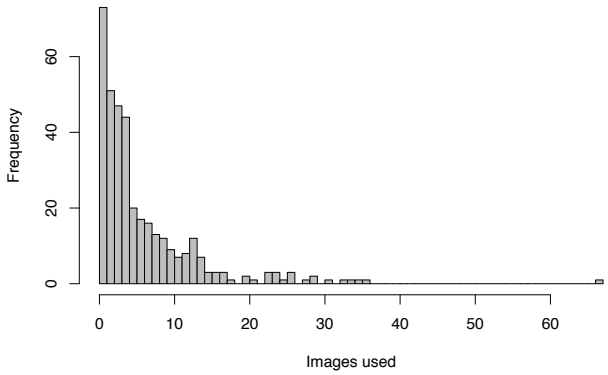


Figure 11: Histogram showing the number of users who use different quantities of images.

5.2 Users and Images

As we can see in Figure 11, most users use a relatively small set of images. There are, however, two surprising features of this data.

Only 20% of users used a single image—a large majority used two or more. We believe that this is due to three factors. First, since our sample period covers four years, many users likely migrated to newer versions of images as operating systems were updated. Second, any user who creates a custom image will use at least two images: they will request the base facility image at least once, then move to the custom image they create. Third, users may have started off using the default images provided by Emulab, found them unsuitable for their needs, and switched to non-default images.

Another surprising feature is that there are a small number of users who use a very large number of images. Twenty users use at least 20 images, and one outlier uses more than 60.

Figure 12: Profile of users making at least 500 disk image requests. Requests for facility images are shown as bars above the axis, and user images are below the axis.

5.3 Behavior of Heavy Users

Because user images are created by customizing facility images, we can expect that all users will employ facility images at least once, and likely a few times. The question remains, however, whether users tend to use primarily facility images, primarily their own images, or some balanced mixture of the two. We are particularly interested in the answer to this question for heavy users of the facility.

Figure 12 shows a profile of the heaviest users (those who made at least 500 image requests) from the Emulab dataset. Two important facts are evident. First, while a few users do mix facility and user images (i.e. have bars both above and below the axis in the figure), most tend to skew heavily towards one or the other. Second, among the twenty heaviest users, twelve employ primarily user images. Past this point, facility images dominate. This clearly establishes that custom user images are a "power user" feature: their dominant use is by a relatively small number of users, who use them heavily.

6 Prediction and Pre-Loading

We now turn our attention to techniques that may allow the facility to service user requests more quickly. The first technique that we consider is pre-loading: if it is possible to predict which images will be requested in the near future, the facility can pre-load them onto idle disks. If the predictions are correct, users requests may be satisfied immediately; if not, the user will have to wait for their image to be loaded. Note that this strategy does not save bandwidth on the datacenter's image distribution network; it simply shifts the image distribution to before the user's request arrives. In fact, pre-loading may *increase* the bandwidth used for distributing images: in the case of mispredictions, a node pre-loaded with one disk image may need to be re-loaded with another.

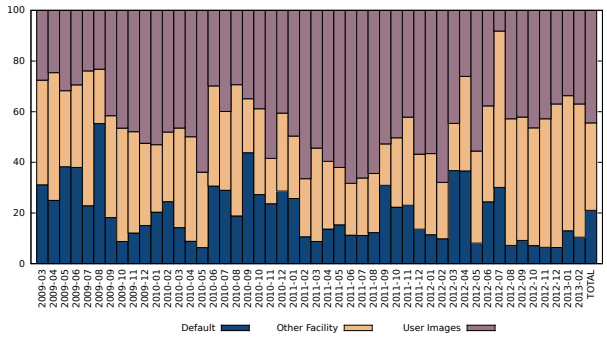


Figure 13: Percentage of requests for three classes of images.

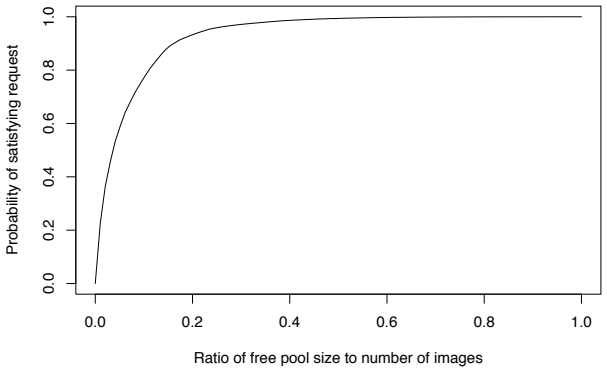


Figure 14: Fraction of requests satisfied from pre-loaded images for varying ratios of free pool size to the working set size.

6.1 Free Pool vs. Working Set Size

We begin with the observation from Section 3.2 that the popularity of user images has a much longer, heavier tail than the set of requests for facility images. Therefore, strategies targeting prediction of facility images are likely to bear more fruit. We also recall from Section 4.1 that there does not exist a consistently dominant image, though Section 4.2 showed us that the working set size over a day is fairly small. This small working set size is encouraging from a prediction standpoint.

An illustration of the potential for prediction can be found in Figure 13, which shows three classes of image requests. Requests for default images can be satisfied by simply pre-loading default images without complicated prediction strategies. This strategy is clearly ineffective in Emulab, as few requests are for the defaults. On top of these are requests for non-default facility images, which represent attractive targets for pre-loading. Finally, we see that approximately 40% of requests are for user images, which are a poor target for prediction because of their long tail. Thus, we target the 60% of requests that are for the relatively predictable facility images.

We first consider how the size of the free pool affects the potential for prediction, where the free pool is defined as the set of idle nodes or disks that are not in use and are thus available for pre-loading. We consider a simple model in which we assume that the inter-arrival time of requests is greater than the time required to load an image. (We will relax this assumption below.) In this model, the determinant of prediction accuracy is the ratio between the size of the free pool and the working set size. In this scenario, the best prediction mechanism is to pre-load those N disks with the N most popular images.

Figure 14 shows the percentage of requests for facility images satisfied under this model, using the empirical request and working set data from Emulab. Intuitively, if there are no disks available for pre-loading, it is not possible to satisfy any requests from pre-loaded machines, and if one can pre-load the entire working set of images (the ratio is 1.0 or greater), it is possible to satisfy all requests. Because the distribution of facility image popularity is roughly exponential, the ability to load the top 25% of images satisfies 95% of all facility requests.

It is interesting to consider how this result applies to different sizes of facilities. In many cases, the size of the free pool will be a fraction of the physical resources, meaning that it is much larger, in absolute terms, for larger facilities. At the same time, we have seen that the working set size of facility images grows linearly with the userbase, but is capped at a relatively small size by the total number of facility images. The practical effect is that small facilities (tens of nodes) are likely to fall on the left side of the curve in Figure 14, meaning that pre-loading is not likely to be particularly effective. Large facilities (thousands of nodes), on the other hand, are likely to be on the far right, with free pool sizes that far exceed the number of facility images—for them, pre-loading is likely to be able to satisfy all requests for facility images. In between these extremes, a facility needs to carefully consider the free pool to working set ratio to determine whether pre-loading makes sense.

6.2 Reload Rate vs. Arrival Rate

Our previous experiment made the simplifying assumption that request inter-arrival time was smaller than the time required to re-load an image; this enables the facility to ensure that the N most popular facility images are loaded at all times, and that only one copy of each image needs to be kept pre-loaded. We now consider the relationship between the arrival rate of new requests and the rate at which the facility can pre-load images in response. If bursts of requests arrive at a faster rate the the facility can re-image, it is useful to have more than one pre-loaded copy of each image. It is also possible for bursts of requests to outpace the facility's ability to keep the image loaded, meaning that there can be mispredictions even for very popular images.

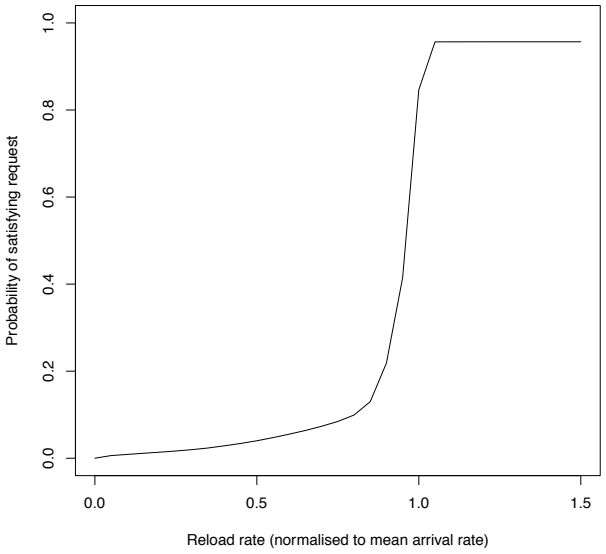


Figure 15: Fraction of requests satisfied against the rate at which images can be pre-loaded.

We model this scenario using standard tools from queuing theory: each image is modeled as a queue, with a number of queue slots equal to the number of disks onto which the image is pre-loaded. The distribution of pre-loaded images is taken directly from the observed distribution of requests; using our results from Section 5.1, we model this distribution as being exponential with $\lambda = 0.143$. We make the standard queuing theory assumption that requests arrive according to a Poisson process [18]. We picked a facility size of 1,000 disks, with an average utilization rate of 90%, meaning that on average, 100 disks are available for pre-loading.

Figure 15 shows the results of a Monte Carlo simulation using this model. We varied the ratio of reload rate to the mean request arrival rate, and find that this ratio is critical. If the facility can reload images at a faster rate than requests arrive (the area to the right of the 1.0 ratio), it can easily keep the proper set of facility images preloaded and can satisfy most requests for these images; this matches the case modeled in Figure 14. If the reload rate is lower (to the left of the 1.0 ratio), the value of preloading falls quickly, as bursts of requests overwhelm the facility's ability to keep a pre-loaded pool that contains the appropriate set of images.

We conclude that pre-loading facility images can be an effective strategy for reducing user wait time, but that the critical determining factors for its success are: (1) the ratio betwen the size of the free pool and the working set size; and (2) the ratio between the facilities' reload rate and the mean arrival rate.

7 Differential Disk Loading

The second optimization we consider targets requests for user images: it may be possible to pre-load facility images, and when requests for user images arrive, load only the blocks that differ. This differential loading strategy is attractive for two reasons. As we saw in Section 5.1, the distribution of user image popularity has a heavy tail, making it difficult to pre-load enough of them to satisfy many requests. But, as we saw in Section 3.4, user images have high levels of similarity to the smaller set of facility images. Thus, we have the potential to reduce user wait times by picking a pre-loaded facility image and doing a fast load of just the blocks that differ. In this section, we develop metrics that quantify the potential benefits of differential disk loading and give us a general understanding of the potential effectiveness of this technique. In order to realize these benefits, additional methods for predicting future requests would need to be developed, which take into account not only image popularity, but also block-level similarity between the pre-loaded images and the images that may be layered on top of them.

We consider only the problem of finding the differences between two disk images, and not the more general problem of taking the difference between a disk image and arbitrary disk state (i.e. the state in which the disk is left by the previous user). Earlier work [6] has shown that disk distribution and installation can run at the full write speed of the target disk, meaning that schemes that require reading disk contents before writing are likely to slow the process down, and are likely to be fruitful only in cases where users do not write much to the disk.

7.1 Limits to Savings

As we have seen, the set of facility images is smaller and more predictable than the set of user images. Thus, as with the last section, we continue to pre-load only facility images; when a user image U is requested, if its base image U_B has been pre-loaded, we need to transfer only $\Delta(U_B, U)$ blocks instead of the full $|u|$ blocks belonging to the image. Clearly, this strategy relies on having the correct set of base images pre-loaded. To simplify, we start by assuming that we have an oracle that tells us what facility images to pre-load or sufficient capacity to preload all facility images; we relax this assumption below.

We begin by defining the number of disk blocks loaded for user images when differential loading is not in use (i.e. the entire user image must be loaded). For an individual image U, this quantity is:

$$|u| \cdot U_C \tag{3}$$

Recall that u is the set of block addresses with defined values in image U, and therefore $|u|$ represents the size

of the image. We define U_C to be the number of times the image is loaded. Intuitively, then, this quantity is simply the number of blocks in the image multiplied by the number of times the image is used.

To obtain the total number of blocks loaded across the universe of all user images, $\mathbb{U}$, we sum the total blocks loaded for each image $U \in \mathbb{U}$:

$$\sum_{U \in \mathbb{U}} |u| \cdot U_C \tag{4}$$

To adapt these equations for differential loading, we substitute $\Delta(U_B, U)$ for $|u|$, giving us the number of blocks that must be loaded assuming the base image has been pre-loaded. This gives us the total number of blocks:

$$\sum_{U \in \mathbb{U}} \Delta(U_B, U) \cdot U_C \tag{5}$$

Differential Savings Potential (DSP): The maximum relative savings from differential loading (assuming the correct U_B images are always loaded) is derived by combining Equation 4 and Equation 5:

$$\text{DSP} = \sum_{U \in \mathbb{U}} \frac{|u| - \Delta(U_B, U)}{|u|} U_C \tag{6}$$

In the Emulab dataset, the values for Equation 4 and Equation 5 are 174 TB and 78 TB, giving a DSP of 0.55. This indicates that, in the presence of an oracle, the Emulab facility could save over half of the blocks it transfers for user images at request time, potentially halving the average time users must wait for custom images to load.

Adjusted Differential Savings (ADS): We next relax the assumption of an oracle. To do so, we use the notation $P[I]$ to indicate the probability that image I is pre-loaded on the facility. We adjust Equation 6 to indicate that with some probability, the user request can be fulfilled with differential loading because the requisite base image is loaded. If not, the entire image must be loaded (resulting in no savings):

$$\text{ADS} = \sum_{U \in \mathbb{U}} P[U_B] \frac{|u| - \Delta(U_B, U)}{|u|} U_C \tag{7}$$

Note that if $P[U_B] = 1$ for all images (perfect prediction), this gives us Equation 6. For smaller $P[U_B]$ values (worse predictions), the adjusted savings are lower than the savings potential, which fits with the intuitive notion that sub-optimal pre-loading will reduce the value of differential loading.

7.2 Savings With Predictions

Figure 16 shows the effectiveness of differential loading as a function of the fraction of facility images that are pre-loaded. The y axis of this graph represents the fraction of blocks that must be loaded at request time, with lower numbers being better, and the limit being $1 - \text{DSP}$ (0.45). Along the x axis, we show the fraction of facility images loaded—we rank facilitiy images by an adjusted popularity that is the sum of their own popularity and the popularity of all users images that use that facility image as a base, and then pre-load the x most popular. What we can see is that relatively few facility images act as bases for user images, so it is necessary to pre-load only a small subset of them (approximately 20%) in order to get most of the benefit of differential loading. This implies that this technique can be effective even on facilities that have low free pool to working set ratios.

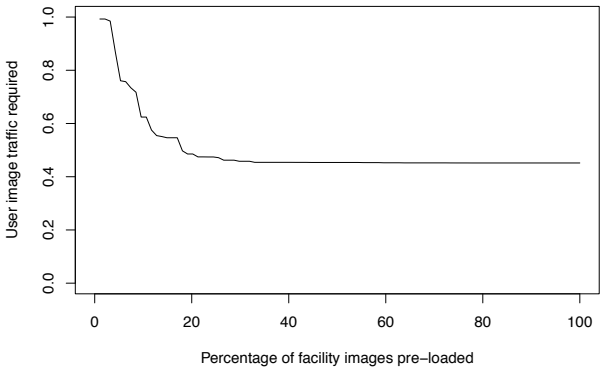


Figure 16: Network traffic required to load user images, when various facility images may be pre-loaded.

Also of interest in Figure 16 is that, for our dataset, the most popular facility images (the default images) are not commonly used as bases for user images—this accounts for the small plateau on the left of the graph. We hypothesize that this is due to the age of Emulab's defaults.

8 Recommendations and Future Work

In our exploration of the Emulab disk image request dataset, we have uncovered a number of properties that can be used to guide the operation and design of disk image storage and installation systems. Based on our analysis, we make the following recommendations:

- Storing images in a deduplicating image store is likely to result in substantial savings. Reads from deduplicating stores can be slow, but the working set size is small enough that it is possible to cache images in faster storage.
- Focusing pre-loading strategies on facility images is likely to produce the best results. The tail of user images is much longer and heavier than the one for facility images, and only a few user images approach the popularity of the heaviest-used facility

images. For very large facilities, it is likely that most facility images appear in the daily working set, making prediction straightforward.

- Pre-loading of a single default image is not a useful strategy, as the diversity of user requests means that no one image, even the default, is dominant on any time scale.

- For small facilities (those where the number of idle disks is significantly smaller than the working set size), pre-loading is likely not a valuable strategy. For large facilities, the number of idle disks is likely to be much larger than the working set size, making simple pre-loading strategies highly effective. To accurately model the effectiveness of pre-loading for mid-sized facilities, additional study of request inter-arrival distributions is necessary.

- Large facilities would do well to focus on techniques that allow them to sustain high reload rates. The only way for pre-loading to be effective is to keep this rate significantly above the request arrival rate, which is likely to be high for large facilities. Techniques of interest include distribution using multicast and image distribution servers spread throughout the datacenter.

- Differential loading has the potential to be effective, especially on facilities with limited free pools. It shows the potential to halve the number of disk blocks transferred to satisfy user requests, but that potential depends on correct predictions when pre-loading the appropriate base images. This changes the criteria for pre-loading, since base images should be selected not only on their own popularity, but also on the popularity of images that may be laid down on top and their block-level similarly with the base image. This complex optimization problem presents an interesting area for future study.

An anonymized version of the dataset used for thus study, plus all code used to analyze it and produce the figures for this paper, can be found at:

`http://aptlab.net/p/tbres/nsdi14`

Acknowledgments

We would like to thank the administrators of Emulab for their assistance in collecting the data used for this study. We would also like to thank Dave Andersen, our shepherd Bruce Maggs, and the anonymous reviewers for their valuable comments. This work was supported by NSF under award CNS-0709427.

References

[1] Amazon Web Services. Amazon EC2 instance store: User guide. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html`.

[2] Amazon Web Services. Amazon Elastic Compute Cloud website. `http://aws.amazon.com/ec2/`.

[3] Amazon Web Services. Amazon Machine Images (AMIs). `https://aws.amazon.com/amis`.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[5] L. A. Barroso and U. Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, volume 6 of *Synthesis Lectures on Computer Architecture*. Morgan and Claypool, 2009.

[6] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with Frisbee. In *Proc. of the USENIX Annual Technical Conference (ATC)*, pages 283–296, San Antonio, TX, June 2003.

[7] Jean-loup Gailly and Mark Adler. zlib website. `http://www.zlib.org`.

[8] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. of SYSTOR, the Israeli Experimental Systems Conference*, May 2009.

[9] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proc. of the Workshop on Cloud Computing and its Applications (CCA)*, 2008.

[10] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, Jan. 2002.

[11] Rackspace US, Inc. Rackspace hosting website. `http://www.rackspace.com/`.

[12] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31–41, Jan. 1997.

[13] P. Sanaga, J. Duerig, R. Ricci, and J. Lepreau. Modeling and emulation of Internet paths. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Apr. 2009.

[14] The OpenStack Team. OpenStack user documentation. `http://docs.openstack.org/user-guide/`.

[15] The OpenStack Team. OpenStack website. `http://www.openstack.org`.

[16] The University of Utah. Emulab website. `http://www.emulab.net/`.

[17] Wikipedia: Heavy-tailed Distribution. `http://en.wikipedia.org/wiki/Heavy-tailed_distribution`.

[18] Wikipedia: Poisson Process. `http://en.wikipedia.org/wiki/Poisson_process`.

VPN Gate: A Volunteer-Organized Public VPN Relay System with Blocking Resistance for Bypassing Government Censorship Firewalls

Operational Systems Track

Daiyuu Nobori and Yasushi Shinjo
Department of Computer Science, University of Tsukuba, Japan

Abstract

VPN Gate is a public VPN relay service designed to achieve blocking resistance to censorship firewalls such as the Great Firewall (GFW) of China. To achieve such resistance, we organize many volunteers to provide a VPN relay service, with many changing IP addresses. To block VPN Gate with their firewalls, censorship authorities must find the IP addresses of all the volunteers. To prevent this, we adopted two techniques to improve blocking resistance. The first technique is to mix a number of innocent IP addresses into the relay server list provided to the public. The second technique is collaborative spy detection. The volunteer servers work together to create a list of spies, meaning the computers used by censorship authorities to probe the volunteer servers. Using this list, each volunteer server ignores packets from spies. We launched VPN Gate on March 8, 2013. By the end of August it had about 3,000 daily volunteers using 6,300 unique IP addresses to facilitate 464,000 VPN connections from users worldwide, including 45,000 connections and 9,000 unique IP addresses from China. At the time VPN Gate maintained about 70% of volunteer VPN servers as unblocked by the GFW.

1. Introduction

Some countries in the world have censorship firewalls operated by their governments to prohibit access to servers in foreign countries. For instance, the Great Firewall (GFW) of China blocks access to Twitter, Facebook, and YouTube. Internet users in countries subject to censorship often use overseas public relay servers to bypass censorship firewalls. Public proxies, VPN servers, and Tor nodes [7] are popular examples of such relay servers. Usually, the IP addresses of relay servers are publically available for user convenience. A censorship authority can easily block these relays, however, by adding the IP addresses to its firewall blocking list. Moreover, the Chinese authority, in particular, scans for unlisted Tor nodes and blocks them automatically [19]. Tor relays currently have no blocking resistance [12] against such scanning activities.

In this research, we have built a public VPN relay server system with blocking resistance to censorship firewalls such as the GFW. We call this system VPN Gate. To achieve blocking resistance, VPN Gate uses frequently changing IP addresses that are provided by volunteers. The central list server, called the *VPN Gate List Server*, manages a list of the IP addresses of all active VPN servers. We call this list the *Server List*. A user can get only part of the Server List and connect his/her PC to an active VPN server in the list. The user can then communicate with blocked Internet servers through the active VPN server. It is hard for a censorship authority to block all the active VPN servers in VPN Gate.

It is important for anti-censorship systems to achieve blocking resistance. We adopted two techniques for blocking resistance: innocent IP mixing and collaborative spy detection. In innocent IP mixing, we include a number of IP addresses, which are unrelated to VPN Gate, in the Server List. For instance, we include vitally important servers (e.g., Windows Update servers). This technique forces a censorship authority to remove innocent IP addresses from the Server List before adding addresses to the firewall blocking list. The second technique, collaborative spy detection, seeks probing activities from censorship authority's computers, called *spies*. In this technique all the volunteer VPN servers work together to create a source IP address list of spies, called the *Spy List*, and they ignore probing packets from spies. This technique makes the authority unable to distinguish between the IP addresses of active VPN servers and innocent IP addresses or those of inactive VPN servers.

The VPN Gate system consists of instances of the VPN Gate Server software, an optional application, the VPN Gate Client software, and a central List Server. Volunteers can easily install and execute VPN Gate Server. For instance, volunteers don't have to configure Network Address Translation (NAT) boxes to open TCP/UDP ports. Users can connect to VPN Gate Server with a Secure Sockets Layer (SSL)-VPN protocol by using VPN Gate Client. Users can also connect to a VPN server with the L2TP/IPsec, OpenVPN, and MS-SSTP protocols by using the built-in, OS-provided VPN clients on PCs and smartphones. As for the third piece of the system, our research group runs the VPN Gate List Server which accepts registration from volunteer servers, generates the Server List, and distributes it to users.

We launched VPN Gate on March 8, 2013. On August 29, we had about 3,000 active VPN Gate servers. This number is comparable to the number of Tor relay nodes. On the same day we had 464,000 connections to the VPN Gate servers. These connections were from 88,000 unique source IP addresses.

VPN Gate has blocking resistance against the GFW. Shortly after we started the service, the GFW authority added the IP addresses of all the volunteer servers into the GFW blocking list. On April 4, the GFW blocked 81% of all volunteers, so only 19% of active volunteers were reachable from China. Hence, we implemented the innocent IP mixing and collaborative spy detection techniques. As a result, we achieved 50% reachability from China on April 26, and 75% on May 9. Moreover, around 40% of our volunteers' IP addresses changed every day. The GFW could not catch up to our increasing number of volunteers and their changing IP addresses. VPN Gate has thus provided stable reachability for Chinese users. At the end of August, 2013, we have about 45,000 daily connections from 9,000 unique IP addresses in China, while Tor had an estimated 3,000 users from China.

VPN Gate is a system for bypassing censorship. It is not an anonymizer. Unlike Tor, VPN Gate volunteer servers record packet logs. VPN Gate also has no multi-hop relaying function.

2. Related Work

VPN Gate organizes VPN servers provided by volunteers. This method is similar to that of the well-known anonymizer Tor [7]. Since communications in Tor are relayed by three Tor nodes to achieve anonymity, they are slow.

Tor nodes are classified into two types: public *relays* and non-public *bridges*. It is easy for censorship authorities to block the public relays. Users behind censorship firewalls must find non-public bridges through web sites, email, and other means of contact. Although bridges are not public, censorship authorities can probe and block them [18, 19]. Using obfsproxy, it is possible to obfuscate the network traffic exchanged between Tor clients and bridges [17]. However, Tor bridges currently have no blocking resistance against such probing activities.

Unlike Tor, VPN Gate focuses on bypassing censorship firewalls and does not provide anonymity. Since communications in VPN Gate are relayed by a single VPN server, they are much faster than in Tor. To use VPN Gate, users behind censorship firewalls must get a list of VPN servers through web sites, email, and so forth. Unlike Tor, VPN Gate also includes innocent IP addresses in a list of VPN servers. We describe this aspect in Section 4.2. Furthermore, VPN Gate has a mechanism making it harder for censorship authorities to probe VPN servers. We describe this aspect in Section 4.3.

It is not trivial to run Tor relay and bridge nodes. Rbox-Tor helps volunteers run Tor nodes by using virtual machines [16]. VPN Gate also helps volunteers run VPN servers by a variety of techniques, including Network Address Translation (NAT) traversal capability. We describe this capability in Section 5.2.

VPN Gate maintains the list of VPN servers in a centralized server. This mechanism is similar to a *tracker* in BitTorrent [6]. It is easy for censorship authorities to block communications to a tracker. To avoid using centralized trackers, BitTorrent introduced a distributed hash table (DHT), implemented in the Mainline and Azureus DHTs [2, 4]. We chose a centralized server instead of a DHT for two reasons. First, we have to return a different partial server list for each client. Second, we have to accumulate all information from all active VPN servers in a central server to analyze unusual usages. We describe these design choices in Section 4.3.

Many researchers are working on censorship-resistant systems [3, 5, 8, 9, 13, 14, 20]. These systems either are Web-access-specific ones or require modifying existing protocol stacks. Here, in contrast, we describe a VPN-based censorship-resistant system that allows using arbitrary protocols without modifying an existing protocol stack.

Many free and commercial VPN services are also used to bypass censorship firewalls [10]. Since most such services use a set of centralized VPN servers with fixed IP addresses, censorship authorities can easily block these services with firewalls. Some VPN services do have a decentralized or peer-to-peer (P2P) architecture [11, 15]. There have been no published reports or results, however, on bypassing methods.

Finally, our collaborative spy detection technique is similar to collaborative intrusion detection [22]. In this paper, we show a specific method to protect VPN servers.

3. VPN Gate Overview

Figure 1 shows an overview of VPN Gate. A volunteer downloads the VPN Gate Server software and runs it on a PC. While VPN Gate Server is running, it registers itself to the VPN Gate List Server. This server maintains the Server List, a list of IP addresses for active VPN Gate Server instances.

Assume here that a VPN Gate user lives behind a censorship firewall and cannot access blocked servers in foreign countries. The user first accesses a web page for the VPN Gate List Server to get a list of VPN servers. To avoid discovery of all VPN servers by censorship authorities, VPN Gate List Server returns only a small part of the entire Server List. Next, the user chooses a VPN Gate Server instance from the partial list. Finally, the user connects his/her PC to the chosen server by using either a native VPN client on the PC or dedicated VPN client

software, called VPN Gate Client. Once the VPN connection is established, the VPN server relays all the user's communications to the Internet.

3.1. Hosting VPN Gate Server as a volunteer

As described above, a volunteer installs and runs VPN Gate Server on a PC. At this time, the volunteer does not need to show his/her name, address, or any other personal information. While VPN Gate Server is running, it waits for new VPN connections from users. It accepts four VPN protocols: L2TP/IPsec, OpenVPN, MS-SSTP, and SoftEther VPN Protocol.

While VPN Gate Server is running, it periodically checks the type of Internet connection on the PC. If the PC is behind a NAT box, VPN Gate Server attempts to open a port via Universal Plug and Play (UPnP) or UDP hole punching. With the recognized type of Internet connection, VPN Gate Server registers itself to VPN Gate List Server, which we describe in Section 3.3.

3.2. Connecting to VPN Gate as a user

A VPN Gate user accesses the web site of the VPN Gate List Server and obtains part of the Server List. This contains information about volunteer servers, including IP addresses and port numbers, geographic locations, line quality parameters such as bandwidth and delay, numbers of current VPN connections, and numbers of cumulative VPN connections. The user thus chooses a preferred VPN server from the subset of the Server List.

Since censorship authorities can easily discover the web site of the VPN Gate List Server, a user in a country subject to censorship likely cannot access the web site directly. Such a user can instead access it via an HTTP relay site provided by VPN Gate Server. Section 4.5 gives the details of this mechanism.

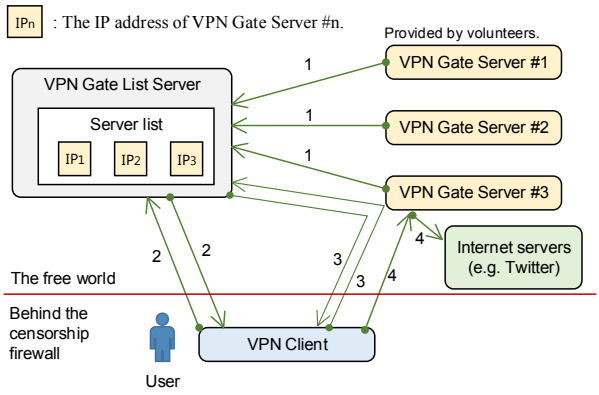
Next, the user can establish a VPN connection by using one of the following methods:

1. Using a built-in VPN client in the operating system (OS).

The user inputs the IP address of the chosen VPN server in the configuration window of the L2TP/IPsec or MS-SSTP VPN client. In this window, the user also fills in the user name and password fields with fixed values, "vpn" and "vpn". The advantage of this method is that it does not require installing any software.

2. Using OpenVPN Client.

The user installs the OpenVPN Client software once. Then, he/she downloads an OpenVPN connection setting file (.ovpn file) from the VPN Gate List Server web site and runs OpenVPN Client with the same setting file each time when he/she connects to VPN Gate.



1. Register itself to the VPN Gate List Server.
2. Get the server list directly.
3. Get the server list with the Indirect Server List Transfer Protocol.
4. Access to Internet servers through the VPN server.

Figure 1. Overview of VPN Gate.

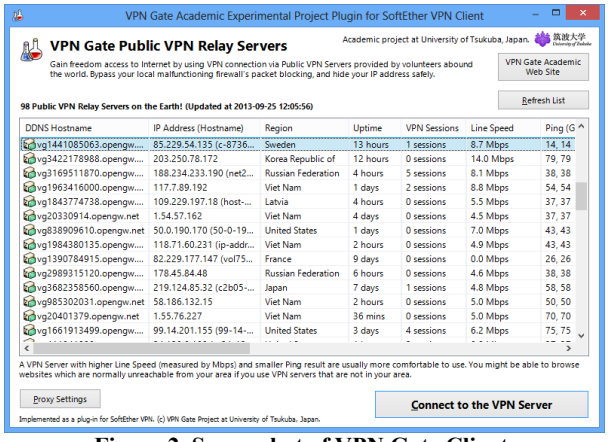


Figure 2. Screenshot of VPN Gate Client.

3. Using VPN Gate Client.

The user installs the VPN Gate Client software once and runs it each time he/she needs a VPN connection. VPN Gate Client displays the user's portion of the Server List, as shown in Figure 2, and he/she chooses a server for connection. The advantage of this method is that it is easy, and it supports the Indirect Server List Transfer Protocol, which we describe in Section 4.4.

3.3. VPN Gate List Server

The VPN Gate List Server software accepts registrations from active VPN Gate Server instances and monitors these servers' statuses. When VPN Gate List Server receives a server list request from a client, it returns a small part of the Server List. In addition, VPN Gate List Server implements the firewall resistance system described in Section 4.

4. Firewall Resistance System

The firewall resistance system in VPN Gate achieves blocking resistance to censorship firewalls. This system is implemented in both VPN Gate Server and VPN Gate List Server. In this section, we first briefly describe the blocking methods of the Chinese censorship firewall. After that, we describe our blocking resistance techniques. The two key techniques are *innocent IP mixing* and *collaborative spy detection*.

4.1. Blocking methods used in the Great Firewall of China

We set our goal in designing the system to achieve blocking resistance to the Chinese GFW. To do so, we studied the GFW's blocking methods. According to various reports [1, 5, 21], the GFW exists at borders between Chinese internet service providers (ISPs) and overseas ISPs, and it can block all IP packets sent to IP addresses on the blocking list. The GFW authority must maintain a blocking list of IP addresses. It exploits both human resources and automated scanners to maintain the blocking list. For instance, the GFW authority performs scanning to detect hidden Tor nodes [19].

4.2. Innocent IP mixing

The first technique for achieving blocking resistance in VPN Gate is innocent IP mixing, illustrated in Figure 3. In this technique, we include a number of fake IP addresses, called *innocent IP addresses*, when VPN Gate List Server returns a list of VPN servers to a user. Innocent IP addresses are chosen from among addresses unrelated to VPN Gate, and they should be addresses of vitally important hosts in the Internet. Examples of good innocent IP addresses include DNS root servers, top-level-domain DNS servers, Windows Update servers, and popular email servers. After a censorship authority notices innocent IP mixing, it cannot automatically add all obtained IP addresses from the Server List to its firewall blocking list. Instead, the authority has to verify whether each of the obtained IP addresses is the real IP address of a VPN Gate Server. We do not have to mix innocent IP addresses every day, all the time; it is sufficient to mix in a small number of innocent IP addresses occasionally to keep the authority's attention.

As a disclaimer, we have included the following warning sentence on the web site for the VPN Gate List Server: "This server list might contain wrong IP addresses, and authorities should not use these IP addresses for firewall blocking lists."

Innocent IP mixing does not affect regular users of VPN Gate. If a user occasionally chooses an innocent IP address, he/she will just get a connection error. The user can then simply try another IP address from the Server List.

Innocent IP mixing does not cause distributed denial of service (DDoS) attacks on innocent servers. Suppose that we have 100 million users each day, and we mix in one innocent IP address for every 1,000 real VPN servers. If each user chooses a target VPN server randomly from the list, the server for an innocent IP address will receive an expected 100,000,000 / 1,000 = 100,000 connection requests each day. If we assume five retry packets per connection request, the server will receive 7 useless packets per second. We believe that such a small number of useless packets is harmless to Internet servers of the present day.

In practice, a typical user does not choose a VPN server randomly but tries servers from top to bottom in the list. A user's list typically has 100 VPN servers, and we can put the innocent IP address in the middle of the list. Since the user will most likely succeed in connecting or stop trying before reaching the innocent IP address, the corresponding server will never receive any connection requests.

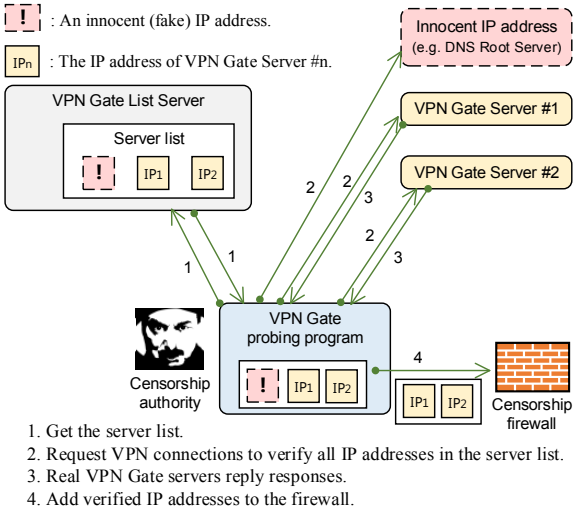


Figure 3. Innocent IP mixing.

4.3. Collaborative spy detection

The second technique for achieving blocking resistance in VPN Gate is collaborative spy detection. This technique detects probing activities from the computers of a censorship authority, called *spies*. To find spies, all instances of VPN Gate Server work together and build a source IP address list of spies, called a *Spy List*. As illustrated in Figure 4, the servers then ignore probing packets from spies in the Spy List. The Spy List contains both IP addresses and ranges of IP addresses. This technique prevents censorship authorities from distinguishing whether VPN Gate Server is running on a specific IP address.

Collaboration is vital to detecting spies in VPN Gate. Since a spy establishes a VPN connection with regular VPN protocol procedures, a single VPN Gate Server instance cannot distinguish between a spy and a regular client. When a single VPN Gate Server instance does find a spy by recognizing the unusual behavior of the spy client, it is too late because the spy has already succeeded in discovering the VPN server by that time. Therefore, the VPN server must distinguish whether a client is a spy before sending its first response to the client. This is impossible for a single VPN Gate Server instance.

To solve this problem, all VPN Gate Servers work together to detect spies, share the Spy List, and deny connections from clients in the Spy List. The process of generating the Spy List consists of the following two procedures:

1. Procedure in VPN Gate Server

VPN Gate Server records VPN connection logs, which we classify into three types: *complete calls*, *incomplete calls*, and *tiny calls*. A complete call means a VPN connection that is normally established between a client and a server, where the amount of actual data transfer exceeds a threshold. An incomplete call is a VPN connection that is disconnected either by a client before a negotiation completes or because of a protocol error. A tiny call is a VPN connection that has either a very short duration or a small amount of data transfer. VPN Gate Server records all these calls with metadata such as source IP addresses, times, data transfer amounts, and durations. Each VPN Gate Server instance regularly sends these logs to the VPN Gate List Server.

2. Procedure in VPN Gate List Server

VPN Gate List Server aggregates the logs from all VPN Gate Server instances in order to find spies by using the following conditions:

I. If many VPN servers received incomplete calls from a specific IP address or a specific range of IP addresses, we mark the address or range as a spy.

II. If many VPN servers received tiny calls from a specific IP address or a specific range of IP addresses, we mark the address or range as a spy.

VPN Gate List Server performs this procedure periodically and distributes the generated Spy List to all VPN servers. We reduce the size of the Spy List by aggregating multiple IP addresses into a range of IP addresses in a /24 block. We apply this aggregation technique when the number of IP addresses in a block exceeds a threshold, which varies according to the frequency of accesses and other conditions. For example, the threshold for Chinese IP addresses is smaller than that for United States IP addresses.

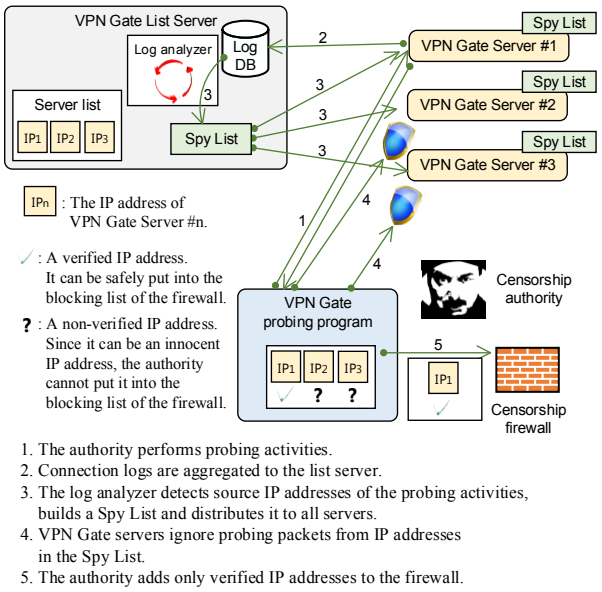


Figure 4. Collaborative spy detection.

4.4. Distributing server lists to users

Anti-censorship systems that use relay servers face the relay server discovery problem: how can clients discover relay servers without having a censorship authority also discover and block these servers [9]? To address this problem, VPN Gate applies several techniques.

First, we use a technique called *keyspace hopping* [9]. In keyspace hopping, each client pseudorandomly uses a unique set of servers, just as a wireless node uses frequency hopping to resist jamming. This technique ensures that each client discovers only a small fraction of the total number of VPN Gate servers. Furthermore, we use the network address of a client as the seed of a pseudorandom number generator in keyspace hopping. This method forces the censorship authority to have a large number of IP addresses in order to collect the IP addresses of all the VPN Gate servers.

The second technique is to introduce the *Indirect Server List Transfer Protocol*. When a user in a country subject to censorship tries to get a fresh server list through VPN Gate Client, a firewall will likely block the communication with VPN Gate List Server. We thus implemented the Indirect Server List Transfer Protocol to solve this problem. This protocol allows VPN Gate Client to get a fresh server list via an intermediate server. The intermediate server is a VPN Gate Server instance known by the client. Note that a server list transferred with this protocol is digitally signed to prevent modification by the intermediate server.

The third technique is dynamic generation of initial server lists. It is useful for a first-time user of VPN Gate Client to have a fresh initial list of VPN Gate servers. To

achieve this, our download Web server dynamically generates a fresh initial server list for each destination and includes it in the installation package of VPN Gate Client. We generate this initial list by applying keyspace hopping, the first technique mentioned above. We also mix innocent IP addresses into the initial list.

On August 19, 2013, our VPN Gate List Server accepted about 379,000 indirect server list transfer requests, representing 23.2% of the total of about 1,630,000 user requests on that date.

4.5. HTTP relay function and Daily Mirror URL Mail Service

It is easy for a censorship authority to block our download web server and the web site of the VPN Gate List Server. To overcome this problem, we implement an HTTP relay function in VPN Gate Server. This function gives users the chance to download VPN Gate Client at the time of first use. This function also provides access to the VPN Gate Server List web site for those who use built-in VPN clients.

As we described in Section 4.1, censorship firewalls can detect and block our HTTP relay function by keyword inspection. To make this inspection task difficult, we respond with gzip-compressed HTTP contents.

VPN Gate also provides a Daily Mirror URL Mail Subscription service. This service emails the latest URL list to subscribers every day. Each list contains the URLs of a small number of active VPN Gate Server instances that enables the HTTP relay function. These URLs are suitable for distribution via online and offline social networks in countries subject to censorship. On September 13, 2013, we had 11,000 subscribers to this mail service. Through keyspace hopping, we disclose only a small fraction of VPN Gate servers in this service. When a subscriber signs up the service, we record the IP address of the subscriber's Web browser and use it as the seed of a pseudorandom number generator for keyspace hopping.

5. Implementation

In this section, we describe the implementation of VPN Gate Server, VPN Gate Client, and VPN Gate List Server.

5.1. Implementation of VPN Gate Server

We have implemented VPN Gate Server as an application program for Windows. The program code is based on SoftEther VPN Server, which is our free VPN server program[1]. VPN Gate Server supports the following four VPN protocols. VPN Gate Server treats all VPN clients using any of these VPN protocols equally.

1. L2TP/IPsec
2. OpenVPN protocol
3. MS-SSTP
4. SoftEther VPN protocol

The SoftEther VPN protocol implements Ethernet over SSL on TCP or UDP. It has affinity with most firewalls. It requires VPN Gate users to install the specific VPN Gate Client in their devices. Unlike MS-SSTP, this VPN protocol is usable in UDP-only environments.

We have also implemented an *Internet sharing function* in VPN Gate Server. This function allows sharing of a single outgoing IP address for the server while allocating a different private address for each VPN client.

5.2. Running VPN Gate Server behind a NAT box

We assume that the PCs of most volunteers running VPN Gate Server are behind NAT boxes. To increase the number of available volunteer servers, it is necessary to make VPN servers reachable from the Internet even when they are behind NAT boxes. Therefore we implemented an automatic port-opening function in VPN Gate Server, via UPnP and UDP hole punching. This function also works in the intermediate servers for the Indirect Server List Transfer Protocol described in Section 4.4.

To increase NAT affinity, we also added UDP support to our SoftEther VPN protocol. The previous SoftEther VPN Protocol was based on SSL and worked only with TCP. To extend it to work with UDP, as well, we designed and implemented a Reliable UDP (RUDP) protocol that has a retransmission control mechanism like that of TCP.

5.3. Status monitoring of VPN servers

VPN Gate List Server performs status checking of all registered VPN servers. It executes this checking not only the first time it registers a VPN server but also periodically thereafter. After VPN Gate List Server verifies that a VPN server is functional, it adds an entry for the VPN server into the Server List.

In addition to functional checking, VPN Gate List Server collects the Internet connection qualities of registered VPN servers. To measure communication delays of the last one mile network, each VPN server sends ICMP echo requests to the Google Public DNS server (8.8.8.8)[2]. To measure communication bandwidths, each VPN server runs a TCP speed test tool with our speed test servers. The VPN servers then report these results to

[1] http://www.softether.org/

[2] Google Public DNS server is located around the world. https://developers.google.com/speed/public-dns/faq

the VPN Gate List Server. Users can view these results on the List Server's web site, thus enabling them to choose a good VPN server instance with a low-delay, high-bandwidth Internet connection.

5.4. VPN connection logs and packet logs

Each VPN server records VPN connection logs when a VPN client establishes a tunnel, and when the user disconnects the tunnel. Each VPN server also records packet logs that include not only TCP and UDP headers but also payloads. A volunteer can read these logs and know the source IP addresses of VPN clients. When a criminal uses a VPN server, the owner of the server may pass these logs to a public authority. The VPN servers also transmit the VPN connection logs to the VPN Gate List Server, which uses them for collaborative spy detection, as described in Section 4.3.

Each VPN server records VPN connection logs when a VPN client establishes a tunnel, and when the user disconnects the tunnel. The VPN servers transmit the VPN connection logs to the VPN Gate List Server, which uses them for collaborative spy detection, as described in Section 4.3.

In addition to connection logs, each VPN server also records the following minimum information for packet logging.

1. TCP Packets

The VPN server records the IP and TCP headers of SYN, SYN+ACK, and ACK packets. It records no payloads other than HTTP request headers.

2. UDP Packets

The VPN server records the IP and UDP headers for DHCP and IPsec/UDP and OpenVPN/UDP initiate packets. We record the headers because these VPN protocols can be used to hide client IP addresses. The VPN server does not record payloads.

Since we do not want VPN Gate to be used as an anonymizer, we intentionally designed it to record these logs so as to prevent abuse by criminals while maintaining the privacy of normal users. Without a packets logging function, criminals could abuse VPN Gate to hide their client IP addresses. When a criminal uses a VPN server, the owner can pass packets logs to a law enforcement agency. The "VPN Gate Anti-Abuse Policy" on the web site clearly states that each VPN Gate server records packet logs in order to prevent such abuse.

5.5. Implementation of VPN Gate Client

We implemented VPN Gate Client as an extension of SoftEther VPN Client, a VPN client program for establishing VPN connections to SoftEther VPN Server instances. SoftEther VPN Client consists of a virtual network adapter kernel-mode driver, a VPN processing module, and a GUI. We modified the GUI by adding a window to show a list of VPN servers (Figure 2). We also implemented a client module for the Indirect Server List Transfer Protocol.

Furthermore, we include the VPN Gate Server function in the VPN Gate Client program. We took this idea from P2P file sharing applications such as BitTorrent. In P2P file sharing, each client also contributes to the network as a server. The VPN Gate Server function in VPN Gate Client is disabled by default. A user who wants to be a volunteer can enable this function manually. This function is also automatically disabled while using VPN Gate Client to connect to another VPN server.

5.6. Dynamic generation of VPN Gate Client package

VPN Gate users download VPN Gate Client from the download server or relays, as described in Sections 4.4 and 4.5. Every time the download server responds to a user, it generates a new ZIP package. The fixed content of each ZIP package consists of the binary of VPN Gate Client. The package also includes variable content in the form of an initial server list. The ZIP file also includes a file with a random filename and random data at the head, for blocking resistance to censorship firewalls. This technique eliminates characteristics of TCP streams for VPN Gate Client downloading traffic.

If the download server generated a temporary ZIP file each time it received a request, this would increase the disk I/O load at the server. To eliminate this load, we implemented a lightweight, in-memory ZIP generator in the download server. For each downloading user, the ZIP generator consumes only a fixed small amount of a buffer in the server's memory.

6. Experiences

We initiated the VPN Gate web site and released the programs VPN Gate Server and Client on March 8, 2013. In this section, we demonstrate achievement of our purposes described in Section 1, by evaluating our experiences for six months after the release of VPN Gate.

6.1. Statistics of users and volunteers

Figure 5 shows the variation in the number of daily VPN connections, and Figure 6 shows the number of unique IP addresses for VPN clients on a daily basis. For instance, we had about 464,000 connections from 88,000 unique IP addresses on August 29. On this day, there

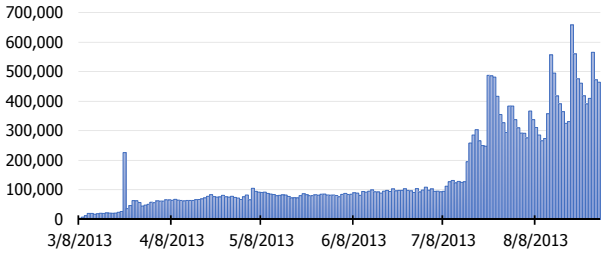


Figure 5. Number of daily VPN connections.

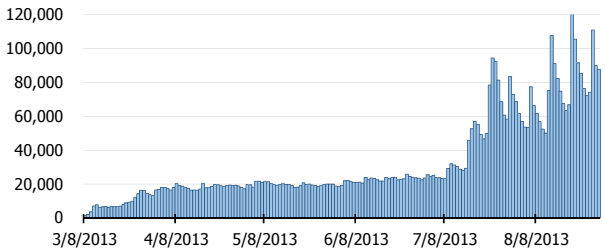


Figure 6. Number of daily unique IP addresses for VPN clients.

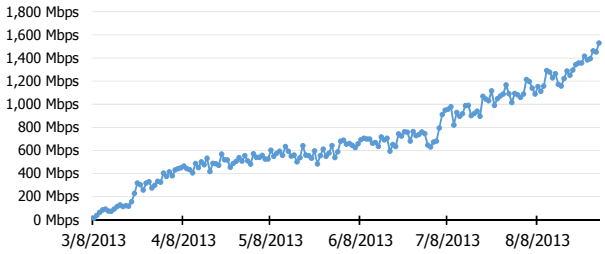


Figure 7. Daily total bandwidth for VPN connections.

were 5.3 VPN connections on average from a unique source IP address. Figure 7 shows the variation in the total bandwidth. We had a total of 1.6 Gbps on August 29. The total bandwidth steadily increased because the number of active volunteers increased along with the number of users.

Table 1 lists the top ten countries in terms of the number of client VPN connections on August 30, 2013. Table 2 lists the top ten countries in terms of the amount of client VPN traffic through August 30, 2013. Each of these tables includes China, Thailand, and Iran, countries that have censorship firewalls. VPN Gate thus helped users in these countries to bypass the firewalls. Table 2 also includes Korea, the United States, Japan, and Taiwan. Since high-speed home Internet lines are popular in Korea, it ranked first in total data transfer but only seventh in the number of VPN connections. On the other hand, while Thailand and Iran had large numbers of VPN connections, they did not yield large amounts of transferred data. This means that most users in these countries have low-speed Internet lines. These results show that the bottlenecks are mostly on the client side, and not on the server side.

Table 1. Numbers of VPN connections at different client locations.

Ranking	Location	Number of VPN connections	Percentage
1	Taiwan	7,253,003	27%
2	China	3,974,954	15%
3	Thailand	3,841,947	14%
4	Iran	2,281,446	8%
5	Japan	1,768,716	6%
6	Vietnam	1,399,833	5%
7	Korea	1,373,906	5%
8	Indonesia	742,640	3%
9	United States	589,148	2%
10	Hong Kong	466,265	2%
	190 other locations	3,536,359	13%
	Total	27,228,217	100%

Table 2. Total transferred data at different client locations.

Ranking	Location	Transferred data amount	Percentage
1	Korea	460.0 TB	35%
2	China	193.4 TB	15%
3	United States	145.7 TB	11%
4	Japan	111.1 TB	8%
5	Taiwan	90.4 TB	7%
6	Iran	45.1 TB	3%
7	Hong Kong	28.2 TB	2%
8	Malaysia	26.3 TB	2%
9	Vietnam	25.8 TB	2%
10	France	18.0 TB	1%
	190 other locations	187.1 TB	14%
	Total	1,331.1 TB	100%

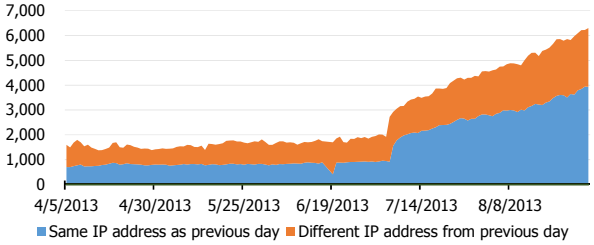


Figure 8. Numbers of VPN servers with changed and unchanged IP addresses.

We gained a total of 16,523 volunteer servers from 127 countries or regions over the course of 175 days. These servers have used 108,633 unique IP addresses.

Figure 8 shows the numbers of VPN servers with changed and unchanged IP addresses from April 5 to August 30. The lower blue area corresponds to servers whose IP addresses were unchanged from the previous day, while the upper orange area corresponds to servers whose IP addresses did change. For instance, on August 30 we had 3,935 unchanged IP addresses and 2,363 changed IP addresses. This means that 38% of the VPN servers had a different IP address from that of the previous day. On average, 40% of VPN servers had new IP addresses every day. This changing of IP addresses contributed to increasing the reachability from countries subject to censorship.

Table 3. Locations of VPN Gate Server instances on August 30, 2013.

Ranking	Location	Number of volunteer servers	Percentage
1	Korea	841	30%
2	Japan	637	23%
3	Vietnam	444	16%
4	United States	181	6%
5	Russia	119	4%
6	France	57	2%
7	Thailand	51	2%
8	United Kingdom	41	1%
9	Indonesia	38	1%
10	Canada	29	1%
	66 other locations	362	13%
	Total	2,800	100%

Table 4. Active VPN servers on August 30, 2013.

	Volunteers	Percentage
Direct connection (non-NAT)	3,884	27%
NAT (UPnP compatible)	7,384	52%
NAT (UDP hole punching compatible)	3,006	21%
Total	14,274	100%

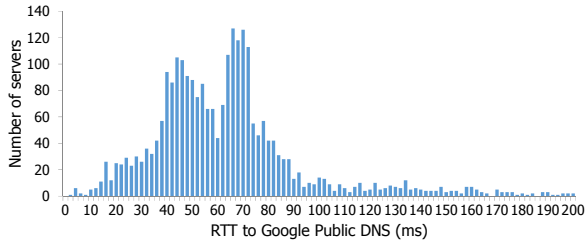


Figure 9. Round-trip time between volunteer servers and Google Public DNS on August 30, 2013.

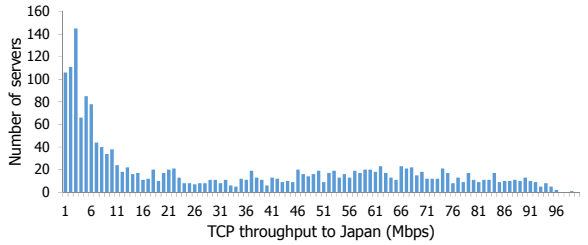


Figure 10. TCP throughput to the Japan server.

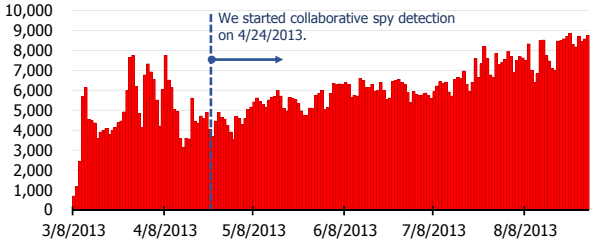


Figure 11. Number of daily unique IP addresses for VPN clients in China.

Table 3 shows the geographical distribution of the 2,800 volunteer servers running at 15:00 (GMT) on August 30, 2013. We resolved the location of each volunteer by using IP address allocation information. We found that 77% of the volunteers were from five countries: Korea, Japan, Vietnam, United States, and Russia.

We examined the quality of the Internet connections provided by the VPN servers. Figure 9 shows the round-trip times (RTTs) between each VPN server and the Google Public DNS server (IP address: 8.8.8.8), on August 30, 2013. Since Google Public DNS Servers are located worldwide, the RTT implies the quality of the last-mile line to the ISP of each VPN server. Most of the VPN servers had RTT values of 100ms or less. This means that most of the VPN servers are connected to the Internet with pretty good lines. Figure 10 shows the TCP bandwidth between each VPN server and our speed test server in Japan. More than 50% of the VPN servers had bandwidth of 5Mbps or faster. We estimated the total available bandwidth as 70Gbps. This is much larger than the used bandwidth of 1.6Gbps shown in Figure 7.

Table 4 lists the types of Internet connections used by the VPN servers on August 30, 2013. The data shows that 72.8% of VPN servers were behind NAT boxes. This means that the NAT affinity function described in Section 5.2 worked well.

6.2. Users from China

The Chinese GFW authority began to block the IP address of the VPN Gate List Server on March 12, 2013. It also began attempting to block all IP addresses of VPN servers listed on the web site for the List Server. Despite this, Figure 11 shows that the number of users from China increased continuously. This figure does not include spies detected by our collaborative spy detection. On August 29, the number of unique IP addresses for clients in China was 8,000, and this occupied 10% of all unique source IP addresses for VPN connections. These results show that our blocking resistance techniques are working effectively.

We measured the blocking rate of our VPN servers by the GFW since March 22. Early on, as shown in Figure 12, the GFW authority succeeded in blocking our VPN servers effectively. At that time only 30% of VPN servers were reachable from China. After we started innocent IP mixing and collaborative spy detection, however, the blocking rate decreased. On June 19, 78.5% of VPN servers were reachable from China. At the end of August, we typically had 60-70% of servers reachable from China. On August 8, the rate of servers blocked by the GFW suddenly decreased. We suppose that this was due to technical problems in the GFW.

In summary, we have achieved strong blocking resistance to China's GFW with VPN Gate Server. This

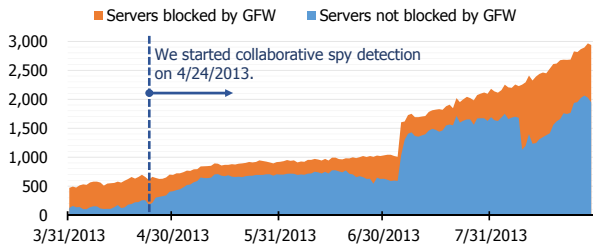


Figure 12. Numbers of VPN servers blocked and not blocked by the GFW.

was achieved by applying two techniques, namely, innocent IP mixing and collaborative spy detection. Rapid changing of server IP addresses has also contributed to this result.

6.3. Cat-and-mouse game with the Great Firewall authority

We played a cat-and-mouse game with the Chinese GFW authority until we implemented innocent IP mixing and collaborative spy detection. Here, we outline the history of this game.

March 8: We launched VPN Gate.
We initiated the web site and released the server and client programs on a Friday. Many Chinese users found VPN Gate within the first four days before blocking. On March 11, we had 5,663 unique IP addresses for clients from China. We assumed that the officers of the GFW authority did nothing on Saturday and Sunday.

March 11: GFW blocked VPN Gate List Server.
The GFW authority blocked the IP address of the VPN Gate List Server. Users in China could no longer visit the VPN Gate web site or download VPN Gate Client after this time. Some users in China began to spread URLs for the relay sites described in Section 4.5 by using domestic Chinese SNS web sites (e.g., Weibo). The relay sites helped Chinese users to visit the VPN Gate List Server web site and download VPN Gate Client. VPN Gate Client users could continue to use it with the support of the Indirect Server List Transfer Protocol.

March 12: GFW started automatic blocking.
The authority started to get the list of active VPN servers from the VPN Gate List Server periodically, and it started adding *all* IP addresses in the list to the GFW. On March 12 and 13, the authority performed this task twice a day. After March 14, the authority performed this task several times a day. We assume that the authority implemented an automated tool for this task. This response revealed that the GFW authority can discover an anti-firewall service and develop a blocking tool for it within only four days after the service starts.

March 13: We discovered a single spy IP address.
We set up 32 servers that did not run VPN Gate Server. We also added code in VPN Gate List Server to mix different portions of the IP addresses of these servers according to each request. We used these IP addresses as steganographic codes. For example, if the source IP address was 1.2.3.4, we mixed IP addresses #7, #14, #20, #21, and #27 into the list sent to the requester. Approximately 30 minutes later, the GFW blocked some of the steganographic IP addresses. We could then calculate that the IP address of the spy was 210.72.128.200. According to Whois, this is an IP address operated by China Science and Technology Network (CSTNET), an institution of the Chinese Academy of Sciences. We confirmed that the authority used this IP address to get our VPN server list and blocked this address from accessing the VPN Gate List Server web site. We also found that the user agent value of the spy program was "Python-urllib". We thus assumed that the authority wrote the spy program in Python.

March 14: GFW started getting the VPN server list from oversea cloud servers.
After we had blocked the source IP address of the authority, it started using Amazon EC2 and Gorilla Servers to get VPN server lists. We found that the user agent value was still "Python-urllib", as shown in Figure 13. The authority obtained the VPN server lists at fixed intervals. We could thus distinguish spies from regular users, but since it was easy for the authority to vary the user agent value and interval, we decided not to use these characteristics for detecting spies. The authority obtained many IP addresses of our VPN servers and put them in the GFW blocking list. After this automated process began, approximately 80% of all VPN servers became unreachable from China.

March 14: We started innocent IP mixing.
We began to mix unrelated IP addresses at the University of Tsukuba into the VPN server lists. We observed that these IP addresses became unreachable from China within 30 minutes. We tested this several times for around four hours. The GFW always blocked our newly mixed IP addresses within 30 minutes or less. This means that the GFW authority trusted our VPN server list at that time, and they did not verify the IP addresses in the list before blocking them. In other words, we had power to control the GFW for a short time.

March 16: GFW suspended using the automated tool.
The authority noticed that the VPN server lists included unrelated IP addresses. It thus suspended using the automated tool for inserting IP addresses in the VPN server

ID	Access Date	Client FQDN	URL	User Agent
3312453	3/23/13 7:40 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3312674	3/23/13 7:41 PM	ec2-50-16-163-135.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3313273	3/23/13 7:45 PM	198-136-27-242.static.gorillaservers.com	http://www.vpngate.net/	Python-urllib/1.17
3313385	3/23/13 7:45 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3313579	3/23/13 7:46 PM	ec2-50-16-163-135.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3314469	3/23/13 7:50 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3314708	3/23/13 7:51 PM	ec2-50-16-163-135.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3315395	3/23/13 7:55 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3315642	3/23/13 7:56 PM	ec2-50-16-163-135.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3316250	3/23/13 8:00 PM	198-136-27-242.static.gorillaservers.com	http://www.vpngate.net/	Python-urllib/1.17
3316252	3/23/13 8:00 PM	198-136-27-242.static.gorillaservers.com	http://www.vpngate.net/en/	Python-urllib/1.17
3316383	3/23/13 8:00 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3316570	3/23/13 8:01 PM	ec2-50-16-163-135.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3317306	3/23/13 8:05 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3317533	3/23/13 8:07 PM	ec2-50-16-163-135.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3318339	3/23/13 8:10 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3318553	3/23/13 8:12 PM	ec2-50-16-163-135.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3319069	3/23/13 8:15 PM	198-136-27-242.static.gorillaservers.com	http://www.vpngate.net/	Python-urllib/1.17
3319072	3/23/13 8:15 PM	198-136-27-242.static.gorillaservers.com	http://www.vpngate.net/en/	Python-urllib/1.17
3319236	3/23/13 8:15 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3319480	3/23/13 8:17 PM	ec2-50-16-163-135.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3320192	3/23/13 8:20 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3320439	3/23/13 8:22 PM	ec2-50-16-163-135.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17
3321185	3/23/13 8:26 PM	ec2-23-20-4-19.compute-1.amazonaws.com	http://www.vpngate.net/en/	Python-urllib/1.17

Figure 13. Spying from Amazon EC2 and Gorilla Servers by the GFW authority.

list into the GFW blocking list. The authority also discharged all blocking of VPN Gate servers. For four days after that, no VPN servers were blocked by the GFW.

March 20: GFW started verifying IP addresses. The authority started verifying IP addresses before inserting them into the GFW blocking list.

April 24: We started collaborative spy detection. We started collaborative spy detection as described in Section 4.3.

6.4. Scalability

As the number of volunteers increases, the total available bandwidth increases. Therefore, the scalability of VPN Gate is bounded by the VPN Gate List Server.

The VPN Gate List Server instance currently consists of three web servers, a database server, a status monitoring server, and a log analysis server. These servers are connected to the Internet via the campus network at our university. Only the web servers receive requests from VPN servers and users. Since these servers execute the same web application, we can easily scale-out their performance.

We use Microsoft SQL Server for our database server, which runs on a PC with an Intel Xeon E3-1230 3.2-GHz processor (Fujitsu PRIMERGY TX100 S3). This PC has 1.0 MB of L2 cache, 32 GB of main memory, and two SSD drives. The load of the database server is lower than the loads of the web servers. The CPU usage of the database server is approximately 5%, while the disk I/O bandwidth is approximately 1.3 MB/s. We estimate that the database server could handle up to 10 times the current load without upgrading the hardware. When this database server becomes overloaded, we can divide VPN servers and clients into several groups and allocate a database server for each group. We think that each group could perform collaborative spy detection independently. This division would also increase availability.

We have found that we can perform status monitoring on 3,000 VPN servers within 10 minutes. We can easily divide this task and allocate subtasks to multiple servers.

6.5. Comparison to Tor

On August 29, 2013, the number of VPN servers was 3,000, which was comparable to the number of Tor nodes. Tor had 4,000 listed relay nodes and 2,000 hidden bridge nodes. We hope that the number of VPN servers in VPN Gate will exceed the number of Tor nodes because we added 500 new servers each in both July and August, 2013.

The number of Chinese users of VPN Gate was larger than that of Tor. At the end of August 2013, we had 9,000 daily unique IP addresses from China, while Tor had an estimated 3,000 daily users from China according to the Tor metrics site[3]. We achieved this result because we focused on bypassing firewalls and implementing collaborative spy detection. In contrast, it is hard for Tor to achieve such collaboration among nodes.

VPN Gate has another advantage over Tor. Since VPN Gate provides a VPN tunnel for IP, a user can use any TCP or UDP application through VPN Gate without having to modify the application or set proxies.

6.6. Problems and discussion

Criminals might use VPN Gate to hide their IP addresses. We can repress such abuse through logging as described in Section 5.4. Another problem is that a volunteer running a VPN server can tap or modify the decapsulated packets of VPN users. This problem is not new. Existing open proxies and Tor exit relays have the same problem, and we currently have no solution to offer. Lastly, a VPN server can potentially use up the bandwidth of a volunteer's Internet line. In response, volunteers can use traffic shaping tools such as NetLimiter[4] to limit the bandwidth of VPN servers.

The innocent IP mixing technique could disturb the owners of innocent IP addresses with users in a country subject to censorship. We have not yet received any complaints from such IP address owners.

Instead of probing VPN servers, a censorship authority could build a whitelist. Maintaining such a whitelist, however, would be quite difficult. Even in a country subject to censorship, Internet access is vital for both residents and visiting businesspeople. One day we might suddenly mix in the IP address of Yahoo! US Mail. The

[3] https://metrics.torproject.org/graphs.html

[4] http://www.netlimiter.com/

Table 5. Monthly transition of the number of GFW authority IP address blocks used for probing.

Year: 2013	IP address blocks (/24)	New blocks	Reused blocks	% reuse
March	2,792	2,792	0	0%
April	2,645	441	2,204	83%
May	1,199	103	1,096	91%
June	1,509	93	1,416	94%
July	1,856	235	1,621	87%
August	1,792	98	1,694	95%
September	1,516	92	1,424	94%
October	1,168	129	1,039	89%

next day we might mix in some important servers for Amazon EC2. Moreover, some servers share the same Akamai or other CDN IP addresses. It would be impossible for a censorship authority to find all important hosts and put their IP addresses into a whitelist in advance.

A censorship authority could also run fake VPN Gate servers to paralyze the VPN Gate network. Such fake servers could send fake logs with false IP addresses to the VPN Gate List Server in order to induce errors in our collaborative spy detection. The false IP addresses could include the valid IP addresses of innocent users. A small number of fake servers cannot impact the entire network, because we can ignore such a small number of false IP addresses. If a censorship authority ran many fake servers, however, these could impact the network. It would be very costly to run so many servers. We assume that a censorship authority would not be willing to pay for such an active attack.

Well-budgeted censorship authorities, like the GFW authority, probably have a large number of IP address blocks available for probing sources. Such IP connectivity infrastructure, however, should have long-term assigned static IP address blocks. Such IP address blocks cannot change frequently. Table 5 lists the actual detected numbers of probing source IP addresses operated by the GFW authority in part of 2013. According to this data, every month the GFW authority reused most of the IP address blocks that had appeared in the previous month. This implies that the GFW authority has only about 4,000 IP address blocks as fixed infrastructure.

6.7. Updated experiences

Since the submission of this paper to NSDI, the following events have happened.

On September 2, 2013 the blocking function of the GFW against VPN Gate became unstable. First, the GFW suddenly stopped blocking all the VPN Gate servers. A few hours later, the GFW recovered and began blocking VPN Gate again. This alternation between blocking and non-blocking continued for a few days.

Since September 5, the GFW has completely stopped blocking all the VPN Gate servers while continuing the probing activity. All servers were reachable through the GFW from September 5, 2013 to February 4, 2014. We do not know why the GFW stopped the blocking.

The number of users and volunteers has increased continuously. On February 4, 2014 we had 5,200 daily volunteers, 1,049,000 daily connections (156,000 unique IP addresses) worldwide, and 123,000 daily connections (16,000 unique IP addresses) from China.

7. Conclusions

We have designed and implemented VPN Gate, a VPN relay system with strong blocking resistance to censorship firewalls such as China's Great Firewall (GFW). In VPN Gate, we use two key techniques to achieve blocking resistance: innocent IP mixing, and collaborative spy detection. We have achieved a proportion of 60-70% of VPN servers not blocked by the GFW. Users in a country subject to censorship can bypass a firewall if they can reach at least one unblocked VPN server. Censorship authorities must block all VPN servers, and this is a very hard task.

VPN Gate works effectively because it relies on many volunteers. We have spent nothing on providing VPN relaying functions. Instead, distributed volunteers contribute small amounts of their electric power and line bandwidth. In contrast, censorship authorities must build expensive censorship infrastructures, implement complex probing programs, and operate them at all times.

The tension between stronger blocking and stronger blocking resistance is essentially a cat-and-mouse game. It is not a fair game, however, and blocking resistance has advantages over blocking. After launching VPN Gate, we played this game with the GFW authority, and we have won the game for the moment. In the future, we are ready to improve our blocking resistance.

In the future, we would like to improve the scalability of the VPN Gate List Server. Additionally, we plan to support IPv6 in VPN Gate.

Note that we did not violate any laws in Japan, where we performed all studies, research, and implementation of blocking resistance to foreign censorship firewalls.

8. Acknowledgements

We would like to thank the NSDI reviewers, our shepherd Professor Sharon Goldberg, Professor Kozo Itano, Professor Kazuhiko Kato, Professor Hisashi Nakai, Professor Akira Sato, Professor Takahiro Shinagawa, Doctor Hiromitsu Takagi, Doctor Tetsuo Sugiyama, Doctor Junpei Kuwana, Doctor Mitsuo Yoshida, Takao Ito, Mei Sharie Ann Yamaguchi, Satoshi Matsumoto, Genya Hatakeyama, Christopher Smith, the Academic Computing and Communications Center of University of Tsukuba, and the National Police Agency of Japan. We are also grateful to the many volunteers and users of VPN Gate.

9. References

[1] Daniel Anderson: "Splinternet Behind the Great Firewall of China", ACM Queue, Vol. 10, No. 11, 2012.

[2] "Azureus User Guide", http://azureus.sourceforge.net/.

[3] Oliver Berthold, Hannes Federrath, and Stefan Koopsell: "Web MIXes: A system for anonymous and unobservable Internet access", Designing Privacy Enhancing Technologies, Springer LNCS 2009, pp 115-129, 2001.

[4] "BitTorrent User Manual", http://www.bittorrent.com/help/manual/.

[5] Richard Clayton, Steven J. Murdoch, and Robert N. M. Watson: "Ignoring the Great Firewall of China", In the Proceedings of the Sixth Workshop on Privacy Enhancing Technologies (PET 2006), pp.20-35, 2006.

[6] Bram Cohen: "Incentives build robustness in BitTorrent", In Proceedings of the Workshop on Economics of Peer-to-Peer Systems, pp.68-72, 2003.

[7] Roger Dingledine, Nick Mathewson, and Paul Syverson: "Tor: the second-generation onion router", In Proceedings of the 13th conference on USENIX Security Symposium, 2004.

[8] Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, and David Karger: "Infranet: Circumventing Web Censorship and Surveillance", In the Proceedings of the 11th USENIX Security Symposium, August 2002.

[9] Nick Feamster, Magdalena Balazinska, Winston Wang, Hari Balakrishnan, and David Karger: "Thwarting web censorship with untrusted messenger discovery", In Proceedings of the 3rd Workshop on Privacy Enhancing Technologies (PET 2003), Springer LNCS 2760, pp. 125-140, 2003.

[10] "Free VPN Info and PC Tips @ VpnSurfing.com", vpnsurfing.com. [Online]. Available: http://www.vpnsurfing.com/. [Accessed: 04-Sep-2013].

[11] David Isaac Wolinsky, Kyungyong Lee, P. Oscar Boykin, Renato Figueiredo: "On the design of autonomic, decentralized VPNs", 6th International Conference on the Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2010.

[12] Stefan Köpsell and Ulf Hilling: "How to achieve blocking resistance for existing systems enabling anonymous web surfing", In Proceedings of the Workshop on Privacy in the Electronic Society (WPES 2004), 2004.

[13] David Martin and Andrew Schulman: "Deanonymizing users of the SafeWeb anonymizing service", Proceedings of the 11th USENIX Security Symposium, 2002.

[14] Damon McCoy and Jose Andre Morales and Kirill Levchenko: "Proximax: A measurement based system for proxies dissemination," Financial Cryptography and Data Security, 2011.

[15] "PrivacyProtectorGVN", Privacy Protector. [Online]. Available: https://privacyprotector.eu/technology/. [Accessed: 04-Sep-2013].

[16] “rbox-tor: an easy to use Tor server”, redct. [Online]. Available: http://redct.info/rbox/tor.html. [Accessed: 13-Sep-2013].

[17] "Tor Project: obfsproxy", torproject.org. [Online]. Available: https://www.torproject.org/projects/obfsproxy. [Accessed: 04-Sep-2013].

[18] Tim Wilde: "Great Firewall Tor probing Circa 09 Dec 2011." [Online]. Available: https://gist.github.com/da3c7a9af01d74cd7de7. [Accessed: 04-Sep-2014].

[19] Philipp Winter and Stefan Lindskog: "How the great firewall of china is blocking Tor," Proceedings of the 2nd USENIX Workshop on Free and Open Communications on the Internet, 2012.

[20] Eric Wustrow, Scott Wolchok, Ian Goldberg, and J. Alex Halderman: "Telex: Anticensorship in the network infrastructure", In the Proceedings of the 20th USENIX Security Symposium, August 2011.

[21] Xueyang Xu, Z. Morley Mao and J. Alex Halderman: "Internet censorship in China: Where does the filtering occur?", In Proceedings of the 12th International Conference on Passive and Active Measurement (PAM 11), pp. 133-142, 2011.

[22] Chenfeng Vincent Zhou, Christopher Leckie, and Shanika Karunasekera: "A survey of coordinated attacks and collaborative intrusion detection", Computers & Security, Vol.29, No.1, pp.124-140. 2010.

Bolt: Data management for connected homes

Trinabh Gupta
University of Texas at Austin

Rayman Preet Singh
University of Waterloo

Amar Phanishayee Jaeyeon Jung Ratul Mahajan

Microsoft Research

Abstract— We present Bolt, a data management system for an emerging class of applications—those that manipulate data from connected devices in the home. It abstracts this data as a stream of time-tag-value records, with arbitrary, application-defined tags. For reliable sharing among applications, some of which may be running outside the home, Bolt uses untrusted cloud storage as seamless extension of local storage. It organizes data into chunks that contains multiple records and are individually compressed and encrypted. While chunking enables efficient transfer and storage, it also implies that data is retrieved at the granularity of chunks, instead of records. We show that the resulting overhead, however, is small because applications in this domain frequently query for multiple proximate records. We develop three diverse applications on top of Bolt and find that the performance needs of each are easily met. We also find that compared to OpenTSDB, a popular time-series database system, Bolt is up to 40 times faster than OpenTSDB while requiring 3–5 times less storage space.

1 Introduction

Our homes are increasingly filled with connected devices such as cameras, motion sensors, thermostats, and door locks. At the same time, platforms are emerging that simplify the development of applications that interact with these devices and query their state and sensed data. Examples include HomeOS [20], MiCasaVerde [5], Nest [6], Philips Hue [2], and SmartThings [9].

While such platforms provide high-level abstractions for device interaction, they do not provide such abstractions for data management other than the local file system. Many applications, however, require richer data manipulation capabilities. For instance, PreHeat stores data from occupancy sensors and uses historical data from specific time windows to predict future occupancy and control home heating [36]; Digital Neighborhood Watch (DNW) [16, 19] stores information about objects seen by security cameras, and shares this information upon request by neighboring homes, based on a time window specified in the requests. Developing such applications today is difficult; developers must implement their own data storage, retrieval, and sharing mechanisms.

Our goal is to simplify the management of data from connected devices in the home. By studying existing and proposed applications, we uncover the key requirements for such a system. First, it should support time-series data and allow for values to be assigned arbitrary tags; device data is naturally time-series (e.g., occupancy sensor readings) and tags can provide a flexible way to assign application-specific semantics (e.g., DNW may use "car" as a tag for object information). Second, the system should enable sharing across homes because many applications need access to data from multiple homes (e.g., DNW application or correlating energy use across homes [14]). Third, the system should be flexible to support application specified storage providers because applications are in the best position to prioritize storage metrics like location, performance, cost, and reliability. Fourth, the system should provide data confidentiality, against potential eavesdroppers in the cloud or the network, because of the significant privacy concerns associated with home data. As we discuss later, none of the existing systems meet these requirements.

We develop Bolt, a system for efficiently storing, querying, and sharing data from connected home devices. It abstracts data as a stream of time-tag-value records, over which it builds indices to support efficient querying based on time and tags. To facilitate sharing, even when individual homes may be disconnected, Bolt uses cloud storage as seamless extension of local storage. It organizes data into chunks of multiple records and compress these chunks prior to the transfer, which boosts storage and network efficiency. To protect confidentiality, the chunks are encrypted as well. Bolt decouples the index from the data to support efficient querying over encrypted data. Applications use the index to identify and download the chunks they need. Our design

leverages the nature of queries in this domain. Applications are often interested in multiple proximate records. Retrieving data at the granularity of chunks, rather than individual records, improves network efficiency through batching of records and improves performance through prefetching records for subsequent queries.

Our current implementation of Bolt supports Windows Azure and Amazon S3 as cloud storage. We evaluate it first using microbenchmarks to understand the overhead of Bolt's streams supporting rich data abstractions compared to raw disk throughput. We find that chunking improves read throughput by up to three times due to strong temporal locality of reads and that the overhead of encrypting data is negligible.

We then develop three diverse applications on top Bolt's APIs and evaluate their read and write performance. We find that Bolt significantly surpasses the performance needs of each application. To place its performance in context, we compare Bolt to OpenTSDB [10], a popular data management system for time-series data. Across the three applications, Bolt is up to 40 times faster than OpenTSDB while requiring 3–5 times less storage space. OpenTSDB does not provide data confidentiality, which makes our results especially notable; by customizing design to the home setting, Bolt simultaneously offers confidentiality and higher performance.

While our work focuses on the home, connected devices are on the rise in many other domains, including factories, offices, and streets. The number of these devices worldwide is projected to surpass 50 billion by 2020 [3]. Effectively managing the deluge of data that these devices are bound to generate is a key challenge in other domains too. The design of Bolt can inform systems for data storage, querying, and sharing in other domains as well.

2 Application Requirements

To frame the requirement for our system, we surveyed several applications for connected homes. We first describe three of them, which we pick for their diversity in the type of data they manipulate and their access patterns. We then outline the requirements that we inferred.

2.1 Example applications

PreHeat: PreHeat uses occupancy sensing to efficiently heat homes [36]. It records home occupancy and uses past patterns to predict future occupancy to turn a home's heating system on or off. It divides a day into 15-minute time slots (i.e., 96 slots/day), and records the occupancy value at the end of a slot: 1 if the home was occupied during the preceding slot, and 0 otherwise. At the start of each slot, it predicts the occupancy value, using the slot occupancy values for the past slots on the same day and corresponding slots on previous days. For instance, for the nth slot on the day d, it uses occupancy values for slots $1 \cdots (n-1)$ on the day d. This is called the *partial occupancy vector* (POV^n_d). Additionally, PreHeat uses $POV^n_{d-1}, POV^n_{d-2}, \ldots, POV^n_1$. Of all past POVs, K POVs with the least Hamming distance to POV^n_d are selected. These top-K POVs are used to predict occupancy, and the heating system is turned on or off accordingly.

Digital Neighborhood Watch (DNW): DNW helps neighbors jointly detect suspicious activities (e.g., an unknown car cruising the neighborhood) by sharing security camera images [16, 19]. The DNW instance in each home monitors the footage from security cameras in the home. When it detects a moving object, it stores the object's video clip as well as summary information such as:

```
Time: 15:00 PDT, 27th September, 2013
ID: 001
Type: human
Entry Area: 2
Exit Area: 1
Feature Vector :{114, 117, ... , 22}.
```

This summary includes the inferred object type, its location, and its feature vector which is a compact representation of its visual information.

When a home deems a current or past object interesting, it asks neighbors if they saw the object around the same time. Each neighbor extracts all objects that it saw in a time window (e.g., an hour) around the time specified in the query. If the feature vector of one of the objects is similar to the one in the query, it responds positively and optionally shares the video clip of the matching object. Responses from all the neighbors allow the original instance to determine how the object moved around in the neighborhood and if its activity is suspicious.

Energy Data Analytics (EDA): Utility companies around the world are deploying smart meters to record and report energy consumption readings. Given its fine-grained nature, compared to one reading a month, data from smart meters allows customers to get meaningful insight into their energy consumption habits [39]. Much recent work has focused on analysing this data, for instance, to identify the consumption of different appliances, user activities, and wastage [14, 22, 29, 33].

A specific EDA application that we consider is where the utility company presents to consumers an analysis of their monthly consumption [14]. It disaggregates hourly home energy consumption values into different categories—base, activity driven, heating or cooling driven, and others. For each home, the variation in consumption level as a function of ambient temperature is analysed by computing for each temperature value the median, 10th, and 90th-percentile home energy consumption. These quantities are then reported to the consumer, along with a comparison with other homes in the neighborhood or city.

Other applications that we surveyed include DigiSwitch [17], which supports elders who reside separately from their caregivers by sharing sensed activity in the home, and a few commercial systems such as Kevo that come with the Kwikset wireless door lock [4]. The requirements, which we describe next, cover these applications as well.

2.2 Data management requirements

We distill the requirements of connected home applications into four classes.

Support time-series, tagged data: Most applications generate time-series data and retrieve it based on time windows. The data may also be tagged and queried using application-specific concepts. For example, object type "human" is a possible tag in DNW and "heating consumption" is a possible tag in EDA.

We make other observations about data manipulation patterns of home applications. First, data in these settings has a *single* writer. Second, writers always generate new data and do not perform random-access updates or deletes. Third, readers typically fetch multiple proximate records by issuing temporal range & sampling queries for sliding or growing time windows. Traditional databases with their support for transactions, concurrency control, and recovery protocols are an overkill for such data [37], and file-based storage offers inadequate query interfaces.

Efficiently share data across homes: It is not uncommon for applications to access data from multiple homes. Both DNW and EDA fall in this category. Online storage services, like Dropbox [1] or OneDrive [7], can simplify cross-home sharing, but they will unnecessarily synchronize large quantities of data. Applications may want to access only part of the data produced by a device. For example, in DNW, it would be wasteful to access the entire day worth of video data if the search for suspicious objects needs to be done only over a few hours.

Support policy-driven storage: Different types of data have different storage requirements for location, access performance, cost, and reliability. A camera that records images might store them locally and delete them once the DNW application has extracted images of objects in them. The DNW application might store these images on a remote server to correlate with images captured by neighbours. Once analysed, they can be stored on cheaper archival storage servers. Applications are in the best position to prioritize storage metrics and should be able to specify these policies.

Ensure data confidentiality & integrity: Applications may use remote storage infrastructure to simplify data management and sharing, but may not trust them for confidentiality or integrity of data. Data generated by home applications may contain sensitive information; DNW contains clips of residents, and data from occupancy sensors and energy meters reveal when residents are away, which can be exploited by attackers. Therefore, the data management system for these applications should guarantee the confidentiality and integrity of stored data. The system should also support efficient changes in access policies, without requiring, for instance, re-encryption of a large amounts of data.

Efficiently meeting all the requirements above is challenging. For example, storing data locally facilitates confidentiality but inhibits efficient sharing, remote access, and reliable storage. By the same token, storing data in the cloud provides reliable storage and sharing, but untrusted storage servers can compromise confidentiality; also, sharing by synchronizing large amounts of data is inefficient. Finally, naïvely storing encrypted data on untrusted servers inhibits efficient sharing.

As we review in detail in §7, existing systems either expose inefficient sharing and querying abstractions for temporal data [21, 23, 30], assume partial or complete trust on the storage servers [31], or store data locally while ignoring application storage policies [25]. In the following sections we describe the design of Bolt to support the storage requirements listed above.

3 Overview of Bolt

The data abstraction exposed by Bolt is a stream in which each record has a timestamp and one or more tag-value pairs, i.e., <timestamp, <tag1,value1>, [<tag2, value2>, ...]>. Streams are uniquely identified by the three-tuple: <HomeID, AppID, StreamID>. Bolt supports filtering and lookups on streams using time and tags.

3.1 Security assumptions and guarantees

Bolt does not trust either the cloud storage servers or the network to maintain data confidentiality or integrity. We assume that the storage infrastructure is capable of unauthorized reading or modification of stream data, returning old data, or refusing to return any data at all. Atop this untrusted infrastructure, Bolt provides the following three security and privacy guarantees:

1. Confidentiality: Data in a stream can be read only by an application to which the owner grants access, and once the owner revokes access, the reader cannot access data generated *after revocation*.
2. Tamper evidence: Readers can detect if data has been tampered by anyone other than the owner. However, Bolt does not defend against denial-of-service attacks, e.g., where a storage server deletes all data or rejects all read requests.
3. Freshness: Readers can detect if the storage server returns stale data that is older than a configurable time window.

3.2 Key techniques

The following four main techniques allow Bolt to meet the requirements listed in the previous section.

Chunking: Bolt stores data records in a log per stream, enabling efficient append-only writes. Streams have an index on the datalog to support efficient lookups, temporal range and sampling queries, and filtering on tags. A contiguous sequence of records constitute a *chunk*. A chunk is a basic unit of transfer for storage and retrieval. Data writers upload chunks instead of individual records. Bolt compresses chunks before uploading them, which enhances transfer and storage efficiency.

Readers fetch data at the granularity of chunks as well. While this means that more records than needed may be fetched, the resulting inefficiency is mitigated by the fact that applications, like the ones we surveyed earlier, are often interested in multiple records in a time window, rather than a single record generated at a particular time. Fetching chunks, instead of individual records, makes common queries with temporal locality efficient, avoiding additional round trip delays.

Separation of index and data: Bolt always fetches the stream index from the remote server, and stores the index locally at readers and writers; data may still reside remotely. This separation opens the door to two properties that are otherwise not possible. First, because the index is local, queries at endpoints use the local index to determine what chunks should be fetched from remote servers. No computation (query engine) is needed in the cloud, and storage servers only need to provide data read-/write APIs, helping reduce the cost of the storage system. Second, it allows Bolt to relax its trust assumptions of storage servers, supporting untrusted cloud providers without compromising data confidentiality by encrypting data. The data can be encrypted before storing and decrypted after retrievals, and the provider needs to understand nothing about it. Supporting untrusted cloud providers is challenging if the provider is expected to perform index lookups on the data.

Segmentation: Based on the observation that applications do not perform random writes and only append new data, streams can grow large. Bolt supports archiving contiguous portions of a stream into segments while still allowing efficient querying over them. The storage location of each segment can be configured, enabling Bolt streams to use storage across different providers. Hence, streams may be stored either locally, remotely on untrusted servers, replicated for reliability, or striped across multiple storage providers for cost effectiveness. This configurability allows applications to prioritize their storage requirements of space, performance, cost, and reliability. Bolt currently supports local streams, Windows Azure storage, and Amazon S3.

Decentralized access control and signed hashes: To maintain confidentiality in the face on untrusted storage servers, Bolt encrypts the stream with a secret key generated by the owner. Bolt's design supports encryption of both the index and data, but by default we do not encrypt indices for efficiency,[1] though in this configuration information may leak through data stored in indices. Further, we use lazy revocation [28] for reducing computation overhead of cryptographic operations. Lazy revocation only prevents evicted readers from accessing *future* content as the content before revocation may have already been accessed and cached by these readers. Among the key management schemes for file systems using lazy revocation, we use the hash-based key regression [24] for its simplicity and efficiency. It enables the owner to share only the most recent key with authorized readers, based on which readers can derive all the previous keys to decrypt the content.

We use a trusted key server to distribute keys. Once a stream is opened, all subsequent reads and writes occur directly between the storage server and the application. This way, the key server does not become a bottleneck.

To facilitate integrity checks on data, the owners generate a hash of stream contents, which is verified by the readers. To enable freshness checks, similar to SFSRO [23] and Sirius [26], freshness time window is part of the stream metadata. It denotes until when the data can be deemed fresh; it is based on the periodicity with which owners expect to generate new data. Owners periodically update and sign this time window, which readers can check against when a stream is opened.

4 Bolt Design

We now describe the design of Bolt in more detail.

4.1 APIs

Table 1 shows the Bolt stream APIs. Applications, identified by the <HomeID, AppID> pair, are the principals that read and write data. On `create` and `open`, they specify the policies shown in Table 2, which include the stream's type, storage location, and protection and sharing requirements. The stream type can be *ValueStream*, useful for small data values such as temperature readings, or *FileStream*, useful for large values such as images or videos. The two types are stored differently on disk.

Each stream has one writer (owner) and one or more readers. Writers add time-tag-value records to the stream using `append`. Records can have multiple tag-value pairs and multiple tags for a value. Tags and values are application-defined types that implement IKey and IValue interfaces, allowing Bolt to hash, compare, and

[1] Index decryption and encryption is a one time cost paid at stream open & close respectively and is proportional to the size of the index.

Function	Description
createStream(name, R/W, policy)	Create a data stream with specified policy properties (see Table 2)
openStream(name, R/W)	Open an existing data stream
deleteStream(name)	Delete an existing data stream
append([tag, value])	Append the list of values with corresponding tags. All get same timestamp
append([tag], value)	Append data labelled with potentially multiple tags
getLatest()	Retrieve latest $< time, tag, value >$ tuple inserted across *all* tags
get(tag)	Retrieve latest $< time, tag, value >$ tuple for the *specified* tag
getAll(tag)	Retrieve all time-sorted $< time, tag, value >$ tuples for specified tag
getAll(tag, t_{start}, t_{end})	Range query: get all tuples for tag in the specified time range
getAll(tag, t_{start}, t_{end}, t_{skip})	Sampling range query
getKeys(tag_{start}, tag_{end})	Retrieve all tags in the specified *tag range*
sealStream()	Seal the current stream segment and create a new one for future appends
getAllSegmentIDs()	Retrieve the list of all segments in the stream
deleteSegment(segmentID)	Delete the specified segment in the current stream
grant(appId)	Grant appId read access
revoke(appId)	Revoke appId's read access

Table 1: Bolt stream APIs: Bolt offers two types of streams: (i) ValueStreams for small data values (e.g., temperature readings); and (ii) FileStreams for large values (e.g., images, videos).

Property	Description
Type	ValueStream or FileStream
Location	Local, Remote, or Remote Replicated
Protection	Plain or Encrypted
Sharing	Unlisted (private) or Listed (shared)

Table 2: Properties specified using Bolt policies

serialize them. Finally, writers can `grant` and `revoke` read access other applications. Readers can filter and query data using tags and time (`get*`). We currently support querying for the latest record, the latest record for a tag, temporal range and sampling queries, and range queries on tags. Range queries return an iterator, which fetches data on demand, when accessed.

4.2 Writing stream data

An owner first creates a new Bolt stream and appends data records to it. Figure 1 shows the data layout for a stream. Streams consist of two parts: a log of data records (DataLog), and an index that maps a tag to a list of data item identifiers. Item identifiers are fixed-size entries and the list of item identifiers in the index is sorted by time, enabling efficient binary searches for range and sampling queries. The index is memory resident and backed by a file; records in the DataLog are on disk and retrieved when referenced by the application. The DataLog is divided into fixed sized chunks of contiguous data records.

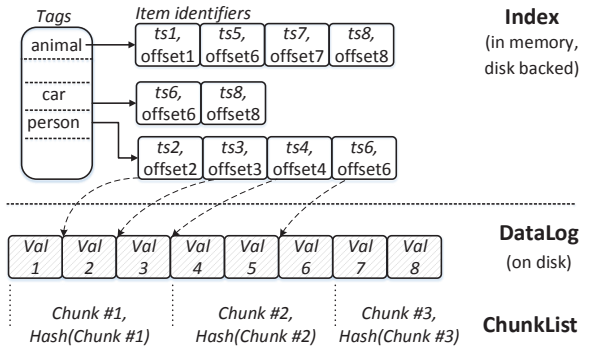


Figure 1: Data layout of a ValueStream. FileStream layout is similar, except that the values in the DataLog are pointers to files that contain the actual data.

To reduce the memory footprint of the index, which can grow large over time, streams in Bolt can be archived. This snapshot of a stream is called a segment, where each segment has its own DataLog and corresponding index. Hence, streams are a time ordered list of *segments*. If the size of the index in memory exceeds a configurable threshold ($index_{tresh}$), the latest segment is *sealed*, its index is flushed to disk, and a new segment with a memory resident index is created. Writes to the stream always go to the *latest* segment and all other segments of the stream are read-only entities. The index for the latest segment of the stream is memory resident and backed by a file (Figure 1); As shown in Figure 2 all other segments are sealed and store their indices on disk with

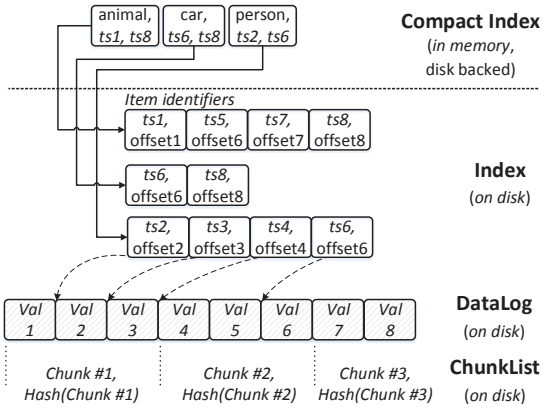


Figure 2: Data layout of a sealed segment in Bolt.

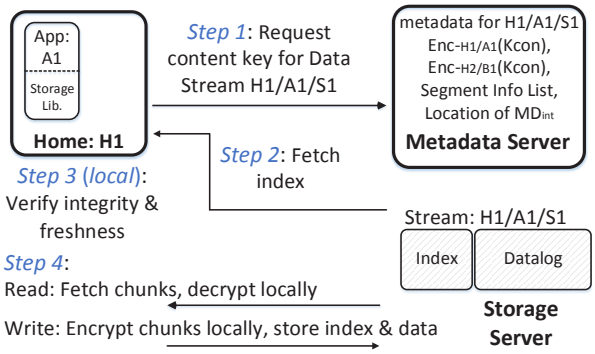


Figure 3: Steps during reads and writes for application A1 in home H1 accessing stream $H1/A1/S1$

a compact in-memory index header that consists of the tags, the timestamp for the first and last identifier in their corresponding item identifier list, and the location of this list in the index file.

4.3 Uploading stream data

Each principal (<HomeID, AppID> pair) in Bolt is associated with a private-public key pair, and each stream in Bolt is encrypted with a secret key, K_{con}, generated by the owner. When a stream is synced or closed, Bolt flushes the index to disk, chunks the segment DataLog, compresses and encrypts these chunks, and generates the `ChunkList`. The per-segment ChunkList is an ordered list of all chunks in the segment's DataLog and their corresponding hashes. Bolt has to do this for all *mutated segments*: new segments generated since the last `close` and the latest segment which may have been modified due to data appends; All other segments in the stream are sealed and immutable.

Bolt then generates the stream's integrity metadata (MD_{int}). Let n denote the number of segments within the stream. Then, MD_{int} is computed as follows: $\text{Sig}_{K_{\text{priv}}{}^{\text{owner}}}[\text{H}[\text{TTL}||\text{H}[\text{I}_i]||...||\text{H}[\text{I}_n]||\text{H}[\text{CL}_i]||...||\text{H}[\text{CL}_n]]$. Bolt uses `TTL` to provide guarantees on data freshness similar to SFSRO and Chefs [23], ensuring any data fetched from a storage server is no older than a configurable writer-specified consistency period, and also no older than any previously retrieved data. As shown in Table 3, MD_{int} is a signed hash of the duration for which the owner guarantees data freshness (TTL) and the per-segment index and ChunkList hashes. For all mutated segments, Bolt uploads the chunks, the updated ChunkList, and the modified index to the storage server. Chunks are uploaded in parallel and applications can configure the maximum number of parallel uploads. Bolt then uploads MD_{int}. Finally Bolt uploads the stream metadata to the meatadata server if new segments were created.[2]

1. TTL: $t^{\text{fresh}}{}_{\text{start}}$, $t^{\text{fresh}}{}_{\text{end}}$
2. H[x]: Cryptographic hash of x
3. Sig$_K$[x]: Digital signature of x with the key K
4. $K_{\text{pub}}{}^{\text{owner}}$, $K_{\text{priv}}{}^{\text{owner}}$: a public-private key pair of owner
4. CL_i: ChunkList of the i^{th} segment
5. I_i: Index of the i^{th} segment
6. ||: Concatenation

Table 3: Glossary

4.4 Granting and revoking read access

A metadata server, in addition to maintaining the principal to public-key mappings, also maintains the following stream metadata: (i) a symmetric content key to encrypt and decrypt data (K_{con}), (ii) principals that have access to the data (one of them is the owner), (iii) the location of MD_{int}, and (iv) per-segment location and key version. K_{con} is stored encrypted — one entry for each principal that has access to the stream using their public key.

To grant applications read access, the owner updates stream metadata with K_{con} encrypted with the reader's public key. Revoking read access also involves updating stream metadata: owners remove the appropriate principal from the accessor's list, remove the encrypted content keys, roll forward the content key and key version for all valid principals as per key-regression [24]. Key regression allows readers with version V of the key to generate keys for all older versions 0 to $V-1$. On a revocation, Bolt seals the current segment and creates a new one. All chunks in a segment are always encrypted using the same version of the content key.

[2] In our initial design, we assume that stream metadata is stored on a trusted key server to prevent unauthorized updates. In Section 8 we discuss how this assumption can be relaxed.

4.5 Reading stream data

Figure 3 shows the steps during reads; Owners also follow these steps when they reopen their streams. Bolt opens a stream (step 1) and fetches stream metadata. Using this information, Bolt then fetches the stream's integrity metadata (MD_{int}) from untrusted storage. On verifying MD_{int}'s integrity using the owner's public key, and freshness using the `TTL` in MD_{int}, the reader fetches index & ChunkList for every segment of the stream (step 2) and verifies their integrity using MD_{int} (step 3).

Owner can store new data record in the stream on verifying the integrity of index data. For readers, once index & ChunkList integrity verifications for all segments complete (step 3), Bolt uses the index to identify chunks that need to be fetched from remote storage to satisfy `get` requests. Chunk level integrity is checked lazily; Bolt downloads these chunks and verifies their integrity by using the segment's ChunkList. Bolt decrypts and decompress the verified chunk and stores these chunks in a local disk-based cache for subsequent reads.

5 Implementation

We have implemented Bolt using C# over the .NET Framework v4.5. Our implementation is integrated into the HomeOS [20] platform, but it can also be used as an independent library. In addition to the applications we evaluate in the next section, several other HomeOS applications have been ported by others to use Bolt. The client-side code is 6077 lines of code, and the metadata server is 473 lines. Our code is publicly available at `labofthings.codeplex.com`.

Our client library uses Protocol Buffers [8] for data serialization and can currently use Windows Azure and Amazon S3 for remote storage. It uses their respective libraries for reading and writing data remotely. On Azure, each segment maps to a container, the index & DataLog each map to a blob, and individual chunks map to parts of the DataLog blob (blocks). On S3, each segment maps to a bucket, the index maps to an object, and chunks of the DataLog map to individual objects.

The communication between the clients and the metadata server uses the Windows Communication Foundation (WCF) framework. The server is hosted in a Windows Azure VM with 4-core AMD Opteron Processor and 7GB RAM and runs Windows Server 2008 R2.

6 Evaluation

We evaluate Bolt in two ways: 1) microbenchmarks, which compare the performance of different Bolt stream configurations to the underlying operating system's performance, 2) applications, which demonstrate the feasibility and performance of Bolt in real-world use cases. Table 4 summarizes the major results.

Finding	Location
Bolt's encryption overhead is negligible, making secure streams a viable default option.	§6.1.1
Chunking in Bolt improves read throughput by up to 3x for temporal range queries.	§6.1.2
Bolt segments are scalable: querying across 16 segments incurs only a 3.6% overhead over a single segment stream.	§6.1.3
Three applications (PreHeat, DNW, EDA) implemented using Bolt abstractions.	
Bolt is up to 40x faster than OpenTSDB.	§6.2
Bolt is 3–5x more space efficient than OpenTSDB.	

Table 4: Highlights of evaluation results

All Bolt microbenchmark experiments (Section 6.1) are run on a VM on Windows Azure. The VM has a 4-core AMD Opteron Processor, 7GB RAM, 2 virtual hard disks and runs Windows Server 2008 R2. All application experiments (Section 6.2) are run on a physical machine with an AMD-FX-6100 6-core processor, 16 GB RAM, 6 Gbps SATA hard disk, running Windows 7.

6.1 Bolt microbenchmarks

Setup. In each experiment a client issues 1,000 or 10,000 write or read requests for a particular Bolt stream configuration. The size of the data-value written or read is one of 10B, 1KB, 10KB or 100KB. We fix the chunk size for these experiments at 4MB unless otherwise specified. We measure the throughput in operations/second and the storage space used (for writes). To compare Bolt's write performance we bypass Bolt and write data directly to disk (referred to as DiskRaw)–either to a single local file (baseline for ValueStream), to multiple files, one for each data value (baseline for FileStream), or upload data directly to Azure (baseline for remote ValueStream and FileStream). Similarly, for data reads i.e., `Get(tag)` and `GetAll(tag,` $time_{start}$`,` $time_{end}$`)` queries, we compare Bolt's read performance to data read directly from a single local file (baseline for ValueStream), data-values read from separate files (baseline for FileStream), and data read by downloading an Azure blob (baseline for remote ValueStream). We report the mean and standard deviation across 10 runs of each experiment.

6.1.1 Write performance

ValueStream: Figure 4 compares the write throughput at the client for three different data-value sizes (10B, 1KB and 10KB). Writes to local ValueStreams are slower than DiskRaw because of the overhead of three additional

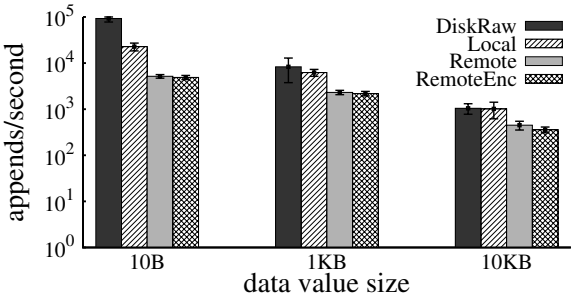


Figure 4: Write throughput (appends/second) for Bolt ValueStreams (local, remote, remote encrypted)

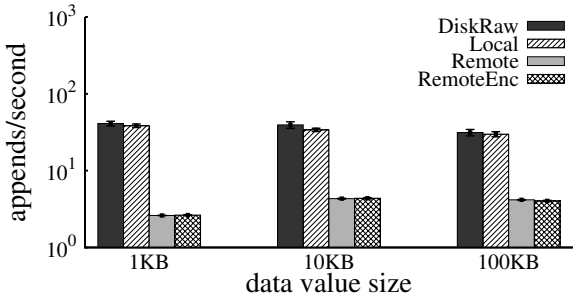


Figure 5: Write throughput (appends/second) for Bolt FileStreams (local, remote, remote encrypted)

subtasks: index update/lookup, data serialization, and writing index & DataLog to disk. Table 5 shows these breakdown for 10,000 appends of 10B values. As the size of individual records inserted goes up, the throughput, measured in MBps, goes up; DiskRaw and local ValueStreams saturate the local disk write throughput. For supporting the new abstractions, the storage space taken by Bolt streams is 0.1× (for large values) to 2.5× (for 10B values) compared to DiskRaw for local writes.

For remote streams, we find that the absolute time taken to perform the above mentioned subtasks is similar, however, a high percentage of the total time is spent uploading the DataLog and index to the cloud. For example, Table 5 shows that 64% of the total time is taken to chunk & upload the DataLog; uploading the index took close to 3% of the total time. Chunking and uploading involves the following six major components: (i) reading chunks from the DataLog and computing the hash of their contents, (ii) checking the blob's existence on the storage server and creating one if not present, (iii) compressing and encrypting chunks if needed, (iv) uploading individual chunks to blocks in a blob, (v) committing the new block list to Azure reflecting the new changes, and (vi) uploading the new chunk list containing chunk IDs and their hashes. For remote-encrypted streams, the time taken to encrypt data is less than 1% of the total time.

FileStream: Figure 5 compares the write throughput for 1000 appends at the client, for three different data-value sizes (1KB, 10KB, and 100KB). Unlike ValueStreams, the latency for writes in FileStreams is influenced primarily by two tasks: writing each data record to a separate file on disk and uploading each file to a separate Azure blob. As a result, the performance of local FileStream is comparable to DiskRaw. For remote streams writing to local files matches the performance of local streams, but creating a new Azure blob for every data record in the stream dominates the cost of writes (over 80% of the total time). Encryption has an overhead of approximately 1%.

Storage overhead: Table 6 shows the storage overhead of Bolt's local ValueStreams over DiskRaw for 10B, 1KB, 10KB value sizes. In DiskRaw tag-value pairs and timestamp are appended to a file on disk. ValueStream's overheads are primarily due to offsets in the index, and index & DataLog serialization data. Bolt stores each unique tag only once in the index, benefiting streams with large tags. We define storage overhead as the amount of additional disk space used by ValueStream compared to DiskRaw, expressed as a percentage. Storage overhead decreases with larger value sizes, but remains constant with increasing number of data records for a given value size. Stream metadata overhead does not change with value size and is small.

Component	Local	Remote	Remote Encrypted
Lookup, Update Index	5%	2.1%	0.6%
Data Serialization	14.3%	2.3%	1.7%
Flush Index	39.8%	9.9%	10.2%
Flush Data	33.2%	7.3%	10.5%
Uploading Chunks	-	63.6%	61.9%
Encrypting Chunks	-	-	0.6%
Uploading Index	-	2.8%	2.6%

Table 5: Percentage of total experiment time spent in various tasks while appending 10,000 items to a ValueStream for 10B value sizes.

6.1.2 Read Performance

ValueStream: Figure 6 compares the read throughput at the client for three different data-value sizes (10B, 1KB and 10KB) using three ValueStream configurations. The client issues 10,000 `Get(tag)` requests with a randomly selected tag on every call.

In DiskRaw, values are read from random parts of

Value Size	% overhead
10 B	30.6
1 KB	0.86
10 KB	0.09

Table 6: Storage overhead of local ValueStreams over DiskRaw. This percentage overhead is independent of the number of data items inserted.

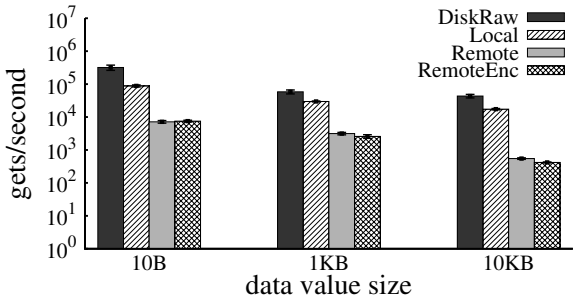


Figure 6: Read throughput, in `Get(key)` calls per second with randomly selected keys, for Bolt ValueStreams (local, remote, and remote encrypted)

the DataLog. Despite random reads, for both DiskRaw and ValueStream streams, the effect of file system buffer cache shows up in these numbers. Local ValueStreams incur an overhead of index lookup and data deserialization. For example, for 1KB sized data values, local ValueStreams spend 5% of the total time in index lookup, 60% reading records from the DataLog (matching DiskRaw speeds), and 30% deserializing data. Remote reads in ValueStream are dominated by the cost of downloading chunks from Azure and storing them in the chunk cache (90% of the read latency).

FileStream: Compared to DiskRaw, FileStreams incur the overhead of index lookup, downloading individual blobs from remote storage, and reading the data record from the file. Figure 7 shows the effect of these on throughput. For remote streams most of the time (99%) is spent downloading individual blobs from remote storage. For remote-encrypted streams, the time taken to decrypt data is less than 1% of the total time.

Effect of chunking on temporal range queries: Figure 8 shows that chunking improves read throughput as it batches transfers and prefetches data for range queries with locality of reference. We experiment with two range queries that retrieve the same amount of data, one with a narrow window of 10 records, and another with a larger window of 100 records; the start times of the windows

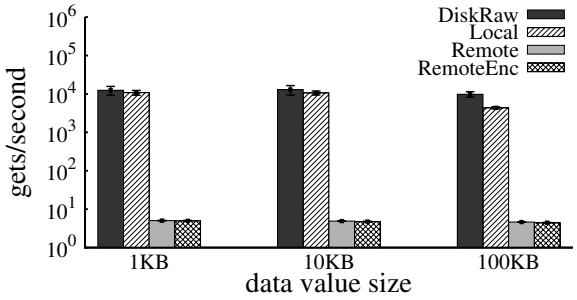


Figure 7: Read throughput, in `Get(key)` calls per second with randomly selected keys, for Bolt FileStreams (local, remote, and remote encrypted).

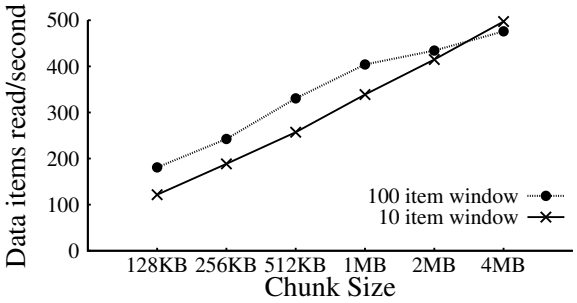


Figure 8: Effect of chunking on reads. Chunking improves throughput because of batching transfers and prefetching data for queries with locality of reference. In this experiment, the stream contains 10,000 data items each 10KB in size.

are picked randomly from the time range of data stored. Larger chunks cause higher read throughput by reducing the number of chunk downloads as chunks are cached locally. For a fixed chunk and value size, queries with a wider window have comparatively larger throughput. This is because wider queries cause fewer downloads by leveraging caching to answer queries. Whereas narrow queries are comparatively dispersed across the DataLog, hence causing more chunk downloads.

6.1.3 Scalability of Bolt ValueStream segments

Figure 9 shows the effect of scaling the number of segments for a local ValueStream on opening the stream (one time cost), index lookup, and reading data records. Each segment has 10,000 keys, and 10,000 Get(key) requests are issued for randomly selected keys. The time taken for opening a stream is dominated by the time to build the segment index in memory and it grows linearly with the number of segments. Query time across segments with compact memory-resident index headers grows negligibly with the number of such segments.

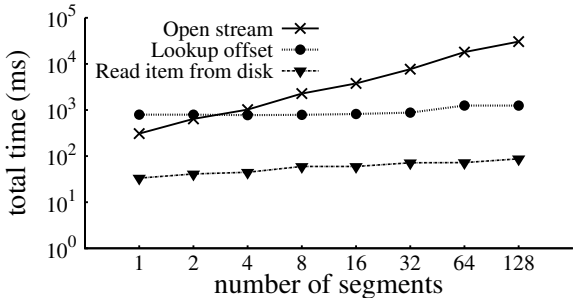


Figure 9: Open, index look-up, and DataLog record retrieval latencies scale well as a function of the number of segments of the stream, while issuing 10,000 Get(key) requests for random keys. Each segment has 10,000 keys.

6.2 Applications

We demonstrate the feasibility and performance of three real-world applications using Bolt: PreHeat, Digital Neighborhood Watch, and Energy Data Analytics. For comparison we also evaluate the performance of these applications using OpenTSDB [10]. It is a popular system for time-series data analytics. It is written in Java and uses HBase to store data. Unlike Bolt's library that is loaded into the client program, OpenTSDB allows querying only over HTTP endpoints. Further, unlike Bolt, OpenTSDB neither provides security guarantees nor the flexibility offered by policy-driven storage.

6.2.1 PreHeat: Occupancy prediction

PreHeat [36], is a system that enables efficient home heating by recording and predicting occupancy information. We described PreHeat's algorithm in Section 2.1. In implementing PreHeat's data storage and retrieval using Bolt, we identify each slot by its starting timestamp. A single *local* unencrypted ValueStream thus stores the *(timestamp, tag, value)* tuples where the *tag* is a string (e.g., "occupancy"). We instantiate two different PreHeat implementations that optimize for either disk storage or retrieval time, and hence store different entities as *values*:

Naïve: In this implementation the value stored for each slot is simply the slot's measured occupancy (0 or 1). Thus for computing predicted occupancy for the nth slot on day d, POVs are obtained by issuing d temporal range queries [`getAll(`k`,` t_s`,` t_e`)`].

Smart: Here the value stored for a slot is its POV concatenated to its measured occupancy value. Thus computing predicted occupancy for the nth slot on day d requires one get(k) query for POV_d^n and $(d-1)$ temporal range queries that return a single value for POV_{d-1}^n, POV_{d-2}^n, ..., POV_1^n. As compared to the naïve implementation, range queries over time are replaced with simple get queries for a particular timestamp. The storage overhead incurred in this approach is larger than the naïve approach, but retrieval latency is reduced due to the reduced number of disk reads.

Naïve + OpenTSDB: We implement the naïve PreHeat approach to store and retrieve data locally from OpenTSDB. It groups occupancy values spanning an hour into one row of HBase (OpenTSDB's underlying datastore). That is, 4 PreHeat slots are grouped into a single HBase row. OpenTSDB's usability is limited by values being restricted to real numbers. Bolt allows byte arrays of arbitrary length to be stored as values. Consequently, an analogous implementation of the smart PreHeat approach is not possible with OpenTSDB.

Of all the 96 slots in a day, the 96th, or last, slot has the maximum retrieval, computation, and append time, as POV_{96}^d is longest $POV_i^d, \forall i \in [1,96]$. Thus to compare the approaches we use the retrieval, computation, and append times for the 96th slot of each day. Figure 10 shows a comparison of time taken to retrieve data for the 96th slot. We observe that as the number of days increase the retrieval latency for both the naïve and smart approaches grows due to increasing number of range and get queries. However, the smart approach incurs less latency than naïve as it issues fewer random disk reads. As compared to OpenTSDB, Bolt performs approximately 40× faster for the naïve approach in analysing 100 days of data. Lastly, as expected, the time taken to perform computation for occupancy prediction is unchanged across the three implementations.

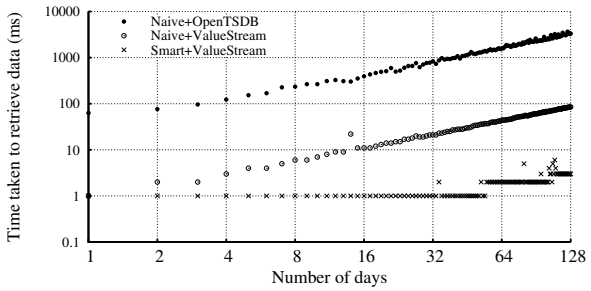


Figure 10: Time to retrieve past occupancy data with increasing duration of a PreHeat deployment.

Table 7 shows the disk footprint incurred by the implementations for 1000 days of operation. We observe that the smart approach uses 8× the storage compared to naïve as it stores slots' POVs in addition to occupancy values. Using Bolt's compressed streams, the naïve scheme achieves up to a 1.6× reduction, and the smart scheme achieves up to a 8× reduction in storage overhead, as compared to their uncompressed stream counterparts. OpenTSDB incurs 3× larger disk footprint than its corresponding implementation using Bolt with uncompressed streams. Row key duplication in HBase is one potential source of storage inefficiency.

To understand the effect of chunk-based prefetching on application performance, we run naïve PreHeat for 10 days using a remote ValueStream, clear the chunk cache, and measure the retrieval time of each slot on the 11th day. Figure 11 shows the average of all 96 slot retrieval

Configuration	Naive	Smart
ValueStream	2.38 MB	19.10 MB
ValueStream using GZip	1.51 MB	3.718 MB
ValueStream using BZip2	1.48 MB	2.37 MB
OpenTSDB	8.22 MB	-

Table 7: Storage space for a 1000-day deployement of PreHeat.

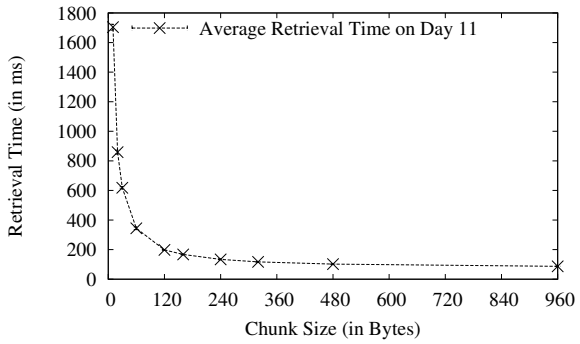


Figure 11: Average retrieval time for all 96 slots on the 11th day of a PreHeat deployment, with a remote ValueStream, decreases with increasing chunk size.

times on the 11^{th} day, for different chunk sizes. As the chunk size increases, the average slot retrieval time decreases, as newly downloaded chunks prefetch additional occupancy values used in subsequent slots' queries.

6.2.2 Digital Neighborhood Watch (DNW)

DNW helps neighbors detect suspicious activities (e.g., an unknown car cruising the neighborhood) by sharing security camera images [16]. We implement DNW's data storage, retrieval and sharing mechanisms using Bolt and OpenTSDB. Due to Bolt's (timestamp, tag, value) abstraction, objects can be stored (and retrieved) in a single remote ValueStream (per home), with each object attribute in a separate tag e.g., type, ID, feature-vector, all bearing the same timestamp. Queries proceed by performing a `getAll(feature-vector, ` t_s`, ` t_e`)` where time window (w)=[t_s,t_e], and then finding a match amongst the retrieved objects. Similarly, in OpenTSDB, each home's objects are recorded against its metric (e.g., home-id), and OpenTSDB tags store object attributes. We run OpenTSDB remotely on an Azure VM. The DNW clients that store and retrieve data, for both Bolt and OpenTSDB, run on our lab machines.

To evaluate DNW, we instrument each home to record an object every minute. After 1000 minutes, one randomly selected home queries all homes for a matching object within a recent time window w. We measure the total time for retrieving all object summaries from ten homes for window sizes of 1 hour and 10 hours.

Figure 12 shows that, for Bolt, larger chunks improve retrieval time by batching transfers. For queries that span multiple chunks, Bolt downloads these chunks on demand. Range queries return an iterator (Section 4.1); When applications uses this iterator to access a data record, Bolt checks if the chunk the data record resides in is present in the cache, and if not downloads the chunk. Hence queries that span many chunks, like the DNW run for 10 hours with a 100KB chunk size, cause these chunks to be downloaded on demand, resulting in multiple round trips and increasing the overall retrieval time. This can be improved by prefetching chunks in parallel, in the background, without blocking the application's range query. For larger chunks, fewer chunks need to be downloaded sequentially, resulting in fewer round trips and improving the overall retrieval time. OpenTSDB has no notion of chunking. Hence OpenTSDB retrieval times are independent of chunk size.

For Bolt, beyond a certain chunk size, additional data fetched in the chunk does not match the query and the benefits of batched transfers on retrieval time plateau out. In fact, because chunk boundaries don't necessarily line up with the time window specified in queries, data records that don't match the query may be fetched even for small chunk sizes. Figure 12(right) shows that as chunk size increases, the data overhead i.e., the percentage of data records in chunks downloaded, that don't match the query's time window w, also increases. Bolt allows applications to chose chunk sizes as per their workloads, by trading overhead for performance.

Lastly, Table 8 shows that Bolt incurs a 3× smaller disk footprint than OpenTSDB.

	DNW	EDA
Bolt	4.64 MB	37.89 MB
OpenTSDB	14.42 MB	212.41 MB

Table 8: Disk space used for 10 homes in DNW for 1000 minutes, and 100 homes in EDA for 545 days.

6.2.3 Energy Data Analytics (EDA)

In this application (Section 2.1), we study a scenario where a utility company presents consumers with an analysis of their consumption on their monthly bill, with a comparison with other homes in the neighborhood, city, or region within a given time window e.g., one month.

In the implementation using Bolt, we use a *single remote ValueStream per home* which stores the smart meter data. For each hour, the energy consumption value is appended to the ValueStream with the mean ambient temperature value of the hour (rounded to the nearest whole number) as the tag. This enables quick retrieval of energy values for a given temperature. In the OpenTSDB based implementation, we create one metric for each temperature value, for each home; i.e. Metric *home-n-T* stores values recorded at T° C, for home n. For each home we retrieve data in the time interval $[t_s,t_e]$ for each temperature T between -30° C and 40° C. The median, 10th, and 90th percentile values computed using one home's data are compared to all other homes. Data for multiple homes is retrieved sequentially.

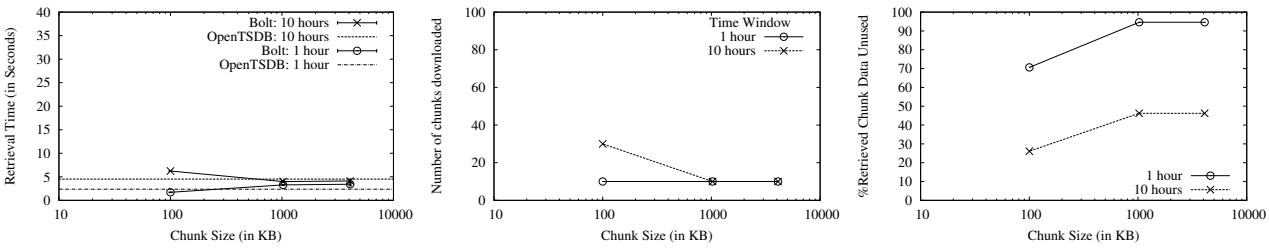


Figure 12: Retrieving DNW summaries from 10 homes, for 1 hour & 10 hour time windows: Chunks improves retrieval time by batching transfers but can get additional data that might not match the query immediately.

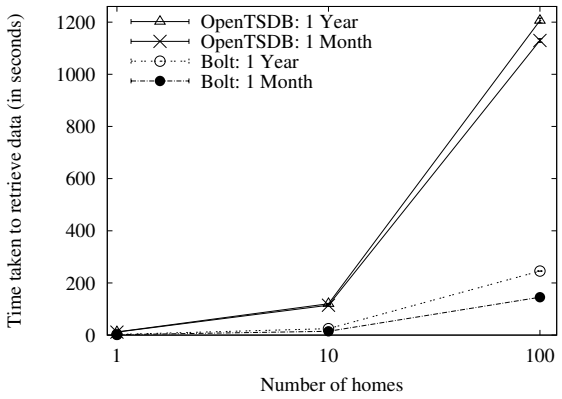


Figure 13: Time taken to retrieve smart meter data of multiple homes for different time windows.

Figure 13 shows the average time, with 95% confidence intervals, taken to retrieve data for two time windows of 1 month, and 1 year, as we increase the number of homes. Bolt uses Windows Azure for storage. We use a home energy consumption dataset from a utility company in Waterloo. Retrieval time for Bolt and OpenTSDB increases proportionally at approximately 1.4 sec/home and 11.4 sec/home respectively, for a one month window; 2.5 sec/home and 12 sec/home respectively for one year window. Bolt outperforms OpenTSDB by an order of magnitude primarily due to prefetching data in chunks; A query for 10° C might be served from the local chunk cache as queries for previous temperature values might have prefetched this data.

Finally, as shown in Table 8, we find that OpenTSDB incurs a 5× larger disk footprint than its corresponding implementation using Bolt.

7 Related Work

Our work is related to three strands of other works: *i*) sharing and managing personal data, *ii*) securing data stored in untrusted remote storage, and *iii*) stream processing systems for temporal data. We discuss each in turn below.

Personal and home data management: Perspective [35] is a semantic file system to help users manage data spread across personal devices such as portable music players and laptops in the home. It exposes a *view* abstraction where a view is an attribute-based description of a set of files with support for queries on file attributes. It allows devices to participate in the system in a peer-to-peer fashion. Security and access control are not a focus of the work. HomeViews [25] eases the management and sharing of files among people. It exposes database-style views over one's files and supports access-controlled sharing of views with remote users in a peer-to-peer manner. Both systems target user-generated data (e.g., photos, digital music, documents) rather than device-generated time series data which is our focus.

Secure systems using untrusted storage: SUNDR [30] is a network file system that provides integrity and consistency guarantees of files stored in untrusted remote storage. SPORC [21] is a framework for building group collaboration services like shared documents using untrusted servers. Venus [38] and Depot [32] expose a key-value store to clients on top of untrusted cloud storage providers. Chefs [23] enables replicating an entire file system on untrusted storage servers in order to support a large number of readers. Farsite [13] uses a loosely coupled collection of insecure and unreliable machines, within an administrative domain, to collaboratively establish a virtual file server that is secure and reliable. It supports the file-I/O workload of desktop computers in a large company or university. All these systems expose a storage interface atop untrusted storage; however, none is suited for supporting semi-structured time series data.

These systems also do not provide configurability on where to store data: local versus remote for privacy concerns, partitioned across multiple storage providers for cost-effectiveness, and replicated across multiple providers for reliability and avoiding vendor lock-in (as in RACS [12] and HAIL [15]). Bolt does not need to deal with file system concurrency control and consistency issues, but instead leverages the nature of time series data to provide these capabilities.

A related line of work focuses on accountability and auditing (see Cloudproof [34]) of cloud behavior but again they are not suitable for the home setting and require server-side changes. Ming et al. [31] store patient health records (PHR) on the cloud and support attribute-based access control policies to enable secure and efficient sharing of PHR's. However, their system again requires cooperation from storage servers. Goh et al. [26] propose a security overlay called SiRiUS that extends local file systems with untrusted cloud storage systems with the support of data integrity and confidentiality. SiRiUS supports multiple writers and readers per file but does not provide any freshness guarantee.

Stream processing systems: Data access in database management systems is pull-based: a user submits a query to the system and an answer is returned. For applications that perform time series data analytics, traditional databases such as PostgresSQL, have made way for specialized time series databases like OpenTSDB [10] (which uses HBase as the backing datastore). In contrast, in stream-based systems, application's data is pushed to a processing system that must evaluate queries in real-time, in response to detected events — these systems offer *straight-through* processing of messages with optional storage. Some stream based systems are centralized (e.g., Aurora [18]), and others distributed (e.g., Borialis [11]), but they assume an environment in which all nodes fall under a single administrative domain. Bolt supports a pull-based model where there is no centralized query processing node, and end points evaluate the query and retrieve relevant data from storage servers in potentially different and untrusted administrative domains.

An early version of our work appears in a workshop paper [27] that outlined the problem and presented a basic design. The current paper extends the design (e.g., with chunking), implements real applications, and evaluates performance.

8 Discussion

We discuss two promising ways to extend Bolt to improve the overall reliability and sharing flexibility.

Relaxing the assumption on the trusted key server: Bolt's current design includes a trusted metadata/key server (i) to prevent unauthorized changes to the principal to public-key mappings and (ii) to prevent unauthorized updates (rollback) of the key version stored in each segment of a stream. The violation of (i) may trick the owner of a stream to grant access to a malicious principal whereas the violation of (ii) may cause the owner to use an invalid content key to encrypt data, potentially exposing the newly written content to principals whose access has been already revoked. Bolt also relies on the metadata/key server to distribute the keys and the metadata of a stream. Moving forward, we are looking into ways to minimize this trust dependency and improve the scalabilty of the metadata server. One approach is to replicate the information stored at the metadata server at $2f+1$ servers and go by majority, to tolerate up to f malicious servers. An alternate solution would be to employ a Byzantine quorum system, similar to COCA [40], to tolerate up to a third of servers being compromised at any given time. Partitioning can be used to implement a scalable distributed metadata service; For example, a set of geographically distributed metadata servers can be used to group the metadata for streams generated at homes in the same geographical locality.

Supporting finer-grained sharing: Currently, readers are granted access to the entire stream. Once their read access has been revoked, they cannot access any new segments of the stream created since the revocation although they could still access all the previous segments. Bolt can potentially support finer-grained read access, by creating a different key for each segment. This approach trades off metadata storage space for segment-level sharing.

9 Conclusion

Bolt is a storage system for applications that manipulate data from connected devices in the home. Bolt uses a combination of chunking, separation of index and data, and decentralized access control to fulfill the unique and challenging set of requirements that these applications present. We have implemented Bolt and ported three real-world applications to it. We find that for these applications, Bolt is up to 40 times faster than OpenTSDB while reducing storage overhead by 3–5x.

Acknowledgments We thank A.J. Brush and Khurshed Mazhar for being early adopters of Bolt; Rich Draves, Danny Huang, Arjmand Samuel, and James Scott for supporting this work in various ways; and our shepherd, Sharon Goldberg, the NSDI '14 reviewers, and John Douceur for feedback on drafts of this paper.

References

[1] Dropbox. https://www.dropbox.com/.

[2] Philips Hue. http://www.meethue.com/.

[3] The Internet of Things. http://share.cisco.com/internet-of-things.html/.

[4] Kwikset Door Locks. http://www.kwikset.com/.

[5] Mi Casa Verde. http://www.micasaverde.com/.

[6] Nest. http://www.nest.com/.

[7] Microsoft OneDrive. onedrive.live.com.

[8] Fast, portable, binary serialization for .NET. http://code.google.com/p/protobuf-net/.

[9] SmartThings. http://www.smartthings.com/.

[10] OpenTSDB. http://www.opentsdb.net/.

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

[12] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *SoCC*, 2010.

[13] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, Jon, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[14] B. J. Birt, G. R. Newsham, I. Beausoleil-Morrison, M. M. Armstrong, N. Saldanha, and I. H. Rowlands. Disaggregating categories of electrical energy end-use from whole-house hourly data. *Energy and Buildings*, 2012.

[15] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *CCS*, 2009.

[16] A. B. Brush, J. Jung, R. Mahajan, and F. Martinez. Digital neighborhood watch: Investigating the sharing of camera data amongst neighbors. In *CSCW*, 2013.

[17] K. E. Caine, C. Y. Zimmerman, Z. Schall-Zimmerman, W. R. Hazlewood, L. J. Camp, K. H. Connelly, L. L. Huber, and K. Shankar. DigiSwitch: A device to allow older adults to monitor and direct the collection and transmission of health information collected at home. *J. Medical Systems*, 35(5):1181–1195, 2011.

[18] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *VLDB*, 2002.

[19] C.-T. Chu, J. Jung, Z. Liu, and R. Mahajan. sTrack: Secure tracking in community surveillance. Technical Report MSR-TR-2014-7, Microsoft Research, 2014.

[20] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *NSDI*, 2012.

[21] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[22] S. Firth, K. Lomas, A. Wright, and R. Wall. Identifying trends in the use of domestic appliances from household electricity consumption measurements. *Energy and Buildings*, 2008.

[23] K. Fu. *Integrity and access control in untrusted content distribution networks*. PhD thesis, MIT, 2005.

[24] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *NDSS*, 2006.

[25] R. Geambasu, M. Balazinska, S. D. Gribble, and H. M. Levy. Homeviews: Peer-to-peer middleware for personal data sharing applications. In *SIGMOD*, 2007.

[26] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*, 2003.

[27] T. Gupta, A. Phanishayee, J. Jung, and R. Mahajan. Towards a storage system for connected homes. In *LADIS*, 2013.

[28] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, 2003.

[29] J. Z. Kolter, S. Batra, and A. Ng. Energy disaggregation via discriminative sparse coding. In *NIPS*, 2010.

[30] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[31] M. Li, S. Yu, K. Ren, and W. Lou. Securing personal health records in cloud computing: Patient-centric and fine-grained data access control in multi-owner settings. In *SecureComm*, 2010.

[32] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *TOCS*, 29(4), Dec. 2011.

[33] D. J. Nelson. Residential baseload energy use: Concept and potential for AMI customers. In *ACEEE Summer Study on Energy Efficiency in Buildings*, 2008.

[34] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *USENIX ATC*, 2011.

[35] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: semantic data management for the home. In *FAST*, 2008.

[36] J. Scott, A. J. B. Brush, J. Krumm, B. Meyers, M. Hazas, S. Hodges, and N. Villar. PreHeat: Controlling home heating using occupancy prediction. In *Ubicomp*, 2011.

[37] I. Shafer, R. R. Sambasivan, A. Rowe, and G. R. Ganger. Specialized storage for big time series. In *HotStorage*, 2013.

[38] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *CCSW*, 2010.

[39] R. P. Singh, S. Keshav, and T. Brecht. A cloud-based consumer-centric architecture for energy data analytics. In *e-Energy*, 2013.

[40] L. Zhou, F. B. Schneider, and R. Van Renesse. COCA: A secure distributed online certification authority. *TOCS*, 20(4), Nov. 2002.

Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications

James Mickens, Edmund B. Nightingale, Jeremy Elson, Krishna Nareddy, Darren Gehring
Microsoft Research

Bin Fan*
Carnegie Mellon University

Asim Kadav†, Vijay Chidambaram‡
University of Wisconsin-Madison

Osama Khan§
Johns Hopkins University

Abstract

Blizzard is a high-performance block store that exposes cloud storage to cloud-oblivious POSIX and Win32 applications. Blizzard connects clients and servers using a network with full-bisection bandwidth, allowing clients to access any remote disk as fast as if it were local. Using a novel striping scheme, Blizzard exposes high disk parallelism to both sequential and random workloads; also, by decoupling the durability and ordering requirements expressed by flush requests, Blizzard can commit writes out-of-order, providing high performance *and* crash consistency to applications that issue many small, random IOs. Blizzard's virtual disk drive, which clients mount like a normal physical one, provides maximum throughputs of 1200 MB/s, and can improve the performance of unmodified, cloud-oblivious applications by 2x–10x. Compared to EBS, a commercially available, state-of-the-art virtual drive for cloud applications, Blizzard can improve SQL server IOp rates by seven-fold while still providing crash consistency.

1 Introduction

As enterprises leverage cloud storage to process big-data workloads, there is increasing pressure to migrate traditional desktop and server applications to the cloud as well. However, migrating POSIX/Win32 applications to the cloud has historically required users to select from a variety of unattractive options. Databases like MySQL and email servers like Exchange can be trivially migrated to the cloud by running the server binaries inside of VMs that reside on cloud servers. Unfortunately, the storage abstractions exposed to those VMs lack the high performance and transparent scaling that are enjoyed by non-POSIX applications written for scale-out cloud stores like HDFS [6] and FDS [30]. For example, Azure and EC2 provide virtual disks that are backed by remote storage and that unmodified POSIX/Win32 applications can mount. However, each virtual drive only provides 50–250 MB/s of throughput [1, 28]; in contrast, a raw cloud store can provide more than a thousand MB/s to clients.

Datacenter operators provide "cloud-optimized" versions of a few popular applications like SQL and ActiveDirectory [3, 4, 15, 26, 27], implicitly acknowledging the difficulty of extracting cloud-scale performance from unmodified POSIX[1] applications. These cloud-optimized programs directly interface with the network storage using raw cloud APIs. This strategy provides higher performance than a naïve VM port, but such cloud-optimized applications offer fewer customization options than traditional POSIX/Win32 versions, making it difficult for users to tweak performance for individual workloads. More importantly, for the long tail of the application distribution, there are no pre-built, cloud-optimized versions. Large or technologically savvy companies may have the resources to write cloud-optimized versions of their applications, but for safety reasons, datacenter operators do not provide external developers with full access to the raw cloud APIs that are needed to maximize performance. Even if customers had such access, many customers would prefer to simply deploy their standard binaries to the cloud and automatically receive fast, scalable IO.

Unfortunately, desktop and server applications have significantly different IO patterns than traditional cloud-

*Work completed as a Microsoft intern; now at Google.
†Work completed as a Microsoft intern; now at NEC Labs.
‡Work completed as a Microsoft intern.
§Work completed as a Microsoft intern; now at Twitter.

[1] For conciseness, we use "POSIX" to mean "POSIX/Win32" in the rest of this paper.

scale applications. A typical MapReduce-style workload issues large, sequential IOs, but POSIX applications issue small, random IOs that are typically 32–128 KB in size [25, 41]. POSIX applications also require finer-grained consistency than many big-data workloads. For an application like an Internet-scale web crawl, append-at-least-once semantics [14] are often reasonable, and losing MBs of append-only data may be an acceptable risk for higher throughput; in contrast, for a POSIX database, compiler, or email server that is randomly accessing small blocks, the loss or duplication of just a few adversarially-chosen blocks can result in metadata inconsistencies and catastrophic data loss. To enforce fine-grained consistency, POSIX applications use disk flushes to order writes [8, 33, 39], but these flushes introduce write barriers. In the context of cloud storage, these barriers make it difficult for POSIX applications to issue large numbers of parallel writes to remote cloud disks. Lower disk parallelism leads to lower performance.

In this paper, we introduce Blizzard, a high-performance block store that exposes unmodified, cloud-oblivious applications to fast, scalable cloud storage. From the perspective of a client application, Blizzard's virtual disk looks like a standard SATA drive. However, Blizzard translates block reads and writes to parallel IOs on remote cloud disks, transparently handling the removal, addition, or failure of those remote disks.

Blizzard is not the first system to propose a virtual disk backed by remote storage [2,11,24]. However, Blizzard's virtual drive has several unique characteristics:

Locality-oblivious, full-bisection bandwidth block access: Blizzard is built atop a CLOS network with no oversubscription [17], i.e., arbitrary client/server pairs can exchange data at full NIC speeds without inducing network congestion. Blizzard also pairs each server disk with enough network bandwidth to read and write that disk at full sequential speed. These properties mean that clients can stripe their data across arbitrary remote disks in a locality-oblivious manner. This simplifies Blizzard's striping algorithm and permits very aggressive sharding (which results in better performance, since spreading data over more disks increases IO parallelism).

Nested striping: POSIX applications typically issue small, random IOs, but even when their IOs are large, client file systems often break such large operations into smaller pieces. A naïve stripe of a virtual drive across remote disks will cause *IOp convoy dilation*—as the small requests belonging to a single large operation travel from the client to a remote disk, the inter-request spacing will increase due to network jitter and scheduling vagaries on the client and the server. The longer the dilation, the less likely that the remote disk can use a single seek to handle all of the adjacent disk requests in the convoy. Blizzard uses a novel striping scheme called *nested striping* which avoids these problems. Nested striping ensures that convoy blocks are spread across multiple disks in parallel. This amortizes the seek costs for the individual blocks, and globally acts to prevent disk hotspots.

Fast flushes with prefix write commits: POSIX applications use disk flushes to order writes and provide crash consistency. However, such flushes restrict IO parallelism, and massive IO parallelism is the primary technique that clients must leverage to unlock cloud-scale IO performance. When Blizzard's virtual disk receives a flush request, it immediately acknowledges the flush to the client application, even though Blizzard has not made writes from that flush epoch durable. Asynchronously, the virtual drive issues writes in a way that respects *prefix epoch semantics*. If the client or the virtual disk crashes, the disk will always recover to a consistent state in which writes from different flush epochs will never be intermingled—all writes up to some epoch $N-1$ will be durable; some writes from epoch N may be durable; and all writes from subsequent epochs are lost. Blizzard's asynchronous writes lengthen the window for potential data loss, but they permits much higher levels of write performance. This approach also reduces the penalty for N-way data replication, since acknowledging a write no longer proceeds at the pace of the slowest replica for that write. Prior work has shown how prefix semantics can be added to the ext4 file system [7], but Blizzard shows how such semantics can be added at the disk level, *in a file-system agnostic manner*, and in a way that also provides high performance to applications that bypass the file system entirely and issue raw disk IOs.

Blizzard has several additional features, like support for disconnected operation[2], and tunable levels of disk parallelism (§2.4).

We have built a Blizzard prototype consisting of 1,200 disks and 150 servers. Using this prototype, we demonstrate that Blizzard can improve the performance of unmodified IO-intensive applications by 2x–10x. Importantly, our Blizzard prototype coexists alongside our FDS [30] deployment, using the same servers, disks, and networking equipment. FDS is optimized for large, business-scale computations. Thus, using Blizzard, cloud providers can leverage a single set of cluster hardware for both big-data computations and POSIX

[2]Not discussed further due to space constraints.

applications, reducing hardware outlays while consolidating administrative costs and providing fast, scalable IO to both types of workloads. From the perspective of developers, Blizzard allows POSIX applications to receive cloud-scale performance and availability without requiring datacenter operators to expose their raw, unsafe cloud APIs—instead, developers simply write applications under the assumption that (virtual) disk drives are extremely fast.

2 Design

Blizzard has two high-level goals. First, it has to run unmodified, cloud-oblivious POSIX applications on the same cloud infrastructure used by traditional big-data applications. Second, it must provide those POSIX applications with the storage performance, scalability, and availability that big-data programs receive. By "cloud-level performance," we mean that a single client should be able to issue hundreds of MBs of IO requests every second. By "cloud-level scalability and availability," we mean that client storage should transparently improve as the cloud operator adds more remote disks or better networking capacity. Also, administrative efforts that help big-data applications should also improve unmodified POSIX applications.

To satisfy these design goals, Blizzard must efficiently handle two aspects of POSIX workloads:

- POSIX applications typically generate small, random IOs between 32 KB and 128 KB in size [25, 41]. To offer high performance to such seek-bound workloads, Blizzard needs to expose applications to massive disk parallelism. This allows applications to issue multiple operations simultaneously and overlap the seek costs.
- POSIX applications use the `fsync()` system call to control the ordering and durability of writes, ensuring consistency after crashes [7, 8, 33, 39]. An `fsync()` call generates a disk flush, and the disk flush acts as a write barrier, preventing writes issued after the flush from completing before all previous writes are durable. These write barriers limit disk parallelism, which results in poor performance. Thus, Blizzard needs to handle `fsync()` calls in a way that preserves notions of write ordering, but does not require clients to wait for synchronous disk events before issuing new writes.

In Section 2.3, we describe how Blizzard leverages full-bisection bandwidth networks to aggressively stripe each client's data across a large number of disks. In the absence of flush requests, this scheme would suffice to provide clients with massive disk parallelism. However, POSIX applications commonly issue flush requests. Thus, as described in Section 2.5, Blizzard uses eventual durability semantics [7] to remove synchronous disk operations from the flush path. When a client issues a flush, Blizzard records ordering information about pending writes; then, Blizzard immediately acknowledges the flush request. Asynchronously, Blizzard writes data to remote disks in a way that respects the client's ordering constraints, and allows the client to recover a consistent view after a crash. In the extreme, Blizzard can issue writes from multiple flush periods completely out-of-order (§2.5.3), removing all synchronization constraints involving writes, but still preserving consistency.

2.1 Blizzard's Storage Abstraction

Frameworks like pNFS [38] and BlueSky [42] expose cloud storage to cloud-oblivious applications by translating (say) NFS operations into operations on cloud disks. However, these systems lock applications into a particular set of file semantics which will not be appropriate for all applications. For example, some applications desire NFS's close-to-open consistency semantics, but other applications require POSIX-style consistency in which a newly written block is immediately visible to readers of the enclosing file [20]. Since all file systems eventually issue reads and writes to a block device (and since some applications issue raw disk commands), we decided to implement Blizzard as a virtual block device which stripes data across remote disks. This allowed us to support heterogeneous POSIX and Win32 file systems like ext3 and NTFS; it also allowed us to expose fast storage to applications like databases that issue raw block-level IOs.

2.2 The Low-level Storage Substrate

Blizzard stripes each virtual drive across several remote physical disks. As the striping factor increases, the virtual drive benefits from greater spindle parallelism (and thus higher IO performance). However, a traditional oversubscribed network constrains how aggressively Blizzard can stripe. In an oversubscribed network, the available cross-rack bandwidth is lower than the available intra-rack bandwidth; thus, for a system with medium-to-high utilization, clients access rack-local disks faster than they access disks in external racks. If Blizzard restricted each virtual drive to use rack-local disks, this would limit spindle parallelism, constrain the total capacity of the virtual drive, and makes job allocation more difficult, since a single job could not harness idle disks spread across multiple racks. However, if Blizzard allowed a virtual disk to span racks, contention in the oversubscribed cross-rack links would prevent the client from fully utilizing rack-external disks.

To avoid this dilemma, Blizzard uses Flat Datacenter Storage (FDS) as its low-level storage substrate [30].

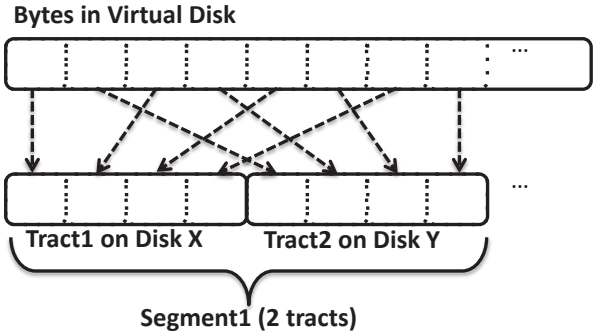


Figure 1: An example of nested striping with a segment size of 2. Virtual disk blocks are striped across tracks, and tracks are scattered across disks.

FDS is a datacenter-scale blob store that connects all clients and disks using a network with full-bisection bandwidth, i.e., no oversubscription [17]. FDS also provisions each storage server with enough network bandwidth to match its aggregate disk bandwidth. For example, a single physical disk has roughly 128 MB/s of maximum sequential access speed. 128 MB/s is 1 Gbps, so if a storage server has ten disks, FDS provisions that server with a 10 Gbps NIC; if the server has 20 disks, it receives two 10 Gbps NICs. The resulting storage substrate provides a locality-oblivious storage layer—any client can access any remote disk at maximum disk speeds[3].

A Blizzard virtual disk is backed by a single FDS blob (although Blizzard does not use a linear mapping between a virtual disk address and the corresponding byte in the FDS blob (§2.3 and §2.5)). FDS breaks each blob into 8 MB segments called tracts, and uses these tracts as the striping unit. Blizzard typically instructs FDS to stripe a blob across 64 or 128 physical disks; optionally, FDS performs N-way replication of each tract.

2.3 Data Placement

FDS provides Blizzard with some nice properties out-of-the-box, and allows Blizzard to satisfy the design goal of running on the same infrastructure that supports traditional big-data applications. However, FDS is optimized for large, sequential IOs. In particular, FDS divides each FDS blob into a series of large tracts. A tract resides in a contiguous location on a particular disk, but FDS scatters a blob's tracts across a large number of different disks. Tracts are 8 MB in size, and FDS's primary read/write interface works at the granularity of tracks. POSIX IOs are typically much smaller than 8 MB. Thus, we had to devise a way for Blizzard's virtual disk to map these small, random IOs onto FDS tracts (which FDS would then map to remote disks).

[3]This assumes that RTTs between clients and storage servers are much smaller than a seek time. We explore the impact of network latency in Section 4.6.

FDS natively supports a raw, low-level interface that lets applications read and write data in chunks that are smaller than a tract. This interface allows applications to process small pieces of tract metadata without having to read or write entire 8 MB tracts. Our first prototype of Blizzard's virtual disk used this interface, and a very simple block mapping, to translate virtual disk addresses to tract-level offsets. In this initial prototype, the virtual disk split its linear address space into contiguous, tract-sized chunks, and assigned each chunk to a separate FDS tract. A virtual disk IO with offset `byteOffset` would go to tract `byteOffset/TRACT_SIZE_BYTES`. This mapping was straightforward, but it led to disappointing performance. A convoy of sequential IOs often hit the same tract (and thus the same remote disk), eliminating opportunities for disk parallelism. Even worse, sequential convoys often experienced *dilation*. Even if the client used a tight loop to issue writes to adjacent offsets, the temporal spacing between those operations often grew as the operations traveled from client to remote disk. Client-side scheduling jitter increased the spacing, as did random network delays and scheduling jitter on the remote server. Thus, a sequential convoy that initially had little inter-request spacing at the client often arrived at the remote hard disk with larger inter-request gaps. In many cases, this prevented the remote disk from efficiently servicing the entire convoy with a single seek. Instead, the convoy operations were handled with multiple seeks and rotational delays, increasing the total IO latency for *all* operations on that disk.

Given these observations, we designed a new mechanism called *nested striping* that maps linear byte ranges to FDS tracts. We define a *segment* as a logical group of one or more tracts; a segment of N bytes contains striped data for a linear byte range of N bytes. Figure 1 demonstrates nested striping when each segment contains two tracts. Intuitively, increasing the segment size allows Blizzard to provide greater disk parallelism for sequential workloads. For example, a segment size of one restricts sequential IOs to one disk. As shown in Figure 1, a segment size of two spreads sequential IOs across two disks. Figure 1 also demonstrates why we call this striping scheme "nested": Blizzard stripes blocks across FDS tracts, and FDS distributes the tracts in a blob across many remote disks. By default, our Blizzard implementation uses a segment size of 128 tracts.

Experiments show that nested striping dramatically decreases the impact of convoy dilation. Using a segment size of 128 provides over 1000 MB/s of sequential read throughput (§4.1). In contrast, using a segment size of one tract results in sequential read throughput of 222 MB/s.

2.4 Role-based Striping

Up to this point, we have assumed that Blizzard is deployed in a shared, public cloud that is utilized by multiple customers. However, Blizzard can also be used in private, enterprise-local clouds. Indeed, one deployment model for Blizzard is a *building-scale deployment*—all of the rooms in the building are connected via a full-bisection bandwidth network, and all desktop and server machines use Blizzard virtual drives to store and manipulate data. Datacenters already deploy full-bisection networks at the scale of thousands of machines [17], so this deployment model is technically feasible, and it would allow an enterprise to consolidate storage and IT effort for a particular building.

In such a building-scale deployment, different users will have different performance needs. For example, a programmer that runs large compilations requires better storage performance than a receptionist who mainly sends emails and performs word processing. For building-wide Blizzard deployments, segment sizes offer a convenient knob that administrators can use to control the amount of disk parallelism that Blizzard exposes to different users with different roles.

2.5 Write Semantics

Blizzard's virtual disk provides three types of data consistency: write-through commits, flush epoch commits with fast acknowledgments, and out-of-order commits with fast acknowledgments. We describe each approach below. Note that, when we say that Blizzard "acknowledges" an operation to a "client," the client is either a file system or an application that issues raw disk IOs. A write request or a flush request is "acknowledged" when that operation returns to the client.

2.5.1 Write-through

In this mode, Blizzard does not acknowledge a virtual disk write until the associated FDS operation has become durable on the relevant remote disk. This approach provides the smallest window of potential data loss if a crash occurs. However, write-through consistency provides the lowest performance, since a thread which issues a blocking write must wait for that write to become durable before issuing additional IOs.

2.5.2 Flush epoch commits with fast acks

Let a *flush epoch* be a period of time between two flush requests from the client. Each flush epoch contains one or more writes. A flush epoch is *issued* if all of its writes have been sent to remote disks. The epoch is *retired* if all of those writes have been reported as durable by the remote disks.

Setup: Blizzard maintains a counter called `epochToIssue`; this counter starts at 0 and represents which writes Blizzard can send to FDS. Blizzard maintains another counter called `currEpoch` that also starts at 0. This counter represents the total number of flush requests that the client has issued. As explained below, `currEpoch` will often be greater than `epochToIssue`.

Acknowledging writes: When Blizzard receives a write request or a flush request, it immediately acknowledges that operation, allowing the client to quickly issue more IO requests. If the incoming operation was a flush, Blizzard increments `currEpoch` by 1. If the operation was a write, Blizzard tags the write with the `currEpoch` value and places the write request in a queue that is ordered by epoch tags.

Draining the write queue: Once a new write is enqueued and acknowledged to the client, Blizzard tries to issue enqueued writes to remote disks. Blizzard iterates from the front of the write queue to the end, i.e., from the oldest unretired epoch to the newest. If the currently examined write is from `epochToIssue`, Blizzard dequeues the write and issues it immediately; otherwise, Blizzard terminates the queue traversal. Later, when a write from epoch *N* completes, Blizzard checks whether epoch *N* has now retired. If so, Blizzard increments `epochToIssue` and tries to release new writes from the head of the write queue.

When Blizzard issues a write, it removes it from the write queue. However, Blizzard keeps the write in a separate cache until the write is durable. Meanwhile, if a read arrives for the write's byte range, Blizzard services the read using the cached, fresh data, instead of issuing a read to the underlying remote disk and possibly getting old data.

Consistency semantics: In this consistency scheme, Blizzard treats a flush as an ordering constraint, but not a durability constraint; using the terminology of optimistic crash consistency, Blizzard provides "eventual durability" [7]. This means that Blizzard issues writes in a way that respects flush-order durability, but a flush epoch may retire at an arbitrarily long time after the flush was acknowledged to the client. Indeed, the epoch may never retire if the client crashes before it can issue the associated writes. However, the rebooted client is guaranteed to see a consistent prefix of all writes that were acknowledged as flushed; this suffices for many applications [9].

2.5.3 Out-of-order commits with fast acks

To maximize the rate at which writes are issued, Blizzard defines a scheme that allows writes to be acknowledged

immediately *and* issued immediately, regardless of their flush epoch. This means that writes may become durable out-of-order. However, Blizzard enforces prefix consistency using two mechanisms. First, Blizzard abandons nested striping and uses a log structure to avoid updating blocks in place; thus, if a particular write fails to become durable, Blizzard can recover a consistent version of the target virtual disk block. Second, even though Blizzard issues each new write immediately, Blizzard uses a deterministic permutation to determine which log entry (i.e., which <tract,offset>) should receive the write. To recover to a consistent state after a crash, the client can start from the last checkpointed epoch and permutation position, and roll the permutation forward, examining log entries and determining the last epoch which successfully retired.

Setup: Let there be V blocks in the virtual disk, where each block is of equal size, a size that reflects the average IO size for the client (say, 64 KB or 128 KB). The V virtual blocks are backed by $P > V$ physical blocks in the underlying FDS blob. Blizzard treats the physical blocks as a log structure. Blizzard maintains a `blockMap` that tracks the backing physical block for each virtual block. Blizzard also maintains an `allocationBitMap` that indicates which physical blocks are currently in use. When the client issues a read to a virtual block, Blizzard consults the `blockMap` to determine which physical block contains the desired data. Handling writes is more complicated, as explained below.

Blizzard maintains a counter called `currEpoch`; this counter is incremented for each flush request, and all writes are tagged with `currEpoch`. Blizzard also maintains a counter called `lastDurableEpoch` which represents the last epoch for which all writes are retired.

The virtual-to-physical translation: When Blizzard initializes the virtual disk, it creates a deterministic permutation of the physical blocks. This permutation represents the order in which Blizzard will update the log. For example, if the permutation begins 18, 3,..., then the first write, *regardless of the virtual block target*, would go to physical block 18, and the second write, regardless of the virtual block target, would go to physical block 3. Importantly, Blizzard can represent a permutation of length P in $O(1)$ space, not $O(P)$ space. Using a linear congruential generator [44], Blizzard only needs to store three integer parameters (a, c, and m), and another integer representing the current position in the permutation. As we will describe later, the serialized permutation will go into the checkpoints that Blizzard creates.

Handling reads is simple: when the client wants data from a particular virtual block, Blizzard uses the `blockMap` to find which physical block contains that data; Blizzard then fetches the data. Handling writes requires more bookkeeping. When a write arrives, Blizzard iteratively calls the deterministic permutation function, and immediately sends the write to the first physical block that is not marked in the `allocationBitMap` as used. However, once the write is issued, Blizzard does *not* update the `allocationBitMap` or the `blockMap`—those structures are reflected into checkpoints, so they can only be updated in a way that respects prefix semantics. So, after Blizzard issues the write, it places the write in a queue. Blizzard uses the write queue to satisfy reads to byte ranges with in-flight (but possibly non-durable) writes. When a write becomes durable, Blizzard checks whether, according to the permutation order, the write was the oldest unretired write in `lastDurableEpoch`+1. If so, Blizzard removes the relevant write queue entry, and updates `blockMap` and `allocationBitMap`. Otherwise, Blizzard waits for older writes to commit first. Once all writes in the associated epoch are durable, Blizzard increments `lastDurableEpoch`.

When Blizzard issues a write to FDS, it actually writes an *expanded block*. This expanded block contains the raw data from the virtual block, as well as the virtual block id, the write's epoch number, and a CRC over the entire expanded block. As we explain below, Blizzard will use this information during crash recovery.

If the client issues a write that is smaller than the size of a virtual block, Blizzard must read the remaining parts of the virtual block before calculating the CRC and then writing the new expanded block. This read-before-write penalty is similar to the one suffered by RAID arrays that use parity bits. This penalty is suffered for small writes, or for the bookends of a large write that straddles multiple blocks. For optimal performance, Blizzard's virtual block size should match the expected IO size of the client. For example, POSIX applications like databases and email servers often have a configurable "page size"; these applications try to issue reads and writes that are integral multiples of the page size, so as to minimize disk seeks. For these applications, Blizzard's virtual block size should be set to the application-level page size. For other applications that 1) frequently generate writes that are not an even multiple of Blizzard's block size, or 2) generate writes that are not aligned on Blizzard's block boundaries, Blizzard should be configured to use write-through mode, or fast acknowledgment mode with nested striping (§4.7).

Checkpointing: Periodically, the client checkpoints the `blockMap`, the `allocationBitMap`, the four permutation parameters, `lastDurableEpoch`, and a CRC over the preceding quantities. For a 500 GB virtual disk, the checkpoint size is roughly 16 MB. Blizzard does not update the checkpoint in place; instead, it reserves enough space on the FDS blob for two checkpoints, and alternates checkpoint writing between the two locations.

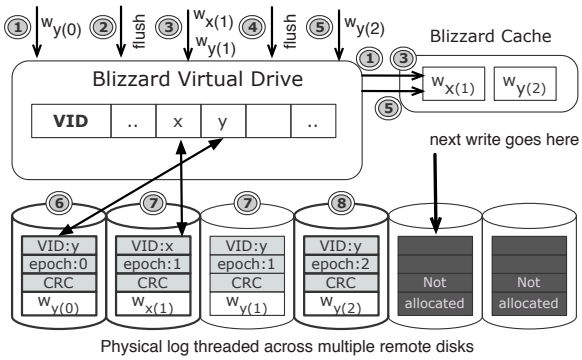


Figure 2: An example of Blizzard's log-based, out-of-order commit scheme. Disks with thick borders contain durable writes. See the main paper text for an explanation of what happens at each time step.

Recovery: To recover from a crash, Blizzard loads the two serialized checkpoints and initializes itself using the checkpoint with the highest `lastDurableEpoch` and a valid CRC. Blizzard then rolls forward from the permutation position in the checkpoint, scanning physical blocks in permutation order. Since Blizzard issues log writes in epoch order, the recovery scan will process writes in epoch order. If the `allocationBitMap` says that the current physical block is in use, Blizzard inspects the next physical block in the permutation. If the `allocationBitMap` says that the current physical block is *not* in use, Blizzard inspects the physical block's epoch number. If the number is less than `lastDurableEpoch`, Blizzard terminates roll-forward. If the CRCs from all of the physical block's replicas are inconsistent or do not match each other, Blizzard terminates roll-forward. Otherwise, Blizzard updates the `allocationBitMap` to mark the physical block as used (and the old physical block for the virtual block as unused); Blizzard also updates the relevant `blockMap` entry to point to the physical block. Finally, Blizzard sets `lastDurableEpoch` to the epoch value in the physical block. The permutation position at which roll-forward terminates will be the position to which Blizzard sends the next write.

Example: Figure 2 provides an example of Blizzard's log-based, out-of-order commits. For simplicity, this example uses the permutation generator `(writeNumber % logSize)`, such that physical log entries are scanned in linear order, from left to right. At time (1), a write arrives for virtual block `Y`. Blizzard issues the write to remote storage immediately, and places the write in an in-memory cache, so that reads for `Y`'s data can access the fresh data without having to wait for the remote write to finish. A flush request arrives at (2), which Blizzard acknowledges immediately. The current flush epoch is now 1. At (3), writes to virtual blocks `X` and `Y` arrive. Blizzard acknowledges those writes immediately, issuing them in parallel to the next positions in the log, and updating the write cache entries for blocks `X` and `Y` (in the latter case, overwriting the old cache value for `Y`). At (4), another flush arrives, and Blizzard increments the flush epoch to 2. At (5), another write arrives for `Y`, causing Blizzard to issue a new write to the next position in the log, and updating `Y`'s write cache value. At (6), the first write to `Y` becomes durable on a remote disk, causing Blizzard to update the `blockMap` entry for `Y` to point to that log entry. At time (7), the write to `X` becomes durable, and Blizzard updates the `blockMap` appropriately. However, at time (7), the second write to `Y` has not committed. At time (7), the client takes a checkpoint (note that the last permutation index that the client knows is durable is the second log entry). At time (8), the client learns that the third write to `Y` is durable. However, since the second write to `Y` is not durable yet, the client does not change the `blockMap`–thus, virtual block `Y` still points to the write from epoch 0.

Suppose that the client crashes immediately after it makes a checkpoint at (7). Further suppose that this crash prevents the write to the third physical block from becoming durable, e.g., because the client needed to retry the write, but crashed before it could do so. After the client reboots, it looks at the checkpoint and extracts the last permutation index known to be durable (log entry two). The client then rolls forward through the log in permutation order. The client examines the third physical log entry and sees that it is marked as unused by the `allocationBitMap`. The client examines the entry's epoch number and CRC. Since the associated write failed, one or both of those quantities will have invalid values. At this point, the client stops the roll-forward. Even though the write to the fourth log block completed, that write is lost to the client. However, the client has recovered to a prefix-consistent view of the virtual block `Y` (and the rest of the disk).

IOp dilation: Even though log-based consistency does not use nested striping, the linear congruential generator produces striping patterns that "jump around" enough to prevent convoy dilation. Adversarial write patterns can still result in dilation, but such patterns are rare in practice.

2.6 Application-perceived Consistency

In write-through mode, Blizzard minimizes application-perceived data loss. Since all writes are synchronous and go directly to remote disks, writes can only be lost if the application transfers write data to the virtual drive, but

the drive or the client machine crashes before Blizzard can write the data to the remote disks. Once Blizzard has acknowledged a write operation to the client, the client knows that the write data is durable.

For the fast acknowledgment schemes, Blizzard trades higher performance for an increased possibility of data loss. In these schemes, if a crashed client issued writes belonging to flush epochs $F_0, F_1, \ldots, F_N$, the client will recover to a virtual disk which contains all writes from epochs $F_0 \ldots F_R$; some, all, or no writes from epoch F_{R+1}; and no writes from epochs $F_{R+2} \ldots F_N$. Denote these three write segments as the preserved epochs, the questionable epoch, and the disavowed epochs, respectively. With traditional flush semantics for a local physical disk, there are no disavowed epochs—clients cannot issue new writes across a flush barrier if prior writes are outstanding, so, at worst, a client can lose some or all writes from its last, questionable epoch. With Blizzard's fast acknowledgment schemes, $N-R$ may be greater than zero, i.e., there may be a questionable epoch and disavowed epochs.

When operating in fast acknowledgment mode, Blizzard can minimize data loss (i.e., $N-R$) by issuing writes as quickly as possible; the log-based write scheme does precisely this. However, for all three of Blizzard's write schemes, unmodified POSIX file systems and applications will always recover to a prefix-consistent version of the Blizzard drive. Using fast acknowledgments, applications are more likely to share acknowledged (but currently non-durable) writes to external parties, meaning that, if the application crashes, it may not be able to recover those externalized writes from the Blizzard drive. In practice, we do not believe that this is a problem, since many users are willing to receive high performance and crash consistency in exchange for potentially externalizing non-durable data [9]. Users that wish to never externalize non-durable data can run Blizzard in write-through mode.

2.7 Server-side Failure Recovery

Blizzard relies on FDS to recover failed tracts belonging to a virtual drive. However, FDS clients are responsible for retrying aborted writes caused by remote disk failures. Thus, if Blizzard detects that an FDS write was unsuccessful, it must contact the FDS metadata server, download the new mapping between tracts and remote disks, and retry the write. If replication is enabled, Blizzard also ensures that each write to a virtual block results in R successful writes to the R replica disks.

2.8 Coexisting Workloads

Blizzard is built atop FDS, and both systems use a shared physical infrastructure of servers, disks, and network equipment. The network provides full-bisection bandwidth, and FDS uses a request-to-send/clear-to-send mechanism which ensures that senders cannot overrun receivers [30]. Thus, neither Blizzard workloads nor native FDS applications can induce network congestion at the core or the edge. This lack of congestion does not guarantee any notion of application-level "network fairness," so operators that desire such properties must use client-side mechanisms like admission control or rate limiting.

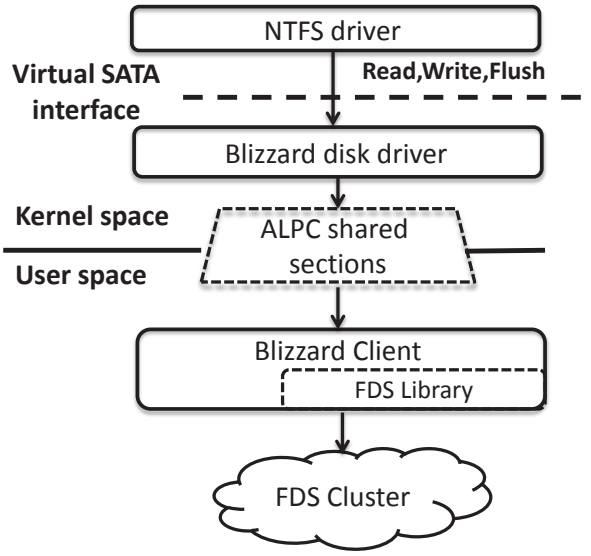


Figure 3: A Blizzard virtual disk on Windows.

Both Blizzard and FDS use aggressive, randomized striping across disks. As the aggregate client IO pressure increases, service times at each disk degrade gracefully, since the workload is spread evenly across each disk (§4.5). In principle, a single disk can store data for both Blizzard applications and native FDS applications. In practice, we typically allocate a single disk to either Blizzard or native FDS applications, but not both. POSIX applications often require low-latency IO in addition to high throughput, so our allocation scheme prevents small POSIX IOs from getting queued behind the much larger IOs from big-data applications.

3 Implementation

Figure 3 shows the architecture for our Blizzard implementation on Windows. The virtual disk contains two pieces: a kernel-mode SATA driver, and a user-mode component which links to the FDS client library. File systems (and applications which issue raw disk IO) send IO request packets (IRPs) to the SATA driver. The driver forwards these requests to the user-mode client, which translates the requests into the appropriate FDS operations. For IRPs that correspond to reads and writes, Blizzard issues reads or writes to the appropriate remote disks. Once the client receives a response from a

remote disk, it hands the response to the kernel mode driver, which then informs the requesting application that the IRP has completed. To minimize the overhead of exchanging data across the user-kernel boundary, Blizzard uses Window's Advanced Local Procedure Calls (ALPC); ALPC provides zero-copy IPC using shared memory pages.

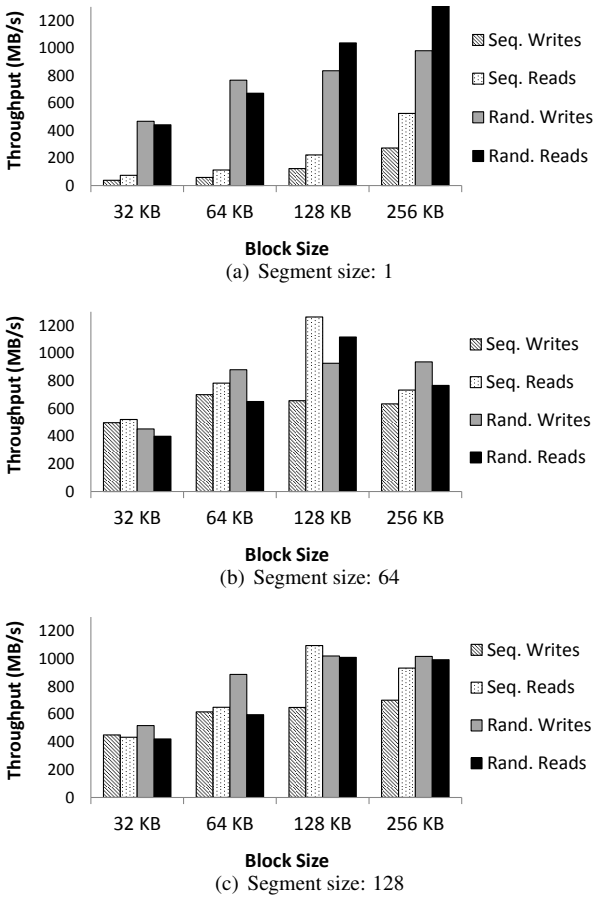
To maximize performance, Blizzard uses asynchronous interfaces to exchange data between the kernel driver, the FDS client, and the FDS cluster. The user-mode component of the virtual disk is multi-threaded, and it maximizes throughput by handling multiple kernel requests and FDS operations in parallel. Such high levels of concurrency and asynchrony make it tricky to preserve the prefix semantics discussed in Section 2.5. In terms of wall-clock time, reads and writes arrive at the user-mode component in the order that the kernel issued them. However, the user-mode component handles each read and write in a separate thread; lacking guidance from the kernel, writes might issue to FDS in a nondeterministic fashion, based on whichever user-mode write threads happened to grab more CPU time. To allow the user-mode component to implement prefix semantics, the kernel driver tags each write request with its flush epoch, its sequence number within that epoch, and the maximum sequence number for writes from the *previous* epoch. This way, different user-mode threads can use lightweight interlocked-increment operations on integers to track the number of writes that have issued for each epoch. Blizzard does eventually require heavyweight locking to add writes to the write queue (§2.5), but this locking takes place in the latter part of the write path, leaving the first part contention-free.

4 Evaluation

In this section, we use a variety of experiments to demonstrate that Blizzard provides low-latency, high-throughput IO to unmodified, cloud-oblivious applications. Unless stated otherwise, when we refer to "Blizzard in fast acknowledgment mode," we refer to the second consistency scheme in Section 2.5, not the log-based approach.

4.1 Microbenchmarks

Figure 4 depicts the raw performance of a Blizzard virtual disk backed by 128 remote disks and using single replication. To generate these results, we ran a custom client program that issued asynchronous, block-level reads and writes to the virtual disk as quickly as possible. Blizzard was configured in write-through mode, to identify the steady-state performance that Blizzard could provide to a completely IO-bound client. The results show that, depending on the block size and the segment size, Blizzard can provide throughputs of 700 MB/s for sequential writes, and over 1000 MB/s for sequential reads, random reads, and random writes.



(a) Segment size: 1

(b) Segment size: 64

(c) Segment size: 128

Figure 4: Throughput microbenchmarks.

From the perspective of a client, increasing the segment size is roughly analogous to adding disks to a private RAID-0 array—it increases the number of disks that a client can access in parallel, improving both performance and load balancing. As shown in Figure 4(a), small segment sizes lead to poor sequential IO performance due to convoy dilation effects (§2.3). However, even for a segment size of one, Blizzard services *random* IOs at 400 MB/s or faster. This is because, at any given time, a random workload accesses more disks than a sequential workload, improving disk parallelism (and thus aggregate throughput). In the rest of this section, unless otherwise specified, all experiments use a segment size of 128, and 128 backing disks.

Increasing the block size beyond 32 KB improves performance, since disks can fetch more data per seek. However, increasing the block size beyond 128 KB leads to diminishing throughput returns.

Figure 5 compares Blizzard's write latency under several consistency settings and replication levels. For this experiment, we intentionally included some old, slow

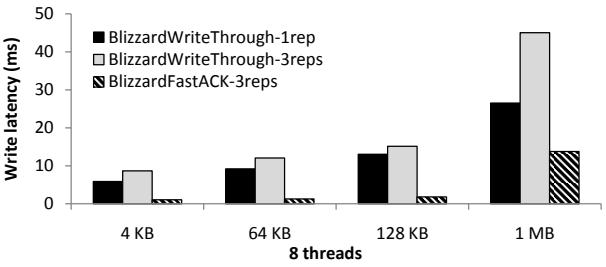


Figure 5: Write latency: Blizzard in write-through mode (1 and 3 replicas) and Blizzard in fast acknowledgment mode (3 replicas).

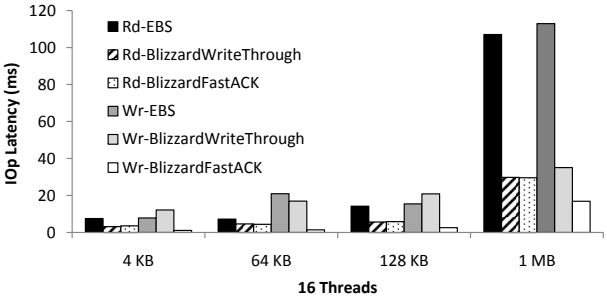


Figure 6: IOp latency: EBS and Blizzard.

disks that we typically exclude from production workloads. Figure 5 shows that fast acknowledgments dramatically reduce the cost of data redundancy—despite the presence of known-slow disks, the write latency of triple replication with fast acknowledgments is 2x–5x lower than the latency of single replication with write-through semantics. Blizzard clients obviously cannot issue an infinite number of low-latency IOs, since, at some point, some resource (e.g., the network or a remote disk) will become saturated. However, our results show that, until that point is reached, Blizzard's fast acknowledgments provide very low-latency IOs.

4.2 Blizzard vs. EBS

In this section, we compare Blizzard's virtual drive to Amazon's Elastic Block Store (EBS) drive. Our EBS deployment used one Amazon EC2 instance (an EBS-optimized m1.xlarge) which had 4 CPUs and 15 GB of memory, and which ran 64-bit Windows 2008 R2 SP1 Datacenter Edition. The virtual EBS drive was backed by 12 disks, each of which was provisioned for 100 IOps and 10 GB of storage; we constructed a RAID-0 striped volume as the backing storage for the EBS drive. The disks and the EC2 instance were connected by a 1 Gbps network connection. To provide a fair comparison to Blizzard, we configured Blizzard's virtual drive to use 12 backing disks, and we used FDS's built-in rate-limiter to restrict the Blizzard client to 1 Gbps of network bandwidth. The Blizzard client had 4 CPUs and 12 GB of RAM, similar to the EBS client.

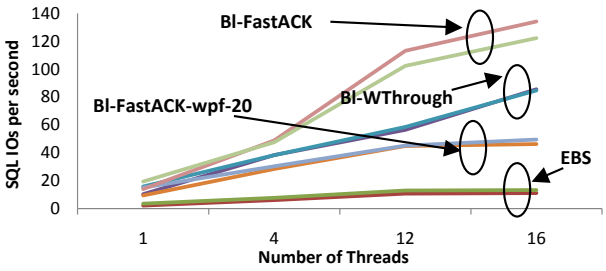


Figure 7: SQL read and write IOs per second: EBS, Blizzard in write-through mode, Blizzard in fast acknowledgment mode, and Blizzard in fast acknowledgment mode with forced epochs every 20 writes. Each pair of circled lines represents the read and write speeds for a particular configuration.

Using a multithreaded synthetic load generator, we tested IO latency in EBS, Blizzard in write-through mode, and Blizzard with fast acknowledgments enabled. Since the load generator did not generate flush requests, the latter Blizzard configuration provided good performance, but not prefix consistency; this configuration is still a useful one to investigate, because many people disable disk flushes for the sake of performance [7, 31]. Figure 6 shows that the read latencies for both Blizzard schemes were 2x–4x lower than EBS's latency. Both Blizzard schemes had similar read latencies because delayed durability tricks do not directly affect the completion times of reads (although a client may generate more reads per second if writes require less time to complete).

For write latencies, Blizzard with fast acknowledgments was 7x–14x faster than EBS. Blizzard in write-through mode was essentially equivalent to EBS for small to medium operations, but much faster for 1 MB writes. It is unclear to us why EBS was so much slower for large writes. Throughput tests (which we elide due to space constraints) showed that, even for large IO sizes, EBS only utilized about 80% of the available network bandwidth; thus, the EBS client may be exchanging control traffic with other nodes that we cannot see.

Figure 7 shows the performance of `sqliosim`, a popular SQL benchmark tool, on EBS and several different configurations of Blizzard. We used `sqliosim` in a configuration that had several threads doing random IO to 4 databases in parallel. Each database was 4 GB in size, with its own log file. In the background, read-ahead and bulk updates occurred. Note that `sqliosim` uses write-through IO, not flushes, to provide consistency, so Blizzard with fast acknowledgment simply writes data to remote disks as quickly as possible and provides no crash consistency. We also ran a variant of fast ac-

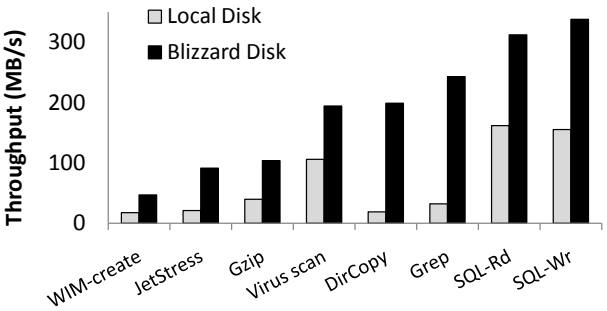


Figure 8: Macrobenchmarks: Blizzard's virtual drive (write-through, single replication) versus a local physical drive.

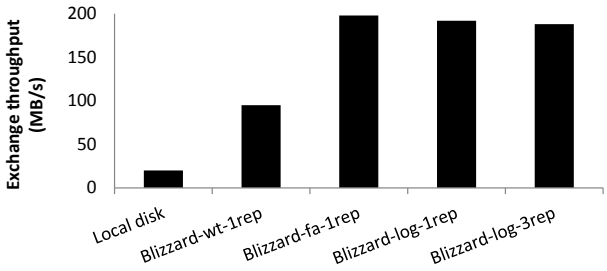


Figure 9: Exchange throughput. For the log-based commit results, Blizzard's block size was set to 64 KB, to match the size of Exchange's transactional IOs. Other Blizzard configurations used a block size of 128 KB.

knowledgment mode which inserted a fake flush every 20 writes. That variant, which immediately acknowledges writes but bounds data loss to 20 writes, performs slower than Blizzard with write-through or vanilla fast acknowledgments; this is because the fake flushes reduced the amount of write parallelism. Nonetheless, all three versions of Blizzard issued significantly more IOps than EBS.

4.3 Macrobenchmarks

Figure 8 shows Blizzard's performance for several IO-intensive Win32 workloads; Blizzard was configured in write-through mode and used single replication. The `WIM-create` workload represents the time needed to generate a bootable WinPE [29] .iso file using the Windows Automated Installation Kit (a WinPE image is a minimal Windows installation that is useful for creating recovery CDs and other diagnostic tools). `JetStress` [23] is a seek-intensive application with 16 threads that emulates the IO load of an Exchange email server. `Gzip` is a file compression program that uses four IO threads. `Virus scan` represents the throughput of a full system analysis performed by System Center 2012 Endpoint. `DirCopy` is derived from the built-in Windows `robocopy` tool, and it recursively copies directories using eight threads. `Grep` is an eight-way threaded program that evaluates regular expressions over file data. `SQL` refers to the `sqliosim` tool that database administrators use to characterize disk performance. By default, the tool launches eight threads that perform random database queries, eight threads that perform sequential queries, and eight threads that perform bulk database updates.

In Figure 8, throughput numbers refer to application-level performance, not disk-level performance. For example, `Gzip` throughput refers to how many MBs of file data the program can compress per second. Similarly, `SQL` throughput refers to how quickly the SQL engine can read or write the database. Figure 8 shows that Blizzard can improve unmodified application's performance by a factor of 2x–10x. Blizzard provides the greatest boost to programs like `DirCopy` and `Grep` which spend very little time on CPU operations.

4.4 Log-based Commit

To test the performance of Blizzard's out-of-order, log-based commit scheme (§2.5), we ran several tests involving the `JetStress` tool, which emulates the workload for an Exchange email server. Unlike `sqliosim`, which issues write-through operations to implement consistency, `JetStress` uses disk flushes. As shown in Figure 9, Blizzard in write-through mode with single replication provides a 4x throughput improvement over a local physical disk, and Blizzard in single replicated, fast acknowledgment mode provides a 9x improvement. Using single replication, Blizzard's log-based, out-of-order commit scheme was only 3% slower than single-replicated fast acknowledgment, despite occasionally needing to perform read-before-writes (§2.5.3). The triple-replicated log scheme was only 5% slower, since Blizzard could hide much of the latency associated with slow replicas (§4.1).

Log-based commits do not provide faster throughput or lower IO latency than simple fast acknowledgments because both schemes acknowledge writes and flushes immediately. However, the log-based scheme *issues* all writes immediately, whereas the simple scheme issues writes in epoch order, waiting for writes from epoch N to commit before issuing writes from epoch $N+1$. Thus, the simple scheme is more prone to data loss in the case of a client crash, since it buffers more writes in-memory than the log-based scheme (§4.7). The increased buffering requirement will also cause client-submitted IO requests to block more often, until Blizzard can deallocate memory belonging to newly retired epochs.

4.5 Multiple Active Clients

Blizzard clients stripe their data across a shared set of disks. As the number of active clients grows, the aggre-

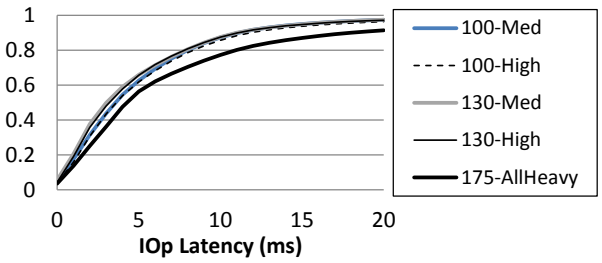


Figure 10: CDF of IOp latency (VDI workloads). The legend format is XXX-YYY, where XXX is the number of clients and YYY describes the client IOp rates. There were 130 remote disks.

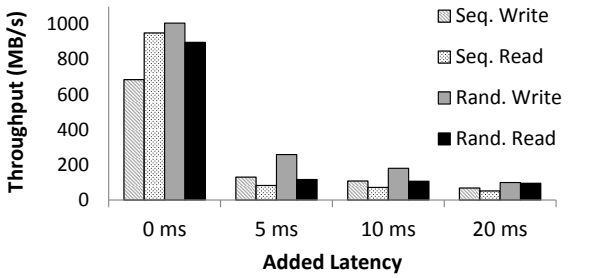


Figure 11: Comparing Blizzard's performance in our deployed network (far left) and our deployed network with synthetic latencies added.

gate request pressure on the disks increases. We designed nested striping (§2.3) and log-based permutation maps (§2.5) so that clients would spread their requests across multiple disks, preventing hotspots from emerging and leading to graceful degradation of disk IO queues. To test our design, we issued IO requests using a synthetic load generator that simulated a variable number of clients with varying levels of IOs per second (IOps). Each emulated client received 10 Gbps of network connectivity, and each client's virtual user was marked as "Light," "Normal," "Power," or "Heavy" based on its IOps rate (5, 10, 20, or 40 respectively). The size of each IOp ranged from 512 bytes to 1 MB, with the statistical distribution of IOp sizes and read/write ratios governed by empirical studies of VDI workloads [12, 13]. Note that a single high-level IOp resulted in multiple FDS operations if the IOp was larger than a Blizzard virtual block.

Figure 10 provides a CDF of IOp latencies for several different client deployments; to measure true IOp latencies, Blizzard was operated in write-through mode. The "Medium" deployment was 10% Light, 50% Normal, 25% Power, and 15% Heavy. The "High" client split was 10%/30%/40%/20%, and the pessimistic "AllHeavy" split was 0%/0%/0%/100%. In all cases, there were 130 remote disks. Figure 10 shows that, with 100 clients (i.e., more than one disk per client) and 130 clients (exactly one disk per client), Blizzard provides IOp latencies that are competitive with those of a local physical disk: at least 62% of IOps had 5 ms of latency or lower, and 85% of IOps had 10 ms of latency or lower.

Interestingly, latencies in the 130 client test were slightly *lower* than those in the 100 client test, e.g., in the "High" IOps test, 65.7% of IOps in the 130 node deployment had 5 ms or less of latency, but this was true for only 61.9% of IOps in the 100 node test. The reason is that nested striping distributes the aggregate client load evenly across all disks. Thus, each disk sees a random stream of seek offsets. When the number of clients increased from 100 to 130, disk queues got deeper, but this was *better* for disks that were serving random workloads—as each disk arm swept across its platters, there were more opportunities to service queued IO requests. Of course, longer disk queues will eventually increase IOp latency, as demonstrated by the pessimistic "AllHeavy" deployment which had 175 clients (i.e., 1.34 clients for each disk) and 40 IOps per client. Even in this case, 56% of IOs had latencies no worse than 5 ms, and 77% had latencies no worse than 10 ms.

4.6 Latency Sensitivity

Blizzard is designed for full-bisection networks in which clients have fast, low-latency connections to remote disks. For example, in our current deployment, clients and storage nodes communicate via links with 500 microseconds of latency. This is an order of magnitude smaller than the seek time for a disk, allowing Blizzard to make network-attached disks as fast to access as local ones. Figure 11 depicts Blizzard's performance with synthetic network latencies added, and with write-through semantics enabled (i.e., the virtual drive does not acknowledge writes to clients until those writes are durable on remote disks). With five milliseconds of additional latency, Blizzard's throughput drops by a factor of 5x–10x, and with twenty milliseconds of additional latency, performance is essentially equivalent to that of a single local disk.

These experiments highlight the importance of Blizzard's congestion-free, full-bisection bandwidth network. In such a system, network delays are negligible, and the client-perceived latency for a write-through IO is governed by the time needed to perform a single seek on a remote storage server (see Figure 10). Figure 11 shows that if the storage network lacks a fast interconnect, then millisecond-level network latencies effectively double or triple the seek penalty for accessing a remote disk. In these scenarios, Blizzard's fast acknowledgment schemes are crucial for eliminating remote access penalties from the critical path of writes.

Write workload	Crash-consistent recoveries		
	Write-through	FastACK	FastACK + log
Only new physical blocks targeted	50/50	50/50	50/50
50% new targets, 50% overwrites	50/50	50/50	50/50
JetStress	50/50	50/50	50/50

Figure 12: For all 450 injected crashes, Blizzard recovered to a prefix-consistent version of the virtual disk.

4.7 Reliability

In this section, we demonstrate two things. First, Blizzard always recovers a crashed virtual drive to a prefix-consistent state. Second, if a Blizzard client primarily issues block-aligned writes for entire blocks of data, the client should use log-based commit to reduce data loss and decrease memory pressure. If clients frequently issue writes that are misaligned, or not an even multiple of Blizzard's block size, clients should use the simple fast acknowledgment scheme (if they wish to maximize performance), or write-through mode (if they wish to minimize data loss).

Recovering to consistent virtual drives: To test Blizzard's reliability in the presence of crashes, we modified the user-mode portion of the virtual driver so that it randomly injected emulated failures. At each emulated failure point, the driver logged the set of writes that were buffered in memory, as well as the subset of those writes that had been issued to remote disks, but not yet acknowledged as being durable. Writes that are buffered but unissued at crash time are lost. Writes that are issued but unacknowledged at crash time may or may not be durable—their durability depends on whether the client OS had actually sent the writes over the network before the crash, and whether remote disks crashed while handling the writes, and so on.

We used three synthetic workloads to explore Blizzard's reliability. Our first workload issued 40 writes per second, such that each write targeted a previously unwritten portion of the virtual disk. Our second workload also issued 40 writes per second, but 50% of the writes targeted new locations, and 50% targeted previously written blocks. Note that a write-only workload of 40 IOps is intense for a POSIX application [12, 13]. We configured Blizzard to use a block size of 128 KB, with half of the writes being 64 KB in size, and the other half being 128 KB, ensuring that, when Blizzard was run in log-based commit mode, the read-after-write code paths would be stressed. Our final workload was JetStress [23], a seek-intensive workload that simulates an Exchange email server. The JetStress tests also used

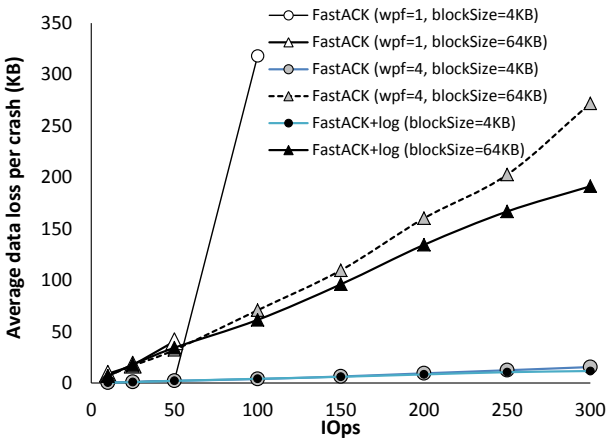


Figure 13: Blizzard loss rates when the size of each write is aligned with Blizzard's block size. "wpf" means "writes per flush."

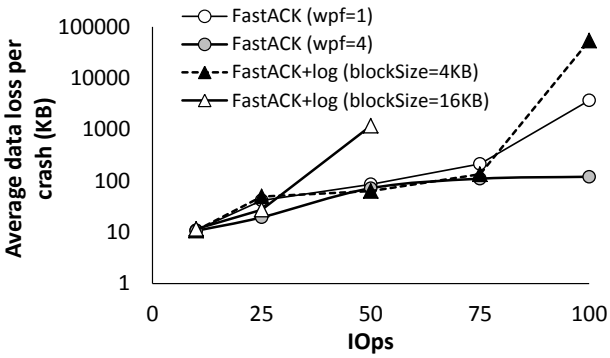


Figure 14: Blizzard loss rates for a simulated write-only VM workload (writes vary between 4KB and 1 MB in size). Note that the y-axis is log-scale.

a Blizzard block size of 128 KB, and in all experiments, the virtual drive used 128 backing disks.

Figure 12 shows the results of our experiments. For all 450 injected crashes, Blizzard recovered a prefix-consistent version of the virtual drive. To validate whether the recovered drive was prefix-consistent, we ran Blizzard's recovery code, and then used the write log to verify that the recovered disk contained a prefix-consistent representation of the write stream.

Bounding data loss: In fast acknowledgment mode and log-based commit mode, Blizzard exchanges performance for the risk of data loss. A recovered virtual drive is always consistent, but the drive may not contain a trailing set of writes that Blizzard acknowledged to the client but failed to make durable before the crash occurred. To measure this data loss, we ran two additional experiments.

In the first experiment, we generated a synthetic stream of write operations. Each write was the exact size of Blizzard's block size, and it was aligned with a block boundary, ensuring that, when Blizzard used log-based commits, there was no read-before-write penalty (§2.5.3). Using our instrumented Blizzard driver, we picked random moments to simulate crashes, and logged the amount of buffered, unacknowledged data that would be lost in the simulated crash. Our load generator also injected a configurable number of flush requests. Figure 13 shows the results. Each data point represents 100 simulated crashes.

As expected, the loss rate increases as the IO rate increases, because Blizzard must buffer more data. With one write per flush, i.e., with a total ordering over all writes, the simple fast acknowledgment scheme can only have one outstanding write to a remote disk. This severely limits the rate at which writes can retire, and it increases the memory pressure needed to buffer writes. With a write size of 4 KB, the fast acknowledgment scheme can only handle 100 IOps before too many writes queue up, and the virtual drive throttles the client to bound potential data loss; with a write size of 64 KB, the scheme can only handle 50 IOps before it throttles the client. The log-based commit scheme can issue writes immediately, regardless of the flush rate, so this scheme can gracefully scale up to 300 IOps.

For a flush rate of once-every-four writes, the fast acknowledgment scheme does much better, scaling all the way to 300 IOps. With a wider flush epoch, the fast acknowledgment scheme can issue more writes in parallel, and when a write from a new epoch arrives, old writes from the prior epoch are likely to already be committed, or at least issued; thus, the new write is unlikely to be delayed by a full seek time on a remote disk. With a write size of 4 KB, the difference in average loss rates between the two schemes is large in relative percentage, but small in absolute value—at 300 IOps, the fast acknowledgment scheme loses 4.1 writes representing 16.4 KB of data, and the log-based scheme loses 2.4 writes representing 11.5 KB of data. For a larger write size of 64 KB, the data loss per dropped write increases, and the two schemes show bigger differences in absolute amounts of data loss. For IOp rates above 100, the log-based scheme decreases loss rates by 12–30%. For example, at 300 IOps with 64 KB writes, the log-based scheme loses 191 KB per crash, whereas the simple fast acknowledgment scheme loses 272 KB.

Blizzard's log-based commit scheme pays a read-before-write penalty for writes that are smaller than Blizzard's block size. If writes are frequently misaligned (or not even multiples of the block size), the log-based scheme will force many writes to wait for synchronous reads. This will increase buffering requirements and the issue latencies for writes, causing loss rates after a crash to increase. Figure 14 shows this effect. In this example, the writes are aligned with Blizzard's block boundaries, but they range in size from 4 KB to 1 MB, as determined by empirical distributions of VM write sizes [12, 13]. In these experiments, the log-based scheme with a block size of 16 KB could only handle up to 50 IOps—beyond that, the read-before-write penalty forced Blizzard to throttle the client's write rate. Decreasing the block size to 4 KB resulted in fewer read-before-writes, allowing the log-based scheme to scale better. At 75 IOps, the log-based scheme beat the fast acknowledgment scheme by 37%, with a data loss of 135 KB instead of 214 KB. However, at 100 IOps, the log-based scheme can no longer hide the read-before-write penalties, and it has an average data loss of 54 MB, an order of magnitude worse than fast acknowledgments with a wpf of 1, and two orders of magnitude worse than fast acknowledgments with a wpf of 4.

5 Related Work

Block-level interfaces: A variety of protocols use a block interface to expose a single disk to remote clients. Examples of such protocols include ATA-over-Ethernet [22] and iSCSI [36]. Blizzard extends the simple block interface, mapping each virtual drive to multiple backing disks, and providing high-level software abstractions like replication and failure recovery across thousands of disks.

Like Blizzard, Petal [24] defines a distributed, software-implemented virtual disk that is backed by remote storage. However, Blizzard can coexist with traditional big-data workloads, and Blizzard leverages a full-bisection bandwidth network to stripe data more aggressively than Petal; the latter exposes Blizzard clients to higher levels of disk parallelism. Blizzard also leverages delayed durability semantics to increase the rate at which clients can issue writes while still achieving crash consistency.

Salus [43] is another example of a virtual block store. Salus is built atop HDFS/HBase [5, 6], and it provides ordered-commit semantics during normal operation, and prefix semantics when failures occur. Salus achieves these properties with pipelined commit, a protocol that resembles two-phase commit. In contrast, Blizzard achieves consistency with only a single round of communication between the client and the remote data stores. This reduces both network traffic and software complexity.

Mapping schemes: Using techniques like nested striping (§2.3) and deterministic permutations (§2.5), Blizzard translates virtual block accesses to FDS-level block accesses. This is similar to how SSDs use a

Flash Translation Layer (FTL) to map virtual blocks to physical ones. SSDs employ a variety of optimizations to minimize the size of the mapping table that is kept in the SSD's small, on-board SRAM (e.g., [18, 45]). While these optimizations could be applied to Blizzard's mapping structures, we have not found a need for such approaches, since Blizzard's tables are small ($\sim$20 MB) and they easily fit within main memory. FTLs must also implement compaction and garbage collection, since SSD writes and SSD erases have different data sizes. Blizzard's log-based commit scheme avoids compaction and garbage collection by using equivalent sizes for writes and erases in the log.

Cloud-scale storage systems: BlueSky [42] uses NFS or CIFS proxies to expose commercial cloud storage like Azure to enterprise clients. BlueSky allows an enterprise to offload the administrative costs of the storage cluster to a third party. However, compared to systems in which clients and servers reside in the same cloud, BlueSky introduces several new sources of overhead. Pulling data from commercial cloud servers injects wide-area latencies into the IO path. BlueSky proxies also use the chatty, text-based HTTP protocol to communicate with remote servers. The HTTP overhead is magnified if the enterprise has asymmetric upload/download speeds to the cloud—slower upload speeds mean that even small HTTP headers can add significant transfer delays [16].

Unlike BlueSky's focus on WAN access to cloud storage, Parallel NFS (pNFS) exposes clients to local (i.e., on-premise) cloud storage [38]. Storage servers can export a block interface, an object interface, or a file interface; pNFS clients transparently convert NFS requests to the appropriate lower-level access format. pNFS requires applications to adhere to NFS semantics, and both pNFS and BlueSky lack key Blizzard features like role-based striping and asynchronous epoch-based commits for writes.

Desktop/server file systems: OptFS demonstrated how to decouple durability from ordering in the context of a journaling file system [7]. Blizzard shows that these ideas can be applied at the disk level, providing OptFS-style performance improvements in a *file system-agnostic way* that does not depend on knowledge of the file system's consistency scheme (e.g., journaling [40] or shadow paging [21, 35]). OptFS requires disks to be modified to provide asynchronous durability notifications; in contrast, Blizzard's virtual disk implements prefix consistency using standard, asynchronous write-through operations on the backing remote disks.

When Blizzard uses log-based commit, it leverages expanded blocks to enable crash recovery and disconnected operation. Transactional Flash [34] and backpointer-based consistency [8] also embed extra metadata in out-of-band areas.

BPFS [10] is a file system for use with byte-addressable, persistent memory hardware (e.g., Phase Change Memory). BPFS introduces an abstraction, called an epoch barrier, that allows ordering guarantees to be expressed without requiring an immediate flush of dirty data in the CPU cache. Epoch barriers provide data consistency while preserving the ability of the memory controller to reorder writes within an epoch. Epoch barriers require custom hardware, and BPFS expects that the persistent memory resides directly on the memory bus. Like BPFS, Blizzard also separates ordering from durability; however, the separation is implemented in the context of a distributed system, rather than a single machine with access to persistent memory.

The Zebra file system [19] combines ideas from RAID [32] and log-based file systems [35], striping a per-client file log across a RAID array. Zebra does not provide mechanisms for asynchronous flush handling, and this constrains the level of disk parallelism that Zebra can provide to applications. Zebra uses compaction and garbage collection to manage dead block data; when such log cleaning occurs, it can introduce unpredictable performance fluctuations [37]. In contrast, when Blizzard operates in log-based asynchronous commit mode, it uses reads-before-writes to only commit full blocks of data. This smooths out the background IO traffic that is required for log maintenance. However, Section 4.7 demonstrates that if clients frequently issue misaligned writes, Blizzard's read-after-write penalty can be large, making Blizzard's simple fast acknowledgment scheme more attractive.

6 Conclusions

Blizzard exposes unmodified, cloud-oblivious POSIX applications to a fast, cloud-scale block store. This block store, which clients mount as a virtual disk, efficiently supports small, random IOs, but it coexists alongside big-data file systems, and deploys atop the same servers, disks, and switches. Using a network with full-bisection bandwidth, Blizzard provides clients with fast access to any remote disk. Using a novel striping scheme, Blizzard maximizes disk parallelism, avoids disk hotspots, and reduces IOp convoy dilation. By carefully ordering writes, Blizzard can immediately acknowledge flush requests while still providing crash consistency; with fewer write barriers, clients can issue writes faster, and better leverage the spindle parallelism of remote storage. A Blizzard prototype improves the speed of unmodified POSIX applications by up to an order of magnitude. In summary, Blizzard makes it much easier for cloud-agnostic POSIX applications to receive cloud-scale performance and availability.

References

[1] Amazon. Amazon EBS-Optimized Instances. AWS Documentation. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSOptimized.html. October 15, 2013.

[2] Amazon. Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs/. 2014.

[3] Amazon. Amazon Relational Database Service (Amazon RDS). http://aws.amazon.com/rds/. 2014.

[4] Amazon. Amazon Simple Email Service (Amazon SES). http://aws.amazon.com/ses/. 2013.

[5] Apache. Apache HBase. http://hbase.apache.org. 2014.

[6] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf. 2007.

[7] V. Chidambaram, T. Pillai, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of SOSP*, pages 228–243, Farmington, PA, November 2013.

[8] V. Chidambaram, T. Sharma, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of FAST*, pages 101–116, San Jose, CA, February 2012.

[9] J. Cipar, G. Ganger, K. Keeton, C. Morrey III, C. Soules, and A. Veitch. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proceedings of EuroSys*, pages 169–182, Bern, Switzerland, April 2012.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of SOSP*, pages 133–146, Big Sky, MT, 2009.

[11] A. Edwards and B. Calder. Exploring Windows Azure Drives, Disks, and Images. Microsoft. http://blogs.msdn.com/b/windowsazurestorage/archive/2012/06/28/exploring-windows-azure-drives-disks-and-images.aspx. June 27, 2012.

[12] D. Feller. Virtual Desktop Resource Allocation. The Citrix Blog. http://blogs.citrix.com/2010/11/12/virtual-desktop-resource-allocation. November 12, 2010.

[13] R. Fellows. Storage Optimization for VDI. Tutorial: Storage Networking Industry Association. http://www.snia.org/sites/default/education/tutorials/2011/fall/StorageStorageMgmt/RussFellowsSNW_Fall_2011_VDI_best_practices_final.pdf. 2011.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, December 2003.

[15] Google. Google Cloud SQL. https://cloud.google.com/products/cloud-sql. 2014.

[16] Google. Performance Best Practices: Minimize request overhead. https://developers.google.com/speed/docs/best-practices/request. March 28, 2012.

[17] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of SIGCOMM*, pages 51–62, Barcelona, Spain, 2009.

[18] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *Proceedings of ASPLOS*, pages 229–240, Washington, DC, March 2009.

[19] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. *ACM Transactions on Computer Systems*, 13(3):274–310, 1995.

[20] D. Hildebrand, A. Nisar, and R. Haskin. pNFS, POSIX, and MPI-IO: A Tale of Three Semantics. In *Proceedings of the Workshop on Petascale Data Storage*, pages 32–36, Portland, OR, November 2009.

[21] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, San Francisco, CA, January 1994.

[22] S. Hopkins and B. Coile. AoE (ATA over Ethernet). http://support.coraid.com/documents/AoEr11.txt. February 2009.

[23] N. Johnson. JetStress 2010: JetStress Field Guide. Microsoft. http://gallery.technet.microsoft.com/Jetstress-Field-Guide-1602d64c. March 27, 2012.

[24] E. Lee and C. Thekkath. Petal: Distributed Virtual Disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, December 1996.

[25] A. Leung, S. Pasupathy, G. Goodson, and E. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of USENIX ATC*, pages 213–226, Boston, MA, June 2008.

[26] Microsoft. Introducing Windows Azure SQL Database. `http://msdn.microsoft.com/en-us/library/windowsazure/ee336230.aspx`. 2014.

[27] Microsoft. Windows Azure Active Directory. `http://www.windowsazure.com/en-us/services/active-directory/`. 2014.

[28] Microsoft. Windows Azure Storage Scalability and Performance Targets. Windows Azure Documentation. `http://msdn.microsoft.com/en-us/library/windowsazure/dn249410.aspx`. June 20, 2013.

[29] Microsoft. Windows PE Technical Reference. `http://technet.microsoft.com/en-us/library/dd744322(WS.10).aspx`. October 22, 2009.

[30] E. Nightingale, J. Elson, O. Hofmann, Y. Suzue, J. Fan, and J. Howell. Flat Datacenter Storage. In *Proceedings of OSDI*, pages 1–15, Hollywood, CA, October 2012.

[31] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the Sync. In *Proceedings of OSDI*, pages 1–14, November 2006.

[32] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *ACM SIGMOD Record*, 17(3):109–116, 1988.

[33] V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of USENIX ATC*, pages 105–120, Anaheim, CA, April 2005.

[34] V. Prabhakaran, T. Rodeheffer, and L. Zhou. Transactional Flash. In *Proceedings of OSDI*, pages 147–160, San Diego, CA, December 2008.

[35] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[36] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI). Technical report, RFC 3720, April, 2004.

[37] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proceedings of the USENIX Winter Technical Conference*, pages 249–264, New Orleans, LA, January 1995.

[38] S. Shepler, M. Eisler, and D. Noveck. Network File System (NFS) Version 4 Minor Version 1 Protocol. Technical report, RFC 5661, January, 2010.

[39] M. Steigerwald. Imposing Order. Linux Magazine, May 2007.

[40] S. Tweedie. Journaling the Linux ext2fs File System. In *Proceedings of the Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[41] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of SOSP*, pages 93–109, Kiawah Island Resort, SC, December 1999.

[42] M. Vrable, S. Savage, and G. Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *Proceedings of FAST*, pages 237–250, San Jose, CA, 2012.

[43] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin. Robustness in the Salus Scalable Block Store. In *Proceedings of NSDI*, pages 357–370, Lombard, IL, April 2013.

[44] E. Weisstein. Linear Congruence Method. MathWorld: A Wolfram Web Resource. `http://mathworld.wolfram.com/LinearCongruenceMethod.html`. 2014.

[45] Y. Zhang, L. Arulraj, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of FAST*, pages 1–16, San Jose, CA, February 2012.

Aggregation and Degradation in JetStream: Streaming analytics in the wide area

Ariel Rabkin, Matvey Arye, Siddhartha Sen*, Vivek S. Pai, and Michael J. Freedman
Princeton University

Abstract

We present JetStream, a system that allows real-time analysis of large, widely-distributed changing data sets. Traditional approaches to distributed analytics require users to specify in advance which data is to be backhauled to a central location for analysis. This is a poor match for domains where available bandwidth is scarce and it is infeasible to collect all potentially useful data.

JetStream addresses bandwidth limits in two ways, both of which are explicit in the programming model. The system incorporates structured storage in the form of OLAP data cubes, so data can be stored for analysis near where it is generated. Using cubes, queries can aggregate data in ways and locations of their choosing. The system also includes adaptive filtering and other transformations that adjusts data quality to match available bandwidth. Many bandwidth-saving transformations are possible; we discuss which are appropriate for which data and how they can best be combined.

We implemented a range of analytic queries on web request logs and image data. Queries could be expressed in a few lines of code. Using structured storage on source nodes conserved network bandwidth by allowing data to be collected only when needed to fulfill queries. Our adaptive control mechanisms are responsive enough to keep end-to-end latency within a few seconds, even when available bandwidth drops by a factor of two, and are flexible enough to express practical policies.

1 Introduction

This paper addresses the problem of analyzing data that is continuously created across wide-area networks. Queries on such data often have real-time requirements that need the latency between data generation and query response to be bounded. Existing stream processing systems, such as Borealis, System-S, Storm, or Spark Streaming [2, 6, 29, 32], address latency in the context of a single datacenter where data streams are processed inside a high-bandwidth network. These systems are not designed to perform well in the wide area, where limited bandwidth availability makes it impractical to backhaul all potentially useful data to a central location. Instead, a system for wide-area analytics must prioritize which data to transfer in the face of time-varying bandwidth constraints.

We have designed and built such a system, called JetStream, that extends today's dataflow streaming programming model in two ways. We incorporate structured storage that facilitates **aggregation**, combining related data together into succinct summaries. We also incorporate **degradation** that allows trading data size against fidelity. Just as MapReduce helps developers by handling fault tolerance and worker placement, JetStream aims to drastically reduce the burden of sensing and responding to bandwidth constraints.

Wide-area data analysis is a problem in a variety of contexts. Logs from content distribution networks and other computing infrastructure are created on nodes spread out across the globe. Smart electric grids, highways, and other infrastructure also generate large data volumes. Much of this data is generated "near the edge," with limited network connectivity over cellular or wireless links. Unlike traditional sensor network deployments, many of these infrastructure sensors are not energy limited, and can have substantial co-located computation and storage.

Wide-area analysis also applies to data that does not resemble traditional logs. Networks of video cameras are used for a wide variety of applications. These include not only urban surveillance but also highway traffic monitoring and wildlife observation. The cost of electronics, including sensors, storage, and processors, is currently falling faster than the cost of wireless bandwidth or of installing new wired connectivity. As a result, bandwidth is already becoming the limiting constraint in such systems [9] and we expect the gap between sensing capacity and bandwidth to increase in the coming years.

In the examples above, a large amount of data is stored at edge locations that have adequate compute and storage capacity, but there is limited or unpredictable bandwidth available to access the data. Today's analytics pipelines lack visibility into network conditions, and do not adapt dynamically if available bandwidth changes. As a result, the developer must specify in advance which data to store or collect based on pessimistic assumptions about available bandwidth. The consequence is that bandwidth is typically over-provisioned compared to average usage, and so capacity is not used efficiently.

*Current affiliation: Microsoft Research

JetStream's goal is to enable real-time analysis in this scenario, by reducing the volume of data being transferred. Storing and aggregating data where it is generated helps, but does not always reduce data volumes sufficiently. Thus, JetStream also includes degradations. These are data transformations analogous to "lossy" compression: they reduce data size at the expense of accuracy. Examples include computing per-minute aggregates for queries requesting per-second data, or dropping some fraction of the data via sampling. Since degradations impose a (tunable) accuracy penalty, JetStream is designed to monitor available bandwidth and use the minimal degree of degradation required to keep latency bounded.

Integrating aggregation and degradation into a streaming system required us to address three main challenges:

(1) Incorporating storage into the system while supporting real-time aggregation. Aggregation for queries with real-time requirements is particularly challenging in an environment where data sources have varying bandwidth capacities and may become disconnected. JetStream integrates structured storage as a first-class abstraction in its programming model, allowing aggregation to be specified in a flexible but unified way.

(2) Dynamically adjusting data volumes to the available bandwidth using degradation mechanisms. Such adaptation must be performed on a timescale of seconds to keep latency low.

(3) Allowing users to formulate policies for collecting data that maximize data value and that can be implemented effectively by the system. The policy framework must be expressive enough to meet the data quality needs of diverse queries. In particular, it should support combining multiple degradation mechanisms.

In meeting these challenges, we created the first wide-area analysis system that adjusts data quality to bound the latency of streaming queries in bandwidth-constrained environments. Our architecture decouples bandwidth sensing from the policy specifying how to aggregate and degrade the data to guarantee a timely response. The interfaces defined by our architecture support a wide-range of sensing methods and response techniques—for example, we implemented a diverse set of degradations, including those using complex data structures and multi-round protocols. We consider our architecture and its associated interfaces to be the key contribution of this paper.

By ignoring bandwidth limitations, previous systems force users to make an unappealing choice: they can be optimistic about available bandwidth and backhaul too much data, leading to buyer's remorse if bandwidth is costly; or they can be pessimistic about bandwidth and backhaul only limited data, leading to analyst's remorse if a desired query cannot be answered. By integrating durable storage into the dataflow and supporting dynamic adjustments to data quality, JetStream allows a user to focus on fundamentally different trade-offs: deciding which query results are needed in real-time and which inaccuracies are acceptable to maintain real-time performance.

2 Design Overview

JetStream is designed for near-real-time analysis of changing data, such as log data or audiovisual data. The system integrates data storage at the edge to allow users to collect data that may be useful for future analysis without necessarily transferring it to a central location. Users can define ad-hoc queries ("give me the video from camera #129 between 6am and 7am last night") as well as standing queries ("send a down-sampled copy of the video data from every camera back to a control center" or "tell me the top-10 domains by number of requests over 10 seconds"). Standing queries can be useful in and of themselves or they can be used to create centralized data structures to optimize the performance of common queries.

Standing queries are considerably more challenging than ad-hoc queries, and therefore are the focus of this paper. A standing query has a hard real-time requirement: if the query cannot handle the incoming data rate, then queues and queuing delays will grow, resulting in stale results. Giving users fresh results means that the system must keep latency bounded. This bound must be maintained even as the incoming data volume and the available bandwidth fluctuate.

Since JetStream aims to provide low-latency results on standing queries, it borrows the basic computation model of many of today's stream-processing systems. A worker process runs on each participating compute node. A query is implemented by a set of linked dataflow operators that operate on streams of tuples. Each operator is placed on a particular host and performs some transformations on the data. The system routes tuples between operators, whether on the same host or connected by the network.

2.1 Integrating structured storage

In a departure from previous stream processing systems, we integrate structured storage inside the operator graph. This storage lets us keep data where it is generated until it is needed. In our vision, nodes at the edge of the network can store hours or days worth of data that the user does not need to immediately analyze, but which may (or may not) be required later.

Because edge storage can involve large data volumes, we use structured storage to reduce query times. Past streaming systems incorporated storage in the form of a durable buffer of input tuples [2]. This would perform poorly for ad-hoc queries, since it would require re-scanning all stored data on every query. We instead adopt the data cube abstraction previously used in OLAP databases [15], which supports queries efficiently. We discuss our use of cubes in detail in Section 3.

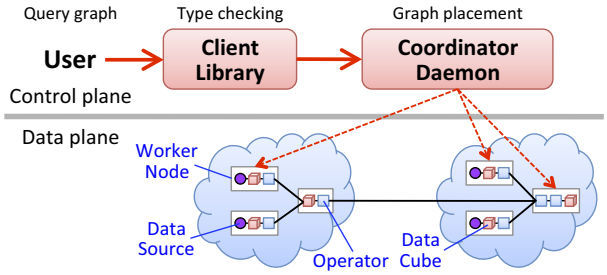


Figure 1: JetStream's high-level architecture. Users define query graphs with operators and cubes. A coordinator deploys the graph to worker nodes.

Integrating structured storage allows us to simplify other operators, enabling more graceful handling of missing or delayed data. Previous streaming systems defined a large set of stateful operators, including Sort, Aggregate, Join, and Resample [2]. When made stateful, each of these operators requires complex semantics to cope with missing inputs. In comparison, in our design, cubes are the only element with durable state and the only place where streams are merged, and so subsume the functionality of these operators. They are responsible not only for storage, but also for aggregation inside queries.

We opt for stream equi-joins over general joins, since general joins across streams would impose a global synchronization barrier, causing excessive latency when links are congested or disabled. We have not found the lack of a general join to be a serious limitation. Cubes can handle stream equi-joins from an arbitrary number of streams and this has been enough for us in practice.

2.2 Reducing data volumes

Operators can apply lossy transformations that reduce data volumes. For example, they can randomly sample tuples from a stream, or drop all tuples with values below a threshold. To maximize data quality, it is desirable to apply data reduction only when necessary to conserve bandwidth. JetStream therefore includes specialized *tunable degradation* operators. The system automatically adjusts these operators to match available bandwidth, thus minimizing the impact on data quality. These mechanisms are described in Section 4.

Because the system dynamically changes degradation levels, it needs a way to indicate the fidelity of the data it is sending downstream. Therefore, we added the ability for operators to send metadata messages both upstream and downstream. Metadata messages are ordered with respect to data messages and act as punctuations [30]. This mechanism is broadly useful in the system: we use it to signal congestion and also to implement multi-round protocols within the operator graph (see Section 5).

Class	Operators
I/O	FileReader, Echo, FileWriter, UnixCmd
Parser	CSVParser, RegexParser
Filter	Grep, LessThan, Equals
Map	ExtendTupleWithConst, Project, AddTimestamp, URL2Domain, Histogram2Quantile
Degradation	VariableSubscriber, DegradeHistogram, Sampling

Table 1: Examples of the operators currently provided by JetStream's client library.

2.3 Programming model

Operator graphs are constructed using a client library. The library's programming interface makes it convenient to configure operators and data cubes and to link them together into dataflow graphs. The system includes a library of pre-defined operators, and users can also define their own operators using a library of base classes. These base classes streamline development for common operator functionality such as map and filter, which both require implementing a single virtual function. We list some examples in Table 1 to give the flavor of the tasks operators perform. We discuss the details of the system architecture and implementation in Section 6.

Figure 2 gives a simple example of our programming model. Suppose there is a set of N nodes, each with a directory full of images. Over time, a camera adds data to this directory. The system will scan the directory and copy new images across the network to a destination, tagging each with the time and hostname at which it was created. In this example, placement is explicit: the semantics of the application requires that the readers are on the source nodes and that the result is produced at the union node. If there had been intermediate processing, the programmer could have let the system handle the placement of this computation. Later in the paper, we will extend this example to cope with insufficient bandwidth; we offer it here to give a sense of the programming model.

JetStream is primarily an execution engine, more like the Dryad or MapReduce engines than like the DryadLINQ or Pig languages [10, 20, 24, 31]. If in the future a widely-accepted declarative programming language for stream processing emerges, we expect that JetStream should be able to support it.

3 Aggregation

Integrating structured storage into a streaming operator graph solves several problems. Cubes, unlike key-value or general relational models, have unambiguous semantics for inserting and aggregating data. This lets JetStream handle several different forms of aggregation in a unified way. Windowed aggregation combines data residing on the same node across time. For example, all web request

```
g = QueryGraph()
dest = Operators.StoreImages(g, IMAGE_OUT_DIR)
dest.instantiate_on(union_node)

for node in source_nodes:
  reader = Operators.FileReader(g, options.dirname)
  reader.instantiate_on(node)
  add_time = Operators.Timestamp()
  add_host = Operators.Extend("$HOSTNAME")
  g.chain([reader, add_time, add_host, dest])
```

Figure 2: Example code for a query. Each source node has an image reader connected by a chain of operators to a common destination operator.

records for the same second can be merged together to produce a per-minute request count. Tree aggregation combines data residing on different nodes for the same period. This can be applied recursively: data can be grouped within a datacenter or point of presence, and then data from multiple locations is combined at a central ("union") point. Both these forms of aggregation are handled in JetStream by inserting data into structured storage and then extracting it by appropriate queries.

While JetStream borrows the cube interface from OLAP data warehouses, we substitute a different implementation. Cubes in data warehouses typically involve heavy precomputation. Many roll-ups or indexes are constructed, incurring high ingest times to support fast data drill-downs. In contrast, JetStream uses cubes for edge storage and aggregation and only maintains a primary-key index. This reduces ingest overhead and allows cubes to be effectively used inside a streaming computation.

Integrating storage with distributed streaming requires new interfaces and abstractions. Cubes can have multiple sources inserting data and multiple unrelated queries reading results. In a single-node or datacenter system it is clear when all the data has arrived since synchronization is cheap. In a wide-area system, however, synchronization is costly, particularly if nodes are temporarily unavailable. In this setting, queries need a flexible way to decide when a cube has received enough data to send updates downstream. Our context also requires cubes to deal with late tuples that arrive after results have been emitted. Before explaining how we solve these problems, we describe our storage abstraction in more detail.

3.1 Data Cubes and Their API

A data cube is a multi-dimensional array that can encapsulate numerical properties and relationships between fields in structured input data, similar to a database relation. It is defined by a set of dimensions, which specify the coordinates (the key) of an array cell, and a set of aggregates, which specify the statistics (values) stored in a cell.

Suppose for example that we are collecting statistics about traffic to a website. We might define a cube with dimensions for URLs and time periods. The cube would

```
g = QueryGraph()
dest = Cube(g, "stored_images")
dest.add_dimension(TIME, "timestamp")
dest.add_dimension(HOSTNAME, "timestamp")
dest.add_aggregate(BLOB, "img_data")
dest.instantiate_on(union_node)

for node in source_nodes:
  reader = Operators.FileReader(g, options.dirname)
  reader.instantiate_on(node)
  add_time = Operators.Timestamp()
  add_host = Operators.Extend("$HOSTNAME")
  g.chain([reader, add_time, add_host, dest])
```

Figure 3: Running example but with the destination now a cube.

then map each unique pair of URL and time period to a cell with a set of aggregates, such as the total number of requests and the maximum request latency. Each web request, when added to the cube, updates the cell corresponding to its URL and time period.

A query can ***slice*** a cube to yield a subset of its values, such as "all URLs starting with foo.com ordered by total requests." A query can ***roll up*** a slice, aggregating together the values along some dimension of the cube, such as asking for the total request count, summed across all URLs in the slice. Roll-ups use the same aggregation function as insertion. Whereas insertion potentially aggregates data at write time, a roll-up performs aggregation at query time. Aggregate functions must be deterministic and order-independent (i.e., `max` and `average`, but not `last-k tuples`); this means that the system need not worry about the ordering between inserts.

Aggregates can be more complex than simple integers. A cube cell can include a histogram or sketch, describing a whole statistical distribution. (This relies on the fact that the underlying sketches or histograms can be combined straightforwardly, without loss of accuracy.) One might, for instance, build a cube not merely of request counts over time, but directly representing the distribution of latencies over time. This allows a query to do powerful statistical processing, such as finding quantiles over arbitrary subsets of events. Histograms and sketches are fixed-size, regardless of the underlying data size; they are an especially compact form of aggregation.

The cube abstraction is powerful enough to directly express all the aggregation we need in JetStream. Windowed operations (such as moving averages) are repeated roll-ups over a time-varying slice of the cube. Both sliding windows and tumbling windows fit into this model.

Even when data is not aggregated together, a data cube is a useful storage abstraction that allows queries on multiple attributes. For example, Figure 3 shows how a cube can be integrated into our running example. Each image frame is stored as an aggregate, with the timestamp and source hostname as dimensions. This allows queries based on any combination of time and source. (Modern

video encoding does not store each frame separately. A more complex example might store short segments of video data in cube cells. Alternatively, a more complex implementation of the cube API might offer programmers the abstraction of a sequence of frames, while using a differential coding scheme underneath.)

Many implementations of the cube abstraction are possible; ours uses MySQL as the underlying data store. This allows us to leverage MySQL's optimizations for handling large data volumes as well as its support for transactions. Transactions greatly simplify failure recovery, as we explain in Section 6.

3.2 Integrating Cubes with Streaming

While the semantics of inserting data into the cube is straightforward ("apply the aggregation function"), extracting data from cubes is not. A query needs to make a policy decision of when and how often to read data from a cube. This decision affects the latency, bandwidth consumption, and completeness of results. In JetStream, these policies are encapsulated in specialized operators called subscribers.

Aggregation trades latency for bandwidth. The longer a query waits before reading the aggregate, the more data potentially can be aggregated into a fixed-size summary, reducing data volumes at the price of latency. There is also a trade-off between latency and completeness. It may not be worth waiting for stragglers before emitting a result. (As discussed below, the system can sometimes correct an inaccurate result later.) In the local area, stragglers can be masked by speculative execution or retries [10]. In the wide area, this strategy is unable to compensate for late data caused by limited bandwidth or connectivity.

We allow users to tune these trade-offs by giving subscriber operators fine-grained control over when to emit the results of aggregation. Subscribers have a richer API than other operators. They are notified whenever a tuple is inserted into the cube. They can also query the cube for slices and rollups. This allows fairly complex policies. A subscriber might repeatedly query for the last 10 seconds of data, relative to the system clock. Or it might track the highest timestamp from each source feeding into the cube, and only query up to the point which all the sources have reached. The default policy in JetStream combines these two policies: if all sources have contributed data, the result is emitted immediately. Otherwise, the subscriber has a fixed timeout before emitting results. Like all operators, the parameters of a subscriber can be adjusted by the user.

If an update arrives at a cube that would modify a result that has already been emitted by a subscriber, that update is termed *backfill*. Because cubes are the location where data is joined, the general problem of late updates only appears in JetStream as backfill at cubes.

We handle backfill using similar techniques to prior stream-processing work [2]. A backfill update results in a subscriber emitting a *delta record* that contains the old and new values. Delta records propagate in the same manner as new data. They can cause tuples to be revoked, e.g., if an operator filters an update that was previously allowed. The effect of a delta update on an aggregate depends on the aggregation function. Some aggregates, such as `average`, can retract an item that was previously added. For other items retraction can be an expensive operation. For example, `max` requires keeping a full list of its inputs to enable updates that reduce the value of the item with the largest value. Like Naiad [23], we only allow such functions if backfill input is impossible or the source data is available locally.

Subscribers are free to query the cube multiple times and are part of the metadata flow. They can therefore take part in nontrivial iterative protocols before emitting data. We exploited the flexibility of this interface when implementing a specialized subscriber that carries out a multi-round filtering protocol for finding the global top-k elements (by a user-determined ranking) over distributed data, without transferring all data. This is discussed further in Section 5.

3.3 Aggregation is Sometimes Insufficient

Not all queries aggregate well. For example, our running example of streams of image data is a case where aggregation is of limited value. There is no straightforward way to combine images taken at different times or from different cameras pointing at different scenes.

For data that can be aggregated in principle, the underlying data distribution may make aggregation ineffective at saving bandwidth. Data where the distribution of aggregate groups has a long tail will not aggregate well. This can depend on the coarseness of aggregation. For example, aggregating web requests by URL is ineffective because the popularity of URLs is long-tailed [16]. Aggregating the same data by domain can be much more effective, since domain popularity is less long-tailed. Figure 4 illustrates the point using data from the Coral content distribution network [13].

4 Adaptive Degradation

Even with partial aggregation at sources, some queries will require more bandwidth than is available. If there is insufficient bandwidth, the query will fall ever farther behind, as new data arrives faster than it can be processed. To keep latency low, JetStream allows queries to specify a graceful degradation plan that trades a little data quality for reduced bandwidth. For example, audiovisual data can be degraded by downsampling or reducing the frame rate. Similarly, quantitative data can be degraded by increasing the coarseness of the aggregation or using sampling. We

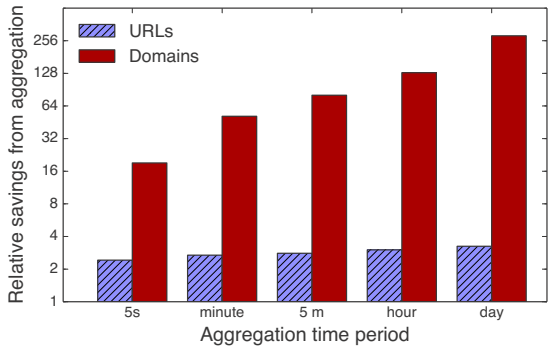


Figure 4: For CoralCDN logs, domains aggregate effectively over time and URLs do not.

discuss techniques applicable to quantitative data in more detail in Section 5.

Since degradations impose an accuracy penalty, they should only be used to the extent necessary. JetStream achieves this by using explicit feedback control [21]. Since wide-area networks can have substantial buffering beyond our visibility or control, waiting for backpressure to fill up queues incurs large delays and provides incomplete congestion information. By using explicit feedback, JetStream can detect congestion before queues fill up, enabling it to respond in a timely manner. Moreover, this feedback reflects the degree of congestion, allowing JetStream to tune the degradation to the right level.

JetStream's congestion response is decentralized: nodes react independently to their inferred bandwidth limits. This avoids expensive synchronization over the wide area, but it also means that sources may send data at different degradation levels, based on their bandwidth conditions at the time. Queries that aggregate data over sources or time must therefore handle varying degradation levels. We discuss how this is done for quantitative data in §5.2.

JetStream achieves adaptive congestion control via three components: (i) degradation operators that apply data transformations to the data stream; (ii) congestion monitors that measure the available bandwidth; and (iii) policies that specify how the system should adjust the level of degradation to the available bandwidth. Figure 5 illustrates the interaction between these three components; we discuss each component in the following sections.

4.1 Degrading data with operators

Degradation operators can be either standard operators that operate on a tuple-by-tuple basis, or cube subscribers that produce tuples by querying cubes. A degradation operator is associated with a set of degradation levels, which defines its behavior on cubes or data streams. For example, our variable subscriber offers the ability to roll-up data across different time intervals (e.g., sending output every 1, 5 or 10 seconds) and characterizes this degradation in terms of the estimated bandwidth use relative to the operator's maximum fidelity (in this example, $[1.0, 0.2, 0.1]$). This interface gives flexibility to operators. Some operators have fine-grained response levels, while others are widely spaced—for example, an audiovisual codec might only support a fixed set of widely spaced bitrates.

As we discuss in §5.1, many useful degradations have a data-dependent bandwidth savings. The interface we adopted gives operators flexibility in how they estimate the bandwidth savings. Levels (and their step size) can be (i) dynamically changed by the operator, e.g., based on profiling, (ii) statically defined by the implementation, or (iii) configurable at runtime (as with our currently implemented operators).

4.2 Monitoring available bandwidth

JetStream uses congestion monitors to estimate the relative available capacity of the system, or the ratio between the maximum possible data rate and the current rate. A ratio greater than one indicates spare capacity, while less than one indicates congestion. A congestion monitor is attached to each queue in the system and allows the associated policy to determine whether the data rate can be increased, or must be decreased, and by how much. Congestion monitoring can be done in many ways; we use two techniques in our prototype.

For detecting network congestion, we track the time required to process data. Sources insert periodic metadata markers in their output, specifying that the data since the last marker was generated over k seconds. When a network receiver processes this marker (which occurs after all prior data tuples are processed), it sends an upstream acknowledgment. The congestion monitor records the time t between seeing the last marker and receiving this acknowledgment, and uses $\frac{k}{t}$ as an estimate of the available capacity. (A similar approach is used to adapt bitrates in HTTP streaming [1].) The advantage of this approach is that it gives a meaningful estimate of how much spare capacity there is when the system is not yet congested.

For detecting bottlenecks in our storage implementation, we have a congestion monitor that uses differences in queue lengths over time to extrapolate ingestion rate. We cannot use the data window measurement to monitor data cubes because cubes can batch writes. With batching, performance is not linear with respect to the data volume in each window. Queue monitoring can detect congestion quickly, and the rate of queue growth indicates how congested the system is. However, it does not indicate how much spare capacity there is if the system is not overloaded and the queue is empty.

Congestion monitors report their capacity ratio upstream, both to other congestion monitors on the same host and across the network using a metadata message.

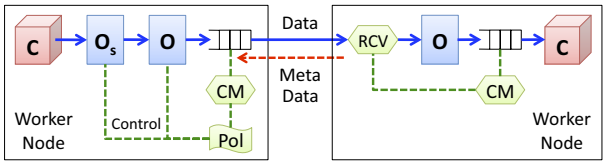


Figure 5: JetStream's mechanisms for detecting and adapting to congestion. Along with cubes and operators, JetStream employs explicit application queues, congestion monitors (CM), policy managers (Pol), and network receivers (RCV) to control when adaptations should be performed.

```
g = QueryGraph()
dest = Cube(g, "stored_images")
dest.add_dimension(TIME, "timestamp")
dest.add_dimension(HOSTNAME, "timestamp")
dest.add_aggregate(BLOB, "img_data")
dest.instantiate_on(union_node)

for node in source_nodes:
  reader = Operators.FileReader(g, options.dirname)
  reader.instantiate_on(node)
  add_time = Operators.Timestamp()
  add_host = Operators.Extend("$HOSTNAME")
  drop_frames = Operators.LowerFramerate()
  downsample = Operators.Downsample(min=0.5)
  g.chain([reader, add_time, add_host, drop_frames,
      downsample, dest])
  g.congest_policy([downsample, drop_frames])
```

Figure 6: Running example with a degradation policy added: downsampling will be applied first, then frame-rate lowering.

When a monitor is queried, it returns the minimum of its own and the downstream ratio. In this way, congestion signals propagate to sources in a multi-step pipeline.

4.3 Congestion response policies

Congestion response policies tune degradation operators based on the available bandwidth reported by congestion monitors. To effectively control data volumes while minimizing errors, policies need to be able to tune multiple degradation operators.

Oftentimes, a given degradation technique is only useful up to a certain level of degradation. Rolling up request logs from a 10-second level to a 30-second level may impose an acceptable delay for a data analysis pipeline, but waiting for longer periods of time may not. If the bandwidth required at the largest-acceptable roll-up coarseness is still too large, it is necessary to use some other technique, such as dropping statistics about unpopular items.

Consider the following policy: "By default, send all images at maximum fidelity from CCTV cameras to a central repository. If bandwidth is insufficient, switch to sending images at 75% fidelity, then 50% if there still isn't enough bandwidth. Beyond that point, reduce the frame rate, but keep the images at 50% fidelity." This policy involves two degradation operators: one to decrease image fidelity (F), and one to drop frames (D). As the system encounters congestion, F should respond first, followed by D once F is fully degraded. However, if capacity becomes available, D should respond first, followed by F. We say that F has higher priority than D.

Figure 6 shows our running example, modified with this policy. The policy is represented in two parts. Each degradation operator is configured with its maximum degradation level. Because this is part of the operator configuration, it can use operator-specific notions, such as frame rates. A separate policy statement specifies the priority of the operators. This policy statement is agnostic to the semantics of the operators in question.

At runtime, each policy is encapsulated in a policy object. Each policy object is attached to a particular congestion monitor and the set of local degradation operators that it manages. The operators periodically query the policy with their available and current degradation levels (per §4.1). The policy returns the degradation level the operator should use, based on the current congestion ratio and the state of the other operators.

Importantly, an operator's priority is unrelated to the operator's position in the dataflow graph. In this example, the dataflow structure does per-frame degradation after frame dropping, to avoid wasted computation on frames that will later be dropped. This is irrespective of which operator has higher priority in the policy.

Our framework can be extended to cope with still-more-complex policies. One might, for instance, desire policies that apply two different degradations simultaneously, such as interleaving time-roll-up steps with some other degradation operation.

5 Degrading Quantitative Data

Above, we discussed the abstractions that JetStream provides for data degradation. We now discuss how these abstractions can be used to degrade quantitative analytics data. While the techniques we describe are mostly well-known, we evaluate them in the new context of wide-area streaming analytics.

A wide range of degradations are possible for quantitative data. Here are some that JetStream supports:

- **Dimension coarsening:** A subscriber that performs roll-ups of data cube dimensions. To reduce output size, the subscriber emits progressively coarser data. For example, rolling up per-second data to output per-minute data or rolling up URLs to the domain level.
- **Local value threshold:** A filter that only forwards elements whose value is above a (tunable) threshold on a particular node. For example, only passing Apache request log entries where the latency of the request exceeded 1 second.

- **Global value threshold:** A filter that only forwards elements whose total value, across all nodes, is above a threshold. This is implemented using a multi-round distributed protocol [7]. For example, one can create a filter that only keeps the top 1000 URLs requested across a CDN in a 10-second window.
- **Consistent sampling:** Drops a fraction of inputs based on the hash of some dimension. Two filters with the same selectivity will pass the same elements.
- **Synopsis approximation:** Replaces a histogram, sketch, or other statistical synopsis with a less accurate but smaller synopsis.

5.1 Which degradations to use?

The best degradation for a given application depends not only on the statistics of the data, but also on the set of queries that may be applied to the data. As a result, the system cannot choose the best degradation without knowing this intended use. We leave for future work the challenge of automatically synthesizing a degradation strategy based on the data and a set of potential queries or other downstream uses of the data. Here, we offer some guidelines for data analysts on which degradations to use for different circumstances.

- For data with synopses that can be degraded, such as histograms or sketches, degrading the synopsis will have predictable bandwidth savings and (for some synopses) predictable increases in error terms.
- If the dimension values for the data have a natural hierarchy, then aggregation can effectively reduce the data volume for many distributions. This is particularly applicable to time-series data. As we discussed in §3.3, coarsening is ineffective on long-tailed distributions, however.
- A long-tailed distribution has a small number of highly ranked "head" items, followed by a long tail of low-ranked items. Many queries, such as "top k items" only concern the head. In these cases, a filter can remove the large-but-irrelevant tail. (Note that this depends on the user's subsequent plans for the data, and not just the statistics of the data itself.) Either a local or global value threshold is applicable here.
- A global value threshold gives exact answers for every item whose total value is above a threshold; a local threshold will have worse error bounds, but can do well in practice in many cases. (We demonstrate this empirically in Sec. 7.3.)
- Consistent samples help to analyze relationships between multiple aggregates. For example, to analyze the correlation between hit count and maximum response latency in a CDN, a query can sample from the space of URLs. This yields exact values for the aggregates of all the URLs in the sample.

One would like degradation operators to have a predictable bandwidth savings, so that JetStream can pick the appropriate degradation level. One would also like degradations to have predictable effects on data quality, so that users can reason about the errors introduced. Most degradation operators have only one or the other of these properties, however. Because no one degradation is optimal, we took pains to allow compound policies. As a result, users can specify an initial degradation with good error bounds, and a fall-back degradation to constrain bandwidth regardless.

5.2 Merging heterogeneous data

As we discussed above, degradation levels will vary over time, and will vary across different nodes feeding into a cube at the same time. It is therefore desirable to be able to combine data of different fidelities without paying an additional penalty. We call this property *mergeability*. Formally, we define mergeability as the ability to combine data of different fidelities so that the merged result has the same error bounds as the input with the lowest fidelity.[1]

Mergeability constrains the ways in which data can be approximated and analyzed. Suppose that two sources are sending data every five seconds to a central point. One source then switches to sending data every six seconds. To represent the merged answer accurately, the data must be further coarsened, to every 30 seconds. This implies that the system sent 5.5x more data than it needed to if both sources had simply sent every 30 seconds.

Mergeability guided several design choices in JetStream. Dimension coarsening is only mergeable if the coarsening steps are consistent and strictly hierarchical. We therefore define an explicit hierarchy for time dimensions. Cubes can store (and subscribers can roll up) time only in fixed intervals. (The first layers of the hierarchy are 1, 5, 10, 30, and 60 seconds.) We also require slices to start at a timestamp that is a multiple of the query's step size. Taken together, these requirements make time coarsening mergeable.

Mergeability constrains other degradations besides dimension coarsening. Histograms are an effective approximation for computing quantiles. Making our histogram implementation mergeable required careful programming. In our implementation, every division between buckets in an n-bucket histogram is also a division in an n+1 bucket histogram, and consequently histograms are mergeable regardless of the number of buckets.

Some degradations are intrinsically not mergeable. As an example, consider web request data generated at two servers. Lets analyze the top-k request counts by URL

[1] This definition is stronger than that typically used in the theory streaming literature, which covers merging data of the same fidelity [3].

at *both* servers, so that we sum the counts for any URLs common to both. For such a query, the top-k lists from each server cannot be merged together.[2] Similarly, for per-minute top-k lists, there is in principle no way to compute a daily top-k. For the same reasons, sets with a value cutoff ("elements with value above x") cannot always be merged to give a correct set with a new value cutoff. This is tolerable in practice for applications that do not need to perform roll-ups, or where the data distribution is such that the error bounds are small. A global filtering protocol, such as the multi-round protocol supported by JetStream, can give correct results when data is spread across many sources.

6 Architecture and Implementation

JetStream's architecture has three main components: worker daemons access and process data on a distributed set of nodes, a coordinator daemon distributes computation across available workers, and a client library defines the computation to be performed and provides an interface to the running system.

The life-cycle of a query: A query starts when a client program creates a dataflow graph and submits it for execution. The library checks the dataflow graph for type and structural errors, such as integer operators being applied to string fields or filter operators with no inputs. The graph is then sent to the coordinator, which chooses the assignment of operators to worker nodes. The placement algorithm attempts to minimize the data traversing wide-area networks, placing operators on the send-side of the link whenever possible.

After determining operator placement, the coordinator sends the relevant subset of the graph to each node in the system. The nodes then create any necessary network connections amongst themselves, and start the operators. The query will keep running until it is stopped via the coordinator, or until all the sources send a `stop` marker indicating that there will be no more data. As discussed above, degradation is handled in a decentralized fashion, and the coordinator's involvement is not required to maintain a running query.

Implementation: The JetStream coordinator is implemented in about 2000 lines of Python, and is purely memory resident. The worker is implemented in a single process, similar to a database management system. The worker is about 15,000 lines of C++, including 3000 for system-defined operators. Operators are implemented as C++ classes, and can be dynamically loaded. Within a node, each chain of operators from a source operator or network link is executed sequentially, without queuing, by a single thread at a time. All tuples are processed sequentially in the order received.

Failure Recovery: In a streaming system, failure recovery requires two things: Each failed piece of the computation (e.g., each query operator on a failed node) must be restarted and reattached to the graph. Additionally, for stateful pieces of the computation, their state must be restored to what it would have been absent the failure. JetStream currently only does the former. The latter can be implemented within our model but is a lower priority for us than for datacenter streaming systems.

The coordinator has a complete view of which operators should be on each node, and therefore can restart them when a node fails and recovers. This is implemented in our existing prototype. Sources periodically try to reconnect to their destinations and therefore will automatically recover when the failed node restarts. The coordinator's state can be made durable using group-consensus tools like Zookeeper [18]. (This is not currently implemented.)

Unlike past streaming systems, we do not attempt to make JetStream failure-oblivious. Datacenter-based streaming systems rely on the presence of an underlying reliable data store (such as a reliable message queue, HDFS or BigTable) that is assumed to be always available [5, 29, 32]. Using this data store, these systems can hide the existence of failures from computations, restarting work immediately and carefully avoiding duplicated or dropped data.

In wide-area analytics, failures cannot be hidden, since data will be inaccessible if the network is partitioned. For many analytic uses, users prefer queries that promptly supply approximate results based on the available data, and revise these results as late data arrives [4]. (We discussed how to incorporate these backfill updates in §3.2.) Thus, the results of a computation will necessarily be affected by failures. Retroactively changing the results to recover completely from the failure is a low priority for us, since enough time might have passed that the temporary results have already been acted upon.

That said, it is possible to do precise failure recovery in the JetStream model, as we sketch below. In the datacenter context, a number of failure recovery techniques have been described. All these schemes have two basic ingredients: (i) The system must keep metadata to track which tuples have been processed by each operator. This metadata must be recorded atomically with the update to ensure process-once semantics. (ii) There must be a mechanism to retransmit the tuple until there is an acknowledgment that it was fully processed by the chain.

Our system meets both these underlying requirements. Since our underlying data store is a database, we already have sufficient transaction support to atomically commit updates along with the appropriate metadata. We could add sequence numbers to tuples and use the fact that

[2] Fagin *et al* [11] prove that it is not generally possible to find the top k_1 elements and their exact values using only the top k_2 from each of the subsets, for any fixed k_2 (see their example 4.4).

	Name	Output	Operators	Cubes	LoC	BW ratio
1	Big requests	Requests above 95th percentile of size, with percentiles computed for past minute.	3n + 8	3n + 3	97	22
2	Slow requests	All requests with throughput less than 10 kbytes/sec.	4n + 2	1	5	24
3	Requests-by-URL	Counts for each URL-response code pair.	3n + 2	n + 1	5	351
4	Success by domain	The fraction of success responses for each domain.	6n + 4	n + 2	30	445
5	Quantiles	95th percentiles of response time and size, for each HTTP response code.	4n + 5	n +1	25	715
6	Top-k domains	Top-10 domains every five seconds.	n + 3	n + 1	40	2300
7	Bad referrers	The 10 domains most responsible for referrals that led to a 404 response, every five seconds.	8n + 2	n + 1	16	18600
8	Bandwidth by node	Overall bandwidth used by each node, over time.	4n + 2	n + 1	15	49800

Table 2: Complexity and efficiency of example queries. For operator and cube counts, *n* is the number of data source nodes in the system. LoC is the number of non-comment lines of code needed to write the query, not counting shared library code. Bandwidth ratio is the ratio between source input size and the data volume transferred over the network (e.g., 22 means a factor-of-22 savings from aggregation and filtering).

tuples are not reordered between data cubes to make sure that upstream cubes retransmit data to downstream cubes until appropriate acknowledgments are received.

7 Evaluation

In this section, we evaluate four main questions about JetStream's design.

- §7.1 Does JetStream make it easy to write distributed analytic queries?
- §7.1 How effectively do hierarchical aggregation and static filtering reduce bandwidth consumption?
- §7.2 What latency can JetStream maintain, in the presence of changing bandwidth limits?
- §7.3 How well does JetStream's adaptive degradation work with complex policies?

Given JetStream's focus on adapting bandwidth consumption, we omit extensive performance benchmarking and comparisons to existing streaming systems. The throughput of JetStream is largely limited by the underlying database and serialization code, neither of which is relevant to our technical contributions.

7.1 Expressivity and Efficiency

To evaluate the usability of JetStream's programming framework, we use it to ask a number of analytical questions about a dataset of CoralCDN logs. Table 2 summarizes the eight queries we evaluate, drawn from our experience observing and managing CoralCDN. The queries include summary counts, histograms, filtered raw logs, top-k queries, and outlier detection.

Our aim is to understand (i) if these questions can be expressed succinctly in JetStream's programming model, and (ii) if many of our queries, even before applying data degradation techniques, experience significant bandwidth savings by storing data near where it is generated.

To test the queries, we select logs from 50 nodes in CoralCDN for January 20, 2013, and transfer each log to a node on the VICCI testbed [26]. To emulate wide-area clusters of CDN nodes, we select 25 VICCI servers from each of the MPI-SWS and Georgia Tech sites (in Saarbruecken, Germany, and Atlanta, GA respectively). These nodes serve as the sources of data; the queries produce their output at a node in Princeton, NJ. The total size of these logs is 51 GB, or 140 million HTTP requests. Since the logs are drawn from actual operational nodes, they are not equal in size. The largest is approximately 2 GB, while the smallest is 0.4 GB. This sort of size skew is a real factor in operational deployments, not an experimental artifact.

We observe large savings compared to backhauling the raw logs, ranging from a factor of 22x to more than four orders of magnitude. The large gains are primarily due to the partial aggregation present in all these examples. Thousands of requests can be tallied together in source cubes to produce a single tuple to be copied across the wide area for merging at the union cube. This demonstrates that edge storage and partial aggregation are valuable design choices in wide-area analytics.

Code size is small but varies across queries. We wrote about 150 lines of shared code for processing command line arguments, parsing and storing CoralCDN logs, and printing results. These are shared lines of code and are therefore not included in the unique LoC measures above. For simple aggregation tree queries, only a few lines are needed, specifying what is to be aggregated. Queries with more complex topologies require more code. In our experience, the code size and complexity is comparable to that of MapReduce programs. As with MapReduce, higher-

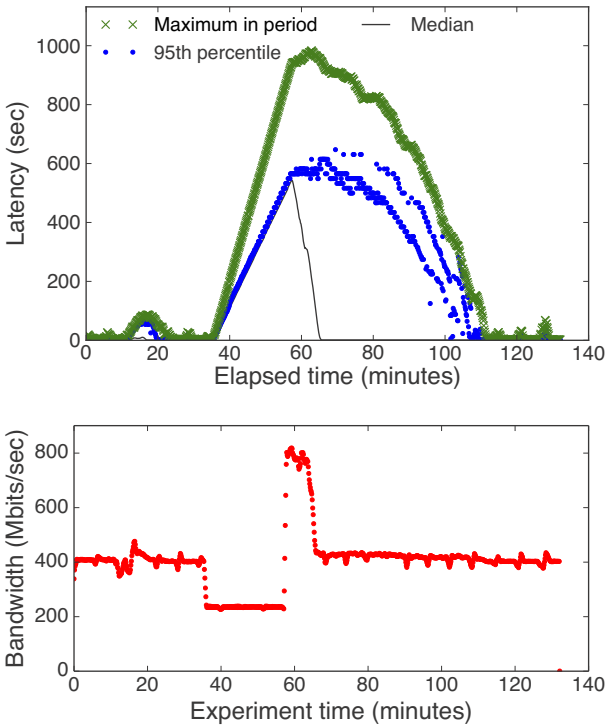


Figure 7: (Top) Latencies, with percentiles computed over each 8-second window. (Bottom) Data rate at receiver, showing extent and duration of traffic shaping. Without degradation, latency grows without bound, and takes a long time to recover from disruption.

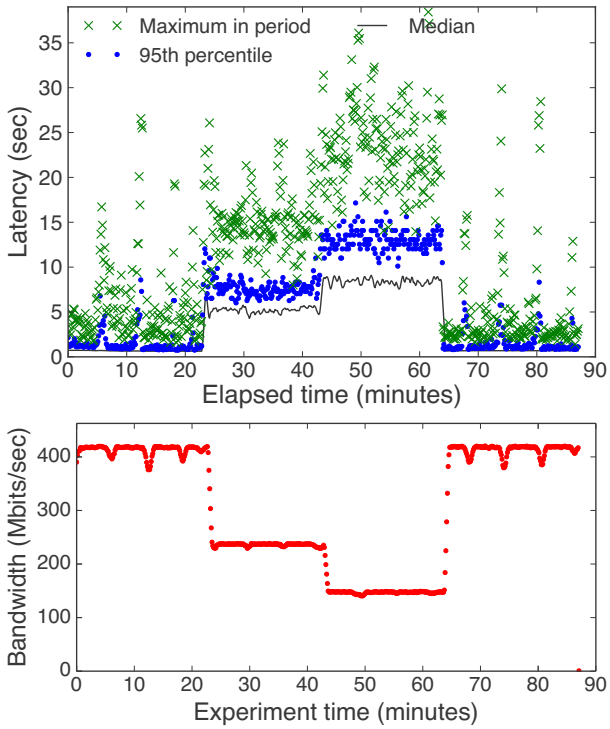


Figure 8: The same as Figure 7, but with degradation enabled. The system rapidly adjusts to available bandwidth, keeping latency low and bounded.

level frameworks could further reduce the code size and complexity. Most programming mistakes were caught by the client-side type checker, reducing the difficulty and time cost of development.

Only two user-defined functions (UDFs) are required for these sample queries; one to convert URLs to domains and a second to compute the success ratio in Query #4. The parsing is performed with a generic configurable operator for parsing field-separated strings. This suggests that the cube API plus our existing operators are sufficient to express a wide range of tasks.

We have preliminary evidence that JetStream's programming model can be learned by non-experts and does not require knowledge of the system internals. Query #3 was written by an undergraduate who did not participate in JetStream's development, and received only limited assistance from the core development team.

7.2 System Throughput and Latency

We benchmark the system's overall throughput and latency characteristics using a relatively simple processing pipeline under several different network configurations. This experiment used 80 source nodes running on the VICCI infrastructure, divided between MPI-SWS (Germany), Georgia Tech and the University of Washington (United States). The source nodes send image data to a single union node at Princeton. All images are the same size (approximately 26 kilobytes). Nodes are configured to send a maximum of 25 images per second, a rate that the network can support without degradation. This is a 400 mb/sec data rate, so we are using nearly half our institution's gigabit WAN link during the experiment. The configured degradation policy is to reduce the frame rate if bandwidth is insufficient.

Figure 7 shows the behavior of the system with degradation disabled. Generally, latency is low: However, around minute 15, a slight drop in available bandwidth resulted in some nodes experiencing uncontrolled queue growth, leading to significant latency. (This is visible as a small bump in the latency plot.) At minute 35, we impose a 400 kbit/sec bandwidth cap on each source node using the Linux kernel's traffic shaping options. The latency of all the nodes starts rising sharply and continuously. Around minute 60, we disable bandwidth shaping and latency starts to drop. Notice that the 95th percentile and maximum latency recovers much more slowly than median latency. Some nodes are able to drain their queues quickly, while other nodes are starving for bandwidth. As a result, it takes roughly 45 minutes for the system to resume its previous behavior.

Figure 8 shows that our degradation mechanisms prevent these unwanted behaviors. We repeated a similar

experiment, but with degradation enabled. Here, after we apply bandwidth shaping, the degradation mechanisms activate, and successfully keep queue size and latency bounded at a few seconds. Approximately 40 minutes into the experiment, we apply more bandwidth shaping, to 250 kb/sec per node, and again latency stays bounded. Notice the absence of a latency spike at each of the bandwidth transitions; the system reacts promptly enough that such spikes are not even visible on the graph. At the 60 minute mark we disable traffic shaping, and the system again reacts promptly, returning to its original state.

7.3 Complex Degradation Policies

In our final experiment, we demonstrate that JetStream's degradation mechanisms, operating on a realistic workload, maintain responsiveness in situations in which full-fidelity data would exceed available resources. For this experiment, use the Requests-by-Url query from Table 2, executing under the same setup as in §7.1. We compress a full day's logs (and thus the diurnal variation) into five minutes of wall-clock time. We impose a bandwidth cap of 80 kb/second per source node, which is artificially low, but serves to emphasize system behavior. The limit is low enough to force the configured degradation policies to activate as data rates shift.

We compare the effect of four degradation policies. Each policy starts by sending data every second, if possible, and performing roll-ups to five-second windows when bandwidth is scarce. One policy (*max window 5*) does no further degradation. The *max window 10* policy will further degrade to 10-second windows. The remaining two policies employ either consistent-sampling (based on a hash of the URL) or a local value threshold (dropping tuples with the lowest counts).

Figure 9 shows the bandwidth and degradation from each of the four policies and the bandwidth used by the null policy (no degradation). As the load increases, most of the source nodes hit the bandwidth cap and switch to 5-second windows. As the load keeps rising, the more heavily-used nodes again reach their cap. Both the thresholding and sampling policies can keep bandwidth usage under the cap.

We noted earlier that many CoralCDN URLs are unique, and therefore do not aggregate well. This is visible in the results here. Time coarsening by a factor of 5 and 10 reduces bandwidth, but by factors much less than 5 or 10. Much bandwidth is used reporting statistics about low-traffic URLs—the tail of the distribution. Local value thresholding effectively truncates this tail. As can be seen on the graph, this has a minuscule effect on the relative error, while reducing bandwidth by a factor of two. (In this context, relative error is the maximum error in the request count for any one URL as a fraction of the total number of requests in that time period).

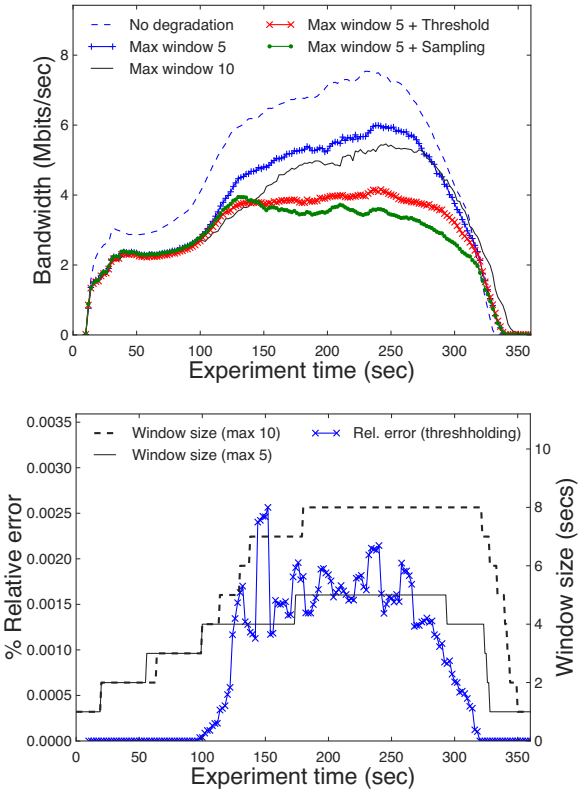


Figure 9: (Top) Output bandwidth for five different degradation choices. (Bottom) Window size and thresholding error over time. For this query, window sizes larger than 5 seconds have limited benefit, while thresholding has minimal accuracy penalty. Degradation policies should depend on the data.

This experiment shows that JetStream is able to effectively combine different degradation techniques. The limits of time coarsening on this workload illustrates why compound policies are useful. The fact that different degradations activate at the same point in the experiment shows that the control loop works effectively with a variety of degradation operators.

8 Related Work

Our work has a large debt to the stream processing community. The original work on streaming [2, 6, 8] addressed the question of how to process incoming updates with minimal latency. In contrast, JetStream targets dispersed and changing (stored) data sets, in the presence of dynamic bandwidth constraints.

Spark-Streaming, MillWheel, and Storm [5, 29, 32] are systems for large-scale stream processing within datacenters. All these systems rely on an underlying fault-tolerant storage system, respectively HDFS, BigTable, or a reliable message queue. Such systems or their implementation techniques, such as Spark's memory-backed resilient storage, could help scale JetStream within datacenters,

but are orthogonal to our concerns of efficiency and low latency across the wide area.

Some research on streaming systems has considered the wide area [19, 27]. One focus of this work was on using redundant paths for performance and fault tolerance. In contrast, we use degradation to cope with insufficient bandwidth. Hourglass [27] places query operators to minimize network usage in an ad-hoc topology, but since it does not include storage, users must choose which data to collect (and thus the set of supported queries) beforehand.

Previous wide-area streaming work assumed that computation resources were scattered in an ad-hoc manner, e.g., on PlanetLab nodes. As a result, sophisticated algorithms were needed for placement. This assumption is overly pessimistic in our context. Due to the rise of centralized datacenters, we expect there to be only two or three options for operator or cube placement: the site where the data is generated, the nearest point-of-presence, or else a centralized datacenter.

Some single-node stream-processing systems, such as TelegraphCQ [8], included relational storage, and others have advocated tighter integration between stream processing and relational databases [12]. The StreamCube system evaluated which layers of a cube hierarchy to materialize in the context of stream processing [17]. These uses of storage do not pose the latency-completeness tradeoffs we address with our subscriber interface, nor do they facilitate bandwidth reduction in distributed contexts.

Tree aggregation has been studied in the sensor network community as a method of reducing bandwidth and power consumption, notably in the Tiny Aggregation Service [22]. Much subsequent sensor network research used mesh topologies to compensate for unreliable connections and faulty nodes. In contrast, our hardware is not power-constrained and we assume that conventional IP networking will deliver suitable routes. Protocols such as RCRT [25], the Rate-Controlled Reliable Transport Protocol for sensor networks, estimate available bandwidth explicitly and convey rate allocation decisions to data sources. They could serve as an alternative implementation of the congestion monitor in JetStream. However, these works do not address how the application reacts to the congestion signals, which in JetStream is specified by the degradation operators, the policy manager, and the interface between them.

Tree aggregation and local storage are also used in the Ganglia [14] monitoring system. Ganglia supports a limited set of queries and is oblivious to bandwidth conditions.

Our work seeks to reduce data volumes while minimizing the reduction in accuracy. Similarly, BlinkDB [4] deploys sampling-based approximations on top of MapReduce and Hive to reduce latency. In BlinkDB, the data is carefully pre-sampled with specific statistical goals; small probing jobs are used to estimate query run-time. In contrast, streaming wide-area analytics systems such as ours have to measure and adapt to available bandwidth, without the benefit of a prior data-import step. We also support a range of degradation techniques, not just sampling.

Our previous workshop publication [28] argued for decentralized wide-area analysis of the form conducted by JetStream. It gave a high-level description of our ideas and discussed use cases at length, but lacked this paper's detailed design, implementation, or evaluation.

9 Conclusions

This paper has presented a system, JetStream, for wide-area data analysis in environments where bandwidth is scarce. Our storage abstraction allows data to be stored where it was generated and efficiently queried when needed. It simplifies aggregation of data both across time and across sources. Degradation techniques supplement aggregation when available bandwidth is insufficient for error-free results.

No single degradation technique is always best; a combination of techniques can perform better than any individual technique. Thus, our system supports combining multiple techniques in a modular and reusable way using policies. Our separation between congestion monitors, degradation operators, and policies creates a powerful, extensible framework for streaming wide-area analysis.

Acknowledgments

The authors appreciate the helpful advice and comments of Jinyang Li, Jennifer Rexford, Erik Nordström, Rob Kiefer, our shepherd Ramesh Govindan, and the anonymous reviewers. This work was funded under NSF awards IIS-1250990 (BIGDATA) and CNS-1217782, as well as the DARPA CSSG program.

References

[1] 3GPP Technical Specification 26.234. Transparent end-to-end packet switched streaming service (PSS); Protocols and codecs, 2013.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[3] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. In *PODS*, 2012.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, 2013.

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

[6] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, Y. Park, and C. Venkatramani. SPC: A distributed, scalable platform for data mining. In *DM-SSP*, 2006.

[7] P. Cao. Efficient top-k query calculation in distributed networks. In *PODC*, 2004.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, 2003.

[9] Y. M. Chen, L. Dong, and J.-S. Oh. Real-time video relay for uav traffic surveillance systems through available communication networks. In *IEEE WCNC*, 2007.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[12] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre. Continuous analytics: Rethinking query processing in a network-effect world. In *CIDR*, 2009.

[13] M. J. Freedman. Experiences with CoralCDN: A five-year operational view. In *NSDI*, 2010.

[14] Ganglia monitoring system. `http://ganglia.sourceforge.net/`, 2013.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[16] L. Guo, E. Tan, S. Chen, Z. Xiao, and X. Zhang. The stretched exponential distribution of internet media access patterns. In *PODC*, 2008.

[17] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai. Stream cube: An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases*, 18(2), 2005.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[19] J.-H. Hwang, U. Cetintemel, and S. B. Zdonik. Fast and reliable stream processing over wide area networks. In *Data Eng. Workshop*, 2007.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.

[21] D. Katabi, M. Handley, and C. E. Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM*, 2002.

[22] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[23] F. McSherry, D. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[25] J. Paek and R. Govindan. RCRT : Rate-Controlled Reliable Transport Protocol for Wireless Sensor Networks. *ACM Trans. Sensor Networks (TOSN)*, 7(3), 2010.

[26] L. Peterson, A. Bavier, and S. Bhatia. VICCI: A programmable cloud-computing research testbed. Technical Report TR-912-11, Princeton Univ., Dept. Comp. Sci., 2011.

[27] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[28] A. Rabkin, M. Arye, S. Sen, V. Pai, and M. J. Freedman. Making every bit count in wide-area analytics. In *HotOS*, May 2013.

[29] Storm. `https://github.com/nathanmarz/storm/`, 2012.

[30] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowledge and Data Eng.*, 15 (3):555–568, 2003.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[32] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *HotCloud*, 2012.

GRASS: Trimming Stragglers in Approximation Analytics

Ganesh Ananthanarayanan[1], Michael Chien-Chun Hung[2], Xiaoqi Ren[3], Ion Stoica[4], Adam Wierman[3], Minlan Yu[2]

[1]*Microsoft Research*, [2]*University of Southern California*, [3]*California Institute of Technology*,
[4]*University of California, Berkeley*

ga@microsoft.com, {chienchun.hung, minlanyu}@usc.edu, {xren, adamw}@caltech.edu, istoica@cs.berkeley.edu

Abstract

In big data analytics, timely results, even if based on only part of the data, are often *good enough*. For this reason, *approximation* jobs, which have deadline or error bounds and require only a subset of their tasks to complete, are projected to dominate big data workloads. Straggler tasks are an important hurdle when designing approximate data analytic frameworks, and the widely adopted approach to deal with them is speculative execution. In this paper, we present GRASS, which carefully uses speculation to mitigate the impact of stragglers in approximation jobs. GRASS's design is based on first principles analysis of the impact of speculation. GRASS delicately balances immediacy of improving the approximation goal with the long term implications of using extra resources for speculation. Evaluations with production workloads from Facebook and Microsoft Bing in an EC2 cluster of 200 nodes shows that GRASS increases accuracy of deadline-bound jobs by 47% and speeds up error-bound jobs by 38%. GRASS's design also speeds up exact computations (zero error-bound), making it a *unified solution for straggler mitigation*.

1 Introduction

Large scale data analytics frameworks automatically compose *jobs* operating on large data sets into many small *tasks* and execute them in parallel on *compute slots* on different machines. A key feature catalyzing the widespread adoption of these frameworks is their ability to guard against failures of tasks, both when tasks fail outright as well as when they run slower than the other tasks of the job. Dealing with the latter, referred to as *stragglers*, is a crucial design component that has received widespread attention across prior studies [1, 2, 3].

The dominant technique to mitigate stragglers is *speculation*—launching speculative copies for the slower tasks, where a speculative copy is simply a duplicate of the original task. It then becomes a race between the original and the speculative copies. Such techniques are state-of-the-art and deployed in production clusters at Facebook and Microsoft Bing, thereby significantly speeding up jobs. The focus of this paper is on speculation for an emerging class of jobs: *approximation* jobs.

Approximation jobs are starting to see considerable interest in data analytics clusters [4, 5, 6]. These jobs are based on the premise that providing a timely result, even if only on part of the dataset, is more important than processing the entire data. These jobs tend to have *approximation bounds* on two dimensions—deadline and error [7]. *Deadline-bound* jobs strive to maximize the accuracy of their result within a specified time deadline. *Error-bound* jobs, on the other hand, strive to minimize the time taken to reach a specified error limit in the result. Typically, approximation jobs are launched on a large dataset and require only a *subset* of their tasks to finish based on the bound [8, 9, 10].

Our focus is on the problem of speculation for approximation jobs.[1] Traditional speculation techniques for straggler mitigation face a fundamental limitation when dealing with approximation jobs, since they do not take into account approximation bounds. Ideally, when the job has many more tasks than compute slots, we want to prioritize those tasks that are likely to complete within the deadline or those that contribute the earliest to meeting the error bound. By not considering the approximation bounds, state-of-the-art straggler mitigation techniques in production clusters at Facebook and Bing fall significantly short of optimal mitigation. They are 48% lower in average accuracy for deadline-bound jobs and 40% higher in average duration of error-bound jobs.

Optimally prioritizing tasks of a job to slots is a classic scheduling problem with known heuristics [11, 12, 13]. These heuristics, unfortunately, do not directly carry over to our scenario for the following reasons. First, they calculate the optimal ordering statically. Straggling of tasks, on the other hand, is unpredictable and necessitates dynamic modification of the priority ordering of tasks according to the approximation bounds. Second, and most importantly, traditional prioritization techniques assign tasks to slots assuming every task to occupy only one slot. Spawning a speculative copy, however, leads to the same task using two (or multiple) slots simultaneously. Hence, this distills our challenge

[1]Note that an error-bound job with error of zero is the same as an exact job that requires all its tasks to complete. Hence, by focusing on approximation jobs, we automatically subsume exact computations.

to *achieving the approximation bounds by dynamically weighing the gains due to speculation against the cost of using extra resources for speculation*.

Scheduling a speculative copy helps make immediate progress by finishing a task faster. However, while not scheduling a speculative copy results in the task running slower, many more tasks may be completed using the saved slot. To understand this *opportunity cost*, consider a cluster with one unoccupied slot and a straggler task. Letting the straggler complete takes five more time units while a new copy of it would take four time units. While scheduling a speculative copy for this straggler speeds it up by one time unit, if we were not to, that slot could finish another task (taking five time units too).

This simple intuition of opportunity cost forms the basis for our two design proposals. First, Greedy Speculative (GS) scheduling is an algorithm that *greedily* picks the task to schedule next (original or speculative) that most improves the approximation goal at that point. Second, Resource Aware Speculative (RAS) scheduling considers the opportunity cost and schedules a speculative copy only if doing so saves both time *and* resources.

These two designs are motivated by first principles analysis within the context of a theoretical model for studying speculative scheduling. An important guideline from our model is that the value of being greedy (GS) increases for smaller jobs while considering opportunity cost of speculation (RAS) helps for larger jobs. As our model is generic, a nice aspect is that the guideline holds not only for approximation jobs but also for exact jobs that require all their tasks to complete.

We use the above guideline to dynamically combine GS and RAS, which we call GRASS. At the beginning of a job's execution, GRASS uses RAS for scheduling tasks. Then, as the job gets *close* to its approximation bound, it switches to GS, since our theoretical model suggests that the opportunity cost of speculation diminishes with fewer unscheduled tasks in the job. GRASS learns the point to switch from RAS to GS using job and cluster characteristics.

We demonstrate the generality of GRASS by implementing it in both Hadoop [14] (for batch jobs) and Spark [15] (for interactive jobs). We evaluate GRASS using production workloads from Facebook and Bing on an EC2 cluster with 200 machines. GRASS increases accuracy of deadline-bound jobs by 47% and speeds up error-bound jobs by 38% compared to state-of-the-art straggler mitigation techniques deployed in these clusters (LATE [2] and Mantri [1]). In fact, GRASS results in near-optimal performance. In addition, GRASS also speeds up exact jobs, that require all their tasks to complete, by 34%. Thus, it is a *unified speculation solution for both approximation as well as exact computations*.

2 Challenges and Opportunities

Before presenting our system design, it is important to understand the challenges and opportunities for speculating straggler tasks in the context of approximation jobs.

2.1 Approximation Jobs

Increasingly, with the deluge of data, analytics applications no longer require processing entire datasets. Instead, they choose to tradeoff accuracy for response time. *Approximate* results obtained early from just part of the dataset are often *good enough* [4, 6, 5]. Approximation is explored across two dimensions—time for obtaining the result (deadline) and error in the result [7].

- *Deadline-bound* jobs strive to maximize the accuracy of their result within a specified time limit. Such jobs are common in real-time advertisement systems and web search engines. Generally, the job is spawned on a large dataset and accuracy is proportional to the fraction of data processed [8, 9, 10] (or tasks completed, for ease of exposition).

- *Error-bound* jobs strive to minimize the time taken to reach a specified error limit in the result. Again, accuracy is measured in the amount of data processed (or tasks completed). Error-bound jobs are used in scenarios where the value in reducing the error below a limit is marginal, *e.g.,* counting of the number of cars crossing a section of a road to the nearest thousand is sufficient for many purposes.

Approximation jobs require schedulers to *prioritize the appropriate subset of their tasks* depending on the deadline or error bound. Prioritization is important for two reasons. First, due to cluster heterogeneities [2, 3, 16], tasks take different durations even if assigned the same amount of work. Second, jobs are often *multi-waved*, i.e., their number of tasks is much more than available compute slots, thereby they run only a fraction of their tasks at a time [17]. For example, when a job with 1000 tasks is given only 100 slots simultaneously (due to, say, fair scheduling), it runs only one-tenth of its tasks at a time. These tasks, though, are independent and can be scheduled in any order. The trend of multi-waved jobs is expected to grow with smaller tasks [18].

2.2 Challenges

The main challenge in prioritizing tasks of approximation jobs arises due to some of them *straggling*. Even after applying many proactive techniques, in production clusters in Facebook and Microsoft Bing, the average

job's slowest task is eight times slower than the median.[2] It is difficult to model all the complex interactions in clusters to prevent stragglers [3, 20]. Ananthanarayanan et al. (Section 2.1.2 in [3]) also show that blacklisting machines based on their likeliness to cause stragglers (in both the short- as well as long-term) has little benefits; machines are neither consistently problematic nor exhibit simple correlations with task durations.

The widely adopted technique to deal with straggler tasks is *speculation*. This is a reactive technique that spawns speculative copies for tasks deemed to be straggling. The earliest among the original and speculative copies is picked while the rest are killed. While scheduling a speculative copy makes the task finish faster and thereby increases accuracy, they also compete for compute slots with the unscheduled tasks.

Therefore, our problem is to *dynamically prioritize tasks based on the deadline/error-bound while choosing between speculative copies for stragglers and unscheduled tasks*. This problem is, unfortunately, NP-Hard and devising good heuristics (i.e., with good approximation factors) is an open theoretical problem.

2.3 Potential Gains

Given the challenges posed by stragglers discussed above, it is not surprising that the potential gains from mitigating their impact are significant. To highlight this we use a simulator with an optimal bin-packing scheduler. Our baselines are the the state-of-the-art mitigation strategies (LATE [2] and Mantri [1]) in the production clusters. Optimally prioritizing the tasks while correctly balancing between speculative copies and unscheduled tasks presents the following potential gains. Deadline-bound jobs improve their accuracy by 48% and 44%, in the Facebook and Bing traces, respectively. Error-bound jobs speed up by 32% and 40%. We next develop an online heuristic to achieve these gains.

3 Speculation Algorithm Design

The key choice made by a cluster scheduling algorithm is to pick the next task to schedule given a vacant slot. Traditionally, this choice is made among the set of tasks that are queued; however when speculation is allowed, the choice also includes speculative copies of tasks that are already running. This extra flexibility means that a design must determine a prioritization that carefully weighs the gains from speculation against the cost of extra resources while still meeting the approximation goals. Thus, we first focus on tradeoffs in the design of the speculation policy. Specifically, using both small examples and analytic modeling we motivate the use of two simple heuristics, Greedy Speculative (GS) scheduling and Resource Aware Speculative (RAS) scheduling that together make up the core of GRASS.

[2]Task durations are normalized by their input sizes to be resistant to data skews [19, 1].

3.1 Speculation Alternatives

For simplicity, we first introduce GS and RAS in the context of deadline-bound jobs and then briefly describe how they can be adapted to error-bound jobs.

3.1.1 Deadline-bound Jobs

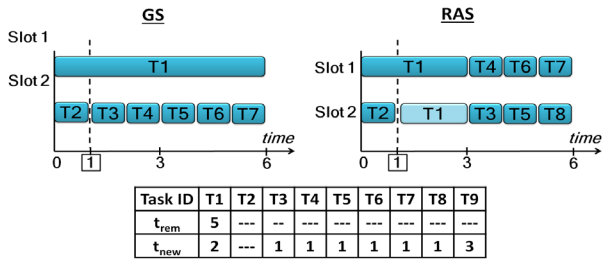
If speculation was not allowed, there is a natural, well-understood policy for the case of deadline-bound jobs: Shortest Job First (SJF), which schedules the task with the smallest processing time. In many settings, SJF can be proven to minimize the number of incomplete tasks in the system, and thus maximize the number of tasks completed, at all points of time among the class of non-preemptive policies [11, 12]. Thus, without speculation, SJF finishes the most tasks before the deadline.

If one extends this idea to the case where speculation is allowed, then a natural approach is to allow the currently running tasks to also be placed in the queue, and to choose the task with the smallest size, i.e., t_{new} (requiring, of course, that the task finishes before the deadline). If the chosen task has a copy currently running, we check that the speculative copy being considered provides a benefit, i.e., $t_{new} < t_{rem}$. So, the next task to run is still chosen according to SJF, only now speculative copies are also considered. We term this policy *Greedy Speculative (GS) scheduling*, because it picks the next task to schedule *greedily*, i.e., the one that will finish the quickest, and thus improve the accuracy the earliest *at present*.

Figure 1 (left) presents an illustration of GS for a simple job with nine tasks and two concurrent slots. Tasks T1 and T2 are scheduled first, and when T2 finishes, the t_{rem} and t_{new} values are as indicated. At this point, GS schedules T3 next as it is the one with the lowest t_{new}, and so forth. Assuming the deadline was set to 6 time units, the obtained accuracy is $\frac{7}{9}$ (or 7 completed tasks).

Picking the earliest task to schedule next is often optimal when a job has no unscheduled tasks (i.e., either single-waved jobs or the last wave of a multi-waved job). However, when there are unscheduled tasks it is less clear. For example, in Figure 1 (right) if we schedule a speculative copy of T1 when T2 finished, instead of T3, 8 tasks finish by the deadline of 6 time units.

The previous example highlights that running a speculative copy has resource implications which are important to consider. If the speculative copy finishes early, both slots (of the speculative copy and the original) be-



Task ID	T1	T2	T3	T4	T5	T6	T7	T8	T9
t_{rem}	5	---	--	---	---	---	---	---	---
t_{new}	2	---	1	1	1	1	1	1	3

Figure 1: **GS and RAS for a deadline-bound job with 9 tasks. The t_{rem} and t_{new} values are when T2 finishes. The example illustrates deadline values of 3 and 6 time units.**

come available sooner to start the other tasks. This *opportunity cost* of speculation is an important tradeoff to consider, and leads to the second policy we consider: *Resource Aware Speculative (RAS) scheduling*.

To account for the opportunity cost of scheduling a speculative copy, RAS speculates only if it saves both time *and* resources. Thus, not only must t_{new} be less than t_{rem} to spawn a speculative copy but the sum of the resources used by the speculative and original copies, when running simultaneously, must be less than letting just the original copy finish. In other words, for a task with c running copies, its resource savings, defined as $c \times t_{rem} - (c+1) \times t_{new}$, must be positive.

By accounting for the opportunity cost of resources, RAS can out-perform GS in many cases. As mentioned earlier, in Figure 1 (right) where RAS achieves an accuracy of $\frac{8}{9}$ versus GS's $\frac{7}{9}$ in the deadline of 6 time units. This improvement comes because, when T2 finishes, speculating on T1 saves 1 unit of resource.

However, RAS is not uniformly better than GS. In particular, RAS's cautious approach can backfire if it overestimates the opportunity cost. In the same example in Figure 1, if the deadline of the job were reduced from 6 time units to 3 time units instead, GS performs better than RAS. At the end of 3 time units, GS has led to three completed tasks while RAS has little to show for its resource gains by speculating T1.

As the example alludes to, the value of the deadline and the number of waves are two important factors impact whether GS or RAS is a better choice. A third important factor, which we discuss later in §4.1, is the estimation accuracy of t_{rem} and t_{new}.

Pseudocode 1 describes the details of GS and RAS. The set T consists of all the running and unscheduled tasks of the jobs. There are two stages in the scheduling process: (i) *Pruning Stage:* In this stage (lines $5-12$), tasks that are not slated to complete by the deadline are removed from consideration. Further, GS removes those tasks whose speculative copy is not expected to finish earlier than the running copy. RAS removes those tasks

```
1:  procedure DEADLINE(⟨Task⟩ T, float δ, bool OC)
                          ▷ OC = 1 → use RAS; 0 → use GS
2:      if OC then
3:          for each Task t in T do
4:              if t.running then
                    t.saving = t.c ×t.t_rem − (t.c+1) × t_new
                                                ▷ PRUNING STAGE
        δ' ← Remaining Time to δ
        ⟨Task⟩Γ ← φ
5:      for each Task t in T do
6:          if t.t_new > δ' then continue    ▷ Exceeds deadline
7:          if OC then
8:              if t.saving > 0 then Γ.add(t)
9:          else
10:             if t.running then
11:                 if t.t_new < t.t_rem then Γ.add(t)
12:             else Γ.add(t)
                                            ▷ SELECTION STAGE
13:     if Γ ≠ null then
14:         if OC then SortDescending(Γ, "saving")
15:         else SortAscending(Γ, t_new)
            return Γ.first()
```

Pseudocode 1: **GS and RAS algorithms for deadline-bound jobs (deadline of δ). T is the set of unfinished tasks with the following fields per task: t_{rem}, t_{new}, and a boolean "running" to denote if a copy of it is currently executing. RAS is used when OC is set. At default, both algorithms schedule the task with the lowest t_{new} within the deadline.**

which do not save on resources by speculation. (ii) *Selection Stage:* From the pruned set, GS picks the task with the lowest t_{new} while RAS picks the task with the highest resource savings (lines $13-15$).

3.1.2 Error-bound Jobs

Though error-bound jobs require a different form of prioritization than deadline-bound jobs, the speculative core of the GS and RAS algorithms are again quite natural. Specifically, the goal of error-bound jobs is to minimize the makespan of the tasks needed to achieve the error limit. Thus, instead of SJF, Longest Job First (LJF) is the natural prioritization of tasks. In particular, LJF minimizes the makespan among the class of non-preemptive policies in many settings [11, 12]. This again represents a "greedy" prioritization for this setting.

Despite the above change to the prioritization of which task to schedule, the form of GS and RAS remain the same as in the case of deadline-bound jobs. In particular, speculative copies are evaluated in the same manner, e.g., RAS's criterion is still to pick the task whose speculation leads to the highest resource savings. Pseudocode 2 presents the details. The pruning stage (lines $5-11$) will remove from consideration those tasks that are not the earliest to contribute to the desired error bound. The

```
1:  procedure ERROR(⟨Task⟩ T, float ϵ, bool OC)
                          ▷ OC = 1 → use RAS; 0 → use GS
                                  ▷ Error ϵ is in #tasks
2:      for each Task t in T do
            t.duration = min(t.t_rem, t.t_new)
3:          if OC then
4:              if t.running then
                    t.saving = t.c × t.t_rem − (t.c+1) × t_new
                                          ▷ PRUNING STAGE
        SortAscending(T, "duration")
        ⟨Task⟩Γ ← ϕ
5:      for each Task t in T[0 : T.count() (1 − ϵ)] do
                                          ▷ Earliest tasks
6:          if OC then
7:              if t.saving > 0 then Γ.add(t)
8:          else
9:              if t.running then
10:                 if t.t_new < t.t_rem then Γ.add(t)
11:             else Γ.add(t)
                                      ▷ SELECTION STAGE
12:     if Γ ≠ null then
13:         if OC then SortDescending(Γ, "saving")
14:         else SortDescending(Γ, t_rem)
            return Γ.first()
```

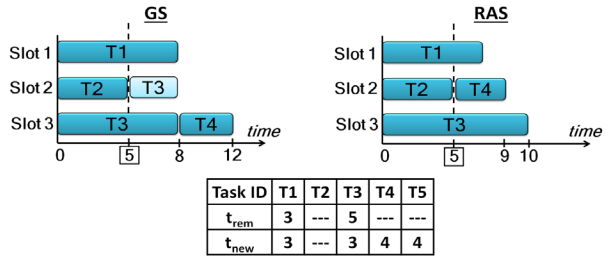
Pseudocode 2: **GS and RAS speculation algorithms for error-bound jobs (error-bound of ϵ). T is the set of unfinished tasks with the following fields per task: t_{rem}, t_{new}, and a boolean "running" to denote if a copy of it is currently executing. The t_{rem} of the task is the minimum of all its running copies. RAS is used when OC is set. At default, both algorithms schedule the task with the highest t_{rem}.**

list of earliest tasks is based on the effective duration of every task, i.e., the minimum of t_{rem} and t_{new}. During selection (lines $12-14$), GS picks the task with the highest t_{rem} while RAS picks the task with the highest saving.

Figure 2 presents an illustration of GS and RAS for an error-bound job with 6 tasks and 3 compute slots. The t_{rem} and t_{new} values are at 5 time units. GS decides to launch a copy of T3 as it has the highest t_{rem}. RAS conservatively avoids doing so. Consequently, when the error limit is high (say, 40%) GS is quicker, but RAS is better when the limit decreases (to, say, 20%).

3.2 Contrasting GS and RAS

To this point, we have seen that GS and RAS are two natural approaches for integrating speculation into a cluster scheduler for approximation jobs. However, the examples we have considered highlight that neither of GS or RAS is uniformly better. In order to develop a better understanding of these two algorithms, as well as other possible alternatives, we have developed a simple analytic model for speculation in approximation jobs. The model assumes wave-based scheduling and constant



Task ID	T1	T2	T3	T4	T5
t_{rem}	3	---	5	---	---
t_{new}	3	---	3	4	4

Figure 2: **GS and RAS for error-bound job with 6 tasks. The t_{rem} and t_{new} values are when T2 finishes. The example illustrates error limit of 40% (3 tasks) and 20% (4 tasks).**

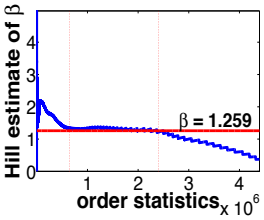


Figure 3: **Hill plot of Facebook task durations.**

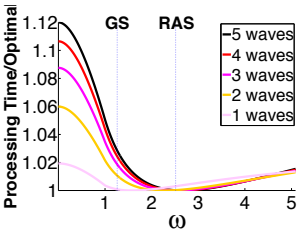


Figure 4: **Near-optimality of GS & RAS under Pareto task durations ($\beta = 1.259$).**

wave-width for a job (see §A for details along with formal results). For readability, here we present only the three major guidelines from our analysis. Most importantly, these guidelines highlight that different speculation policies are required during the early waves of a job than during the final wave.

Guideline 1 *During the early waves of a job, speculation is only valuable if task durations are extremely heavy tailed, e.g., Pareto with infinite variance (i.e., with shape parameter $\beta < 2$). In this case, it is optimal to speculate conservatively, using ≤ 2 copies of a task.*

This guideline is relevant because task durations are indeed heavy-tailed for the Facebook and Bing traces (see the Hill plot in Figure 3), which suggests that task durations have a Pareto tail (i.e., $P(\tau > x) = \theta(x^{-\beta})$) with shape parameter $\beta = 1.259$.[3] While both GS and RAS speculate during early waves, RAS is more conservative than GS and thus outperforms it during early waves.

Guideline 2 *During the final wave of a job, speculate aggressively to fully utilize the allotted capacity.*

[3] A Hill plot provides a more robust estimation of Pareto distributions than the, more commonly used, regression on a log-log plot [21]. To interpret the plot, a flat region corresponds to an estimate of β. The fact that the curve in Figure 3 is flat over a large range of order statistics (on the x-axis), but not all order statistics, indicates that the distribution of task sizes is not exactly Pareto distribution in its body, but is well-approximated by a Pareto (power-law) tail.

This guideline says that, even if all tasks are currently scheduled, if a slot becomes available it should be filled with a speculative copy. While both GS and RAS do this to some extent, GS speculates more aggressively than RAS and thus, outperforms RAS during the final wave.

The previous two guidelines highlight a tradeoff between RAS and GS, which we formalize next.

Guideline 3 *For jobs that require more than two waves RAS is near-optimal, while for jobs that require fewer than two waves GS is near-optimal.*

To make this point more salient, consider the general class of speculative replication policies that waits until a task has run ω time before starting a speculative copy. We study this broad class in §A, and GS and RAS correspond to particular rules for how to choose ω. To see this, we can define $t_{\text{new}} = E[\tau]$ and $t_{\text{rem}} = E[\tau - \omega | \tau > \omega]$, where τ is a random task size. Then, under GS, ω is the time when $E[\tau] = E[\tau - \omega | \tau > \omega]$, and, under RAS, ω is the time when $2E[\tau] = E[\tau - \omega | \tau > \omega]$.

Figure 4 contrasts the performance of all the replication policies in this more general class. Specifically, it shows the ratio of the response time of the replication policy with parameter ω normalized to the optimal response time. It illustrates this ratio for jobs of differing numbers of waves, and for $\omega \in [0, 5]$. To highlight GS and RAS, they are shown via vertical lines. The response times shown in the figure are computed using the model and analysis described in §A. The main conclusion from this figure is, as described in the guideline above, that neither GS or RAS is universally optimal, but each is near-optimal for jobs with a certain number of waves: RAS for jobs with large numbers of waves and GS for jobs with small numbers of waves.

4 GRASS Speculation Algorithm

In this section, we build our speculation algorithm called GRASS.[4] Our theoretical analysis summarized in §3.2 motivates a design that uses RAS during the early waves of jobs and GS during the final two waves. A simple *strawman* solution to achieve this would be as follows. For deadline-bound jobs, switch from RAS to GS when the time to the deadline is sufficient for at most two waves of tasks. Similarly, for error-bound jobs, switch when the number of (unique) scheduled tasks needed to satisfy the error-bound makes up two waves.

Identifying the final two waves of tasks is difficult in practice. Tasks are not scheduled at explicit wave boundaries but rather as and when slots open up. In addition, the wave-width of jobs does not stay constant but varies considerably depending on cluster utilization. Finally, task durations are varied and hard to estimate.

In light of these difficulties, we interpret the guideline as follows: RAS is better when the deadline is loose or the error limit is low, while otherwise GS performs better. This mimics the intuition from the examples in §3.1. Therefore, GRASS seeks to switch from RAS to GS as it gets *close to the job's approximation bound*.

The complexities in these systems mean that precise estimates of the optimal switching point cannot be obtained from our model. Instead, we adopt an indirect learning based approach where inferences are made based on executions of previous jobs (with similar number of tasks) and cluster characteristics (utilization and estimation accuracy). We compare our learning approach to the strawman described above in §6.3.

4.1 Learning the Switching Point

An ideal approach would accumulate enough samples of job performance (accuracy or completion time) based on switching to GS at different points. For deadline-bound jobs, this is decided by the remaining time to the deadline. For error-bound jobs, this is decided by the number of tasks to complete towards meeting the error. To speed up our sample collection, instead of accumulating samples of switching to GS, we simply generate samples of job performance using GS or RAS *throughout* the job (described shortly in §4.2).

An incoming job starts with RAS and periodically compares samples of jobs smaller than its size during its execution to check if it is better to switch to GS. It checks by using its remaining work at any point (measured in time remaining or tasks to complete). It steps through all possible points in its remaining work at which it could switch and estimates the optimal point using job samples of appropriate sizes. It continues with RAS until the optimal switching point turns out to be *at present*. The above calculation for the optimal switching point is performed periodically during the job's execution.

For example, when a deadline-bound job has 6s of its deadline remaining, GRASS compares the potential accuracy obtained if it were to switch at each point in its future (at 1s granularity). The accuracy if it were to switch after, say, 2s is the sum of accuracies of jobs with deadlines of 2s that used only RAS and those with 4s that used only GS. Switching happens if among all such points, the best accuracy is obtained by switching now.

The size of the job alone is insufficient to calculate the optimal switching point. Even jobs of comparable size might have different number of waves depending on the number of available slots. Therefore, we augment our samples of job performance with the number of waves, simply approximated using current *cluster utilization*.

[4] GRASS is a concatenation of GS and RAS

Finally, *estimation accuracy* of t_{rem} and t_{new} also decides the optimal switching point. RAS's cautious approach of considering the opportunity cost of speculating a task is valuable when task estimates are erroneous. In fact, at low estimation accuracies (along with certain values of utilization and deadline/error-bound), it is better to not switch to GS at all and employ RAS all along. §6.3.2 analyzes the impact of these three factors.

Therefore, GRASS obtains samples of job performance with both GS and RAS across values of deadline/error-bound, estimation accuracy of t_{rem} and t_{new}, and cluster utilization. It uses these three factors collectively to decide when (and if) to switch from RAS to GS. We next describe how the samples are collected.

4.2 Generating Samples

As described above, GRASS compares samples of job performance that use only GS or RAS throughout, to decide when to switch. These samples have to be updated continuously to stay abreast with dynamic changes in clusters. To continuously generate such samples, we introduce a *perturbation* in GRASS's switching decision. With a small probability ξ, GRASS decides to *not* switch and instead picks one of GS or RAS for the entire duration of the job (both GS and RAS are equally probable). Such perturbation helps us obtain comparable samples.

The crucial trade-off in setting ξ is in balancing the benefit of obtaining such comparable samples with the performance loss incurred by the job due to not making the right switching decision. Theoretical analyses of such multi-armed bandit problems in prior work defines an optimal value of ξ by making stochastic assumptions about the distribution of the costs and the associated rewards [22]. Our setup, however, does not yield itself to such assumptions as the underlying distribution can be arbitrary. Another class of techniques that we considered modified ξ with time [23]. Over time, the value of ξ is gradually reduced using a damping function, thus indicating higher confidence in the learned value. We decided against such damping of ξ because clusters constantly evolve with new software and hardware modules, leading to newer interactions between them.

Therefore, we pick a constant value of ξ using empirical analysis. A job is marked for generating performance samples with a probability of ξ, and we pick GS or RAS with equal probability. In practice, we bucket jobs by their number of tasks and compare only within jobs of the same bucket.

5 Implementation

We implement GRASS on top of two data-analytics frameworks, Hadoop (version 0.20.2) [14] and Spark (version 0.7.3) [15], representing batch jobs and interactive jobs, respectively. Hadoop jobs read data from HDFS while Spark jobs read from in-memory RDDs. Consequently, Spark tasks finished quicker than Hadoop tasks, even with the same input size. Note that while Hadoop and Spark use LATE[2] currently, we also implement Mantri[1] to use as a second baseline.

Implementing GRASS required two changes: task executors and job scheduler. Task executors were augmented to periodically report progress. We piggyback on existing update mechanisms of tasks that conveyed only their start and finish. Progress reports were configured to be sent every 5% of data read/written. The job scheduler collects these reports, maintains samples of completed tasks and jobs, and decides the switching point.

5.1 Task Estimators

GRASS uses two estimates for tasks: remaining duration of a running task (t_{rem}) and duration of a new copy (t_{new}).
Estimating t_{rem}: Tasks periodically update the scheduler with *progress reports* containing the fraction of input read and output written. Since tasks of analytics jobs are IO-intensive, we extrapolate the remaining duration of the task based on the time elapsed thus far.
Estimating t_{new}: We estimate the duration of a new task by sampling from durations of completed tasks (normalized to input and output sizes). The t_{new} values of all tasks are updated whenever a task completes.
Accuracy of estimation: While the above techniques are simple, the downside is the error in estimation. Our estimates of t_{rem} and t_{new} achieve moderate accuracies of 72% and 76%, respectively, on average. When a task completes, we update the accuracy using the estimated and actual durations. GRASS uses the accuracy of estimation to appropriately switch from RAS to GS.

5.2 DAG of Tasks

Jobs are typically composed as a DAG of tasks with *input* tasks (*e.g.,* map or extract) reading data from the underlying storage and *intermediate* tasks (*e.g.,* reduce or join) aggregating their outputs. Even in DAGs of tasks, the accuracy of the result is decided by the fraction of completed input tasks. This makes GRASS's functioning straightforward in error-bound jobs—complete as many input tasks as required to meet the error-bound and all intermediate tasks further in the DAG.

For deadline-bound jobs, we use a widely occurring property that intermediate tasks perform similar functions across jobs. Further, they have relatively fewer stragglers. Thus, we estimate the time taken for intermediate tasks by comparing jobs of similar sizes and then subtract it to obtain the deadline for the input tasks.

Input tasks of a job, typically, read equal amounts of data. Thus, the fraction of tasks completed represents fraction of data processed too, thus making it a good indicator of the result's accuracy.

6 Evaluation

We evaluate GRASS on a 200 node EC2 cluster. Our focus is on quantifying the performance improvements compared to current designs, i.e., LATE [2] and Mantri [1], and on understanding how close to the optimal performance GRASS comes. Our main results can be summarized as follows.

1. GRASS increases accuracy of deadline-bound jobs by 47% and speeds up error-bound jobs by 38%. Even non-approximation jobs (i.e., error-bound of zero) speed up by 34%. Further, GRASS nearly matches the optimal performance. (§6.2)

2. GRASS's learning based approach for determining when to switch from RAS to GS is over 30% better than simple strawman techniques. Further, the use of all three factors discussed in §4.1 is crucial for inferring the optimal switching point. (§6.3)

6.1 Methodology

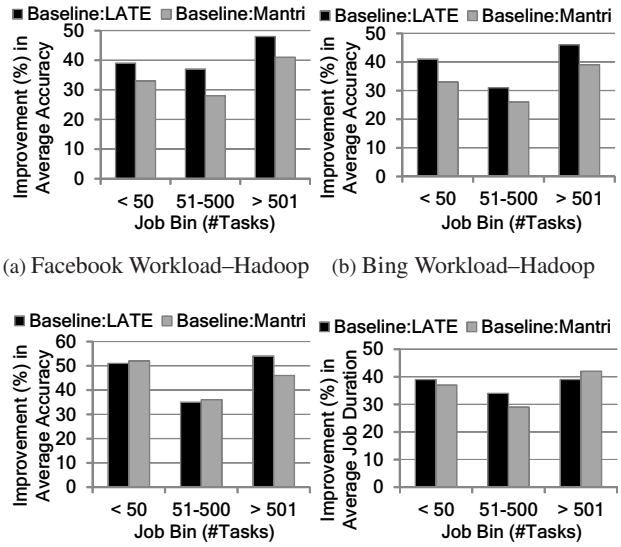
Workload: Our evaluation is based on traces from Facebook's production Hadoop [14] cluster and Microsoft Bing's production Dryad [24] cluster. The traces capture over half a million jobs running across many months (Table 1). The clusters run a mix of interactive and production jobs whose performance have significant impact on productivity and revenue. The jobs had diverse resource requirements of CPU, memory and IO. To create our experimental workload, we retain the inter-arrival times, input files and number of tasks of jobs. The jobs were, however, not approximation queries and required all their tasks to complete. Hence, we convert the jobs to mimic deadline- and error-bound jobs as follows.

For experiments on error-bound jobs, we pick the error tolerance of the job randomly between 5% and 30%. This is consistent with the experimental setup in recently reported research [4, 25]. Prior work also recommends setting deadlines by calibrating task durations [4, 9]. For the purpose of calibration, we obtain the ideal duration of a job in the trace by substituting the duration of each of its task by the median task duration in the job, again, as per recent work on straggler mitigation [3]. We set the deadline to be an additional factor (randomly between 2% to 20%) on top of this ideal duration.

Job Bins: We bin jobs by their number of tasks. We use three distinctions "small" (< 50 tasks), "medium" ($51 - 500$ tasks), and "large" (> 500 tasks).

	Facebook	Microsoft Bing
Dates	Oct 2012	May-Dec 2011
Framework	Hadoop	Dryad
Script	Hive [26]	Scope [27]
Jobs	575K	500K
Cluster Size	3,500	Thousands
Straggler–mitigation	LATE [2]	Mantri [1]

Table 1: **Details of Facebook and Bing traces.**



(a) Facebook Workload–Hadoop (b) Bing Workload–Hadoop

(c) Facebook Workload–Spark (d) Bing Workload–Spark

Figure 5: **Accuracy Improvement in deadline-bound jobs with LATE [2] and Mantri [1] as baselines.**

EC2 Deployment: We deploy our Hadoop and Spark prototypes on a 200-node EC2 cluster and evaluate them using the workloads described above. Each experiment is repeated five times and we pick the median. We measure improvement in the average accuracy for deadline-bound jobs and average duration for error-bound jobs.

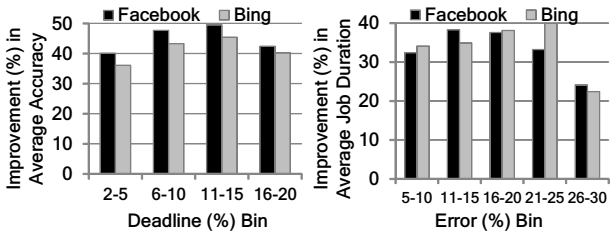
We also use a trace-driven simulator to evaluate at larger scales and over longer durations. The simulator replays all the task properties including their straggling.
Baseline: We contrast GRASS with two state-of-the-art speculation algorithms—LATE [2] and Mantri [1].

6.2 Improvements from GRASS

We contrast GRASS's performance with that of LATE [2], Mantri [1], and the optimal scheduler.

6.2.1 Deadline-bound jobs

GRASS improves the accuracy of deadline-bound jobs by 34% to 40% in the Hadoop prototype. Gains in both the Facebook and Bing workloads are similar. Figure 5a and 5b split the gains by job size. The gains compared



(a) Deadline Bins (b) Error Bins

Figure 6: GRASS**'s overall gains (compared to LATE) binned by the deadline and error bound. Deadlines are binned by the factor over ideal job duration (see §6.1)**

to LATE as baseline are consistently higher than Mantri. Also, the gains in large jobs are pronounced compared to small and medium sized jobs because their many waves of tasks provides plenty of potential for GRASS.

The Spark prototype improves accuracy by 43% to 47%. The gains are higher because Spark's task sizes are much smaller than Hadoop's due to in-memory inputs. This makes the effect of stragglers more distinct. Again, large jobs gain the most, improving by over 50% (Figure 5c and 5d). Large multi-waved jobs improving more is encouraging because smaller task sizes in future [18] will ensure that multi-waved executions will be the norm. Unlike the Hadoop case, gains compared to both LATE and Mantri are similar because both have only limited effect when the impact of stragglers is high.

Figure 6a dices the improvements by the deadline (specifically, the additional factor over the ideal job duration (see §6.1)). Note that gains are nearly uniform across deadline values. This indicates that GRASS can not only cope with stringent deadlines but be valuable even when the deadline is lenient.

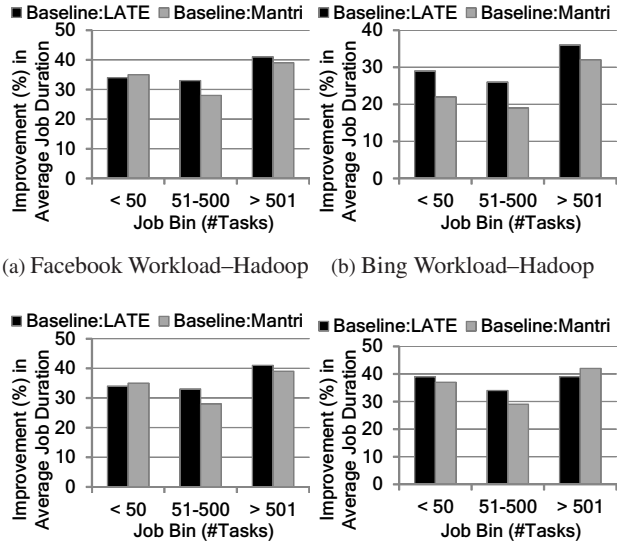
Gains with simulations are consistent with deployment, indicating not only that GRASS's gains hold over longer durations but also the simulator's robustness.

6.2.2 Error-bound jobs

Similar to deadline-bound jobs, improvements with the Spark prototype (33% to 37%) are higher compared to the Hadoop prototype (24% to 30%). This shows that GRASS works well not only with established frameworks like Hadoop but also upcoming ones like Spark.

Note that the gains are indistinguishable among different job bins (Figures 7a and 7b) in the Spark prototype; large jobs gain a touch more in the Hadoop prototype (Figures 7c and 7d). Again, our simulation results are consisten with deployment, and so are omitted.

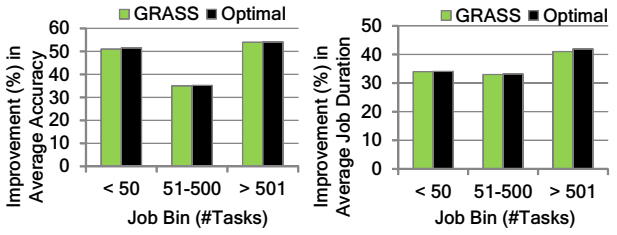
As Figure 6b shows, GRASS's gains persist at both tight as well as moderate error bounds. At high error bounds, there is smaller scope for GRASS beyond LATE.



(a) Facebook Workload–Hadoop (b) Bing Workload–Hadoop

(c) Facebook Workload–Spark (d) Bing Workload–Spark

Figure 7: **Speedup in error-bound jobs with LATE [2] and Mantri [1] as baselines.**



(a) Deadline-bound Jobs (b) Error-bound Jobs

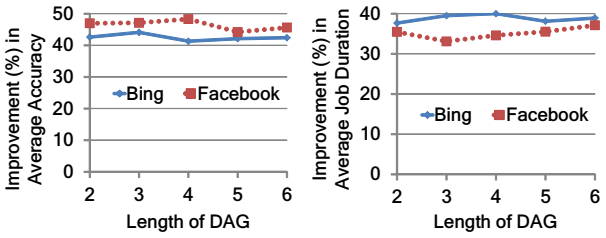
Figure 8: GRASS**'s gains matches the optimal scheduler.**

The gains at tight error bounds is noteworthy because these jobs are closer to *exact* jobs that require all (or most of) their tasks to complete. In fact, exact jobs speed up by 34%, thus making GRASS valuable even in clusters that are yet to deploy approximation analytics.

6.2.3 Optimality of GRASS

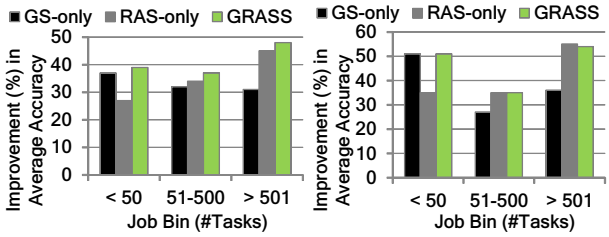
While the results above show the speed up GRASS provides, the question remains as to whether further improvements are possible. To understand the room available for improvement beyond GRASS, we compare its performance with an optimal scheduler that knows task durations and slot availabilities in advance.

Figure 8 shows the results for the Facebook workload with Spark. It highlights that GRASS's performance matches the optimal for both deadline as well as error-bound jobs. Thus, GRASS is an efficient near-optimal solution for the NP-hard problem of scheduling tasks for approximation jobs with speculative copies.



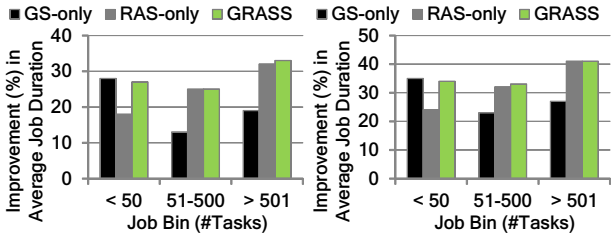
(a) Deadline-bound Jobs. (b) Error-bound Jobs.

Figure 9: GRASS's gains holds across job DAG sizes.



(a) Hadoop (b) Spark

Figure 10: GRASS's switching is 25% better than using GS or RAS all through for deadline-bound jobs. We use the Facebook workload and LATE as baseline.



(a) Facebook Workload–Hadoop (b) Facebook Workload–Spark

Figure 11: GRASS's switching is 20% better than using GS or RAS all through for error-bound jobs. We use the Facebook workload and LATE as baseline.

6.2.4 DAG of tasks

To complete the evaluation of GRASS we investigate how performance gains depend on the length of the job's DAG. Intuitively, as long as our estimation of intermediate phases is accurate, GRASS's handling of the input phase should remain unchanged, and Figure 9 confirms this for both deadline and error-bound jobs. Gains from GRASS remain stable with the length of the DAG.

6.3 Evaluating GRASS's Design Decisions

To understand the impact of the design decisions made in GRASS, we focus on three questions. First, how important is it that GRASS switches from RAS to GS? Second, how important is it that this switching is learned adaptively rather than fixed statically? Third, how sensitive is GRASS to the perturbation factor ξ? In the interest of space, we present results on these topics for only the Facebook workload using LATE as a baseline; results for the Bing workload with Mantri are similar.

6.3.1 The value of switching

To understand the importance of switching between RAS and GS we compare GRASS's performance with using only GS and RAS all through the job. Figure 10 performs the comparison for deadline-bound jobs. GRASS's improvements, both on average as well as in individual job bins, are strictly better than GS and RAS. This shows that if using only one of them is the best choice, GRASS automatically avoids switching. Further, GRASS's overall improvement in accuracy is over 20% better than the best of GS or RAS, demonstrating the value of switching as the job nears its deadline. The above trends are consistent with error-bound jobs as well (Figure 11), though GRASS's benefit is slightly lower.

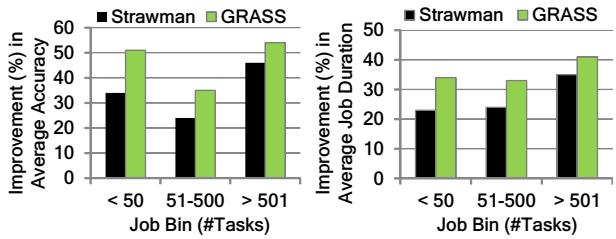
The contrast of GS and RAS is also interesting. GS outperforms RAS for small jobs but loses out as job sizes increase. The significant degradation in performance in the unfavorable job bin (medium and large jobs for GS, versus small jobs for RAS) illustrates the pitfalls of statically picking the speculation algorithm.

6.3.2 The value of learning

Given the benefit of switching, the question becomes when this switching should occur. GRASS does this adaptively based on three factors: deadline/error-bound, cluster utilization and estimation accuracy of t_{rem} and t_{new}. Now, we illustrate the benefit of this approach compared to simpler options, i.e., choosing the switching point statically or based on a subset of these three factors. Note that we have already seen that these three factors are enough to be near optimal (Figure 8).

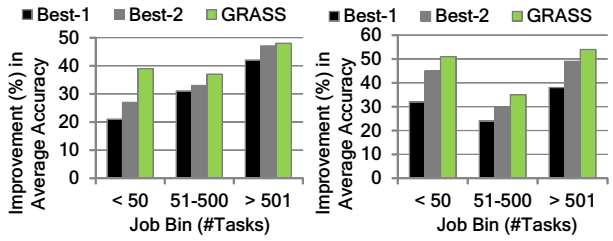
Static switching: First, when considering a static design, a natural "strawman" based on our theoretical analysis is to estimate the point when there are two remaining waves as follows. For deadline-bound jobs, it is the point when the time to the deadline is sufficient for at most two waves of tasks. For error-bound jobs, it is when the number of (unique) scheduled tasks sufficient to satisfy the error-bound make up two waves. The strawman uses the current wave-width of the job and assumes task durations to be median of completed tasks.

Figure 12 compares GRASS with the above strawman. Gains with the strawman are 66% and 73% of the gains with GRASS for deadline-bound and error-bound jobs,



(a) Deadline-bound Jobs (b) Error-bound Jobs

Figure 12: **Comparing GRASS's learning based switching approach to a strawman that approximates two waves of tasks. GRASS is $30\% - 40\%$ better than the strawman.**



(a) Hadoop (b) Spark

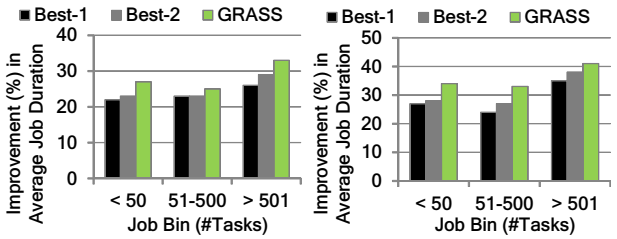
Figure 13: **Using all three factors for deadline-bound jobs compared to only one or two is $18\% - 30\%$ better.**

respectively. Small and medium jobs lag the most as wrong estimation of switching point affects a large fraction of their tasks. Thus, the benefit of adaptively determining the switching point is significant.

Adaptive switching: Next, we study the impact of the three factors used to adaptively learn the switching threshold. To do this, Figures 13 and 14 compares the designs using the best one or two factors with GRASS.

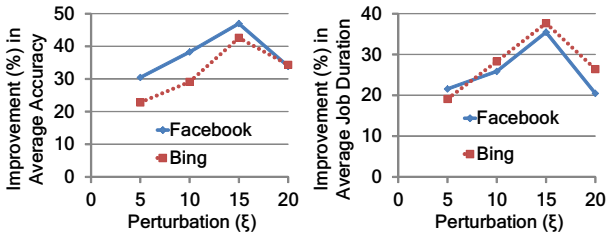
When only one factor can be used to switch, picking the deadline/error-bound provides the best results. This is intuitive given the importance of the approximation bound to the ordering of tasks. When two factors are used, in addition to the deadline/error-bound, cluster utilization matters more for the Hadoop prototype while estimation accuracy is important for the Spark prototype. Tasks of Hadoop jobs are longer and more sensitive to slot allocations, which is dictated by the utilization. While the smaller Spark tasks are more fungible, this also makes them sensitive to estimation errors.

Using only one factor is significantly worse than using all three factors. The performance picks up with deadline-bound jobs when two factors are used, but error-bound jobs' gains continue to lag until all three are used. Thus, in the absence of a detailed model for job executions, the three factors act as good predictors.



(a) Hadoop (b) Spark

Figure 14: **Using all three factors for error-bound jobs compared to one or two factors is $15\% - 25\%$ better.**



(a) Deadline-bound Jobs (b) Error-bound Jobs

Figure 15: **Sensitivity of GRASS's performance to the perturbation factor ξ. Using $\xi = 15\%$ is empirically best.**

6.3.3 Sensitivity to Perturbation

The final aspect of GRASS that we evaluate is the perturbation factor, ξ, which decides how often the scheduler does *not* switch during a job's execution (described in §4.2). This perturbation is crucial for GRASS's learning of the optimal switching point. All results shown previously set ξ to 15%, which was picked empirically.

Figure 15 highlights the sensitivity of GRASS to this choice. Low values of ξ hamper learning because of the lack of sufficient samples, while high values incur performance loss resulting from not switching from RAS to GS often enough. Our results show, that this exploration–exploitation tradeoff is optimized at $\xi = 15\%$, and that performance drops off sharply around this point. Deadline-bound jobs are more sensitive to poor choice of ξ than error-bound jobs. Using ξ of 15% is consistent with studies on multi-armed bandit problems [28], which is related to our learning problem.

7 Related Work

The problem of stragglers was identified in the original MapReduce paper [29]. Since then solutions have been proposed to mitigate them using speculative executions [2, 1, 24]. These solutions, however, are not for approximation jobs. These jobs require proritizing the right subset of tasks by carefully considering the opportunity cost of speculation. Further, our evaluations show

that GRASS speeds up even for exact jobs that require all their tasks to complete. Thus, it is a unified solution that cluster schedulers can deploy for both approximation as well as non-approximation computations.

Prioritizing tasks of a job is a classic scheduling problem with known heuristics [11, 12]. These heuristics assume accurate knowledge of task durations and hence do not require speculative copies to be scheduled dynamically. Estimating task durations accurately, however, is still an open challenge as acknowledged by many studies [3, 20]. This makes speculative copies crucial and we develop a theoretically backed solution to optimally prioritize tasks with speculative copies.

Modeling real world clusters has been a challenge faced by other schedulers too. Recently reported research has acknowledged the problem of estimating task durations [16], predicting stragglers [3, 20] as well as modeling multi-waved job executions [17]. Their solutions primarily involve sidestepping the problem by not predicting stragglers and upfront replicating the tasks [3], or approximating number of waves to file sizes [17]. Such sidestepping, however, is not an option for GRASS and hence we build tailored approximations.

Finally, replicating tasks in distributed systems have a long history [30, 31, 32] with extensive studies in prior work [33, 34, 35]. These studies assume replication *upfront* as opposed to *dynamic* replication in reaction to stragglers. The latter problem is both harder and unsolved. In this work, we take a stab at this problem that yields near-optimal results in our production workloads.

8 Concluding Remarks and Future Work

This paper explores speculative task scheduling in the context of approximation jobs. From the analysis of a generic analytic model, we develop a speculation algorithm, GRASS, that uses opportunity cost to determine when to speculate early in the job and then switches to more aggressive speculation as the job nears its approximation bound. Prototypes on Hadoop and Spark, deployed on a 200 node EC2 cluster shows that GRASS improves accuracy fo deadline-bound jobs by 47% and speeds up error-bound jobs by 38%, in production workloads from Facebook and Bing. Further, the evaluation highlights that GRASS is a unified speculation solution for both approximation and exact computations, since it also provides a 34% speed up for exact jobs.

A topic that requires further work is scheduling speculative copies for stragglers *across* jobs. While GRASS intelligently picks between scheduling a speculative copy for a running task versus scheduling a new task of a job, it does so within the slots allocated to the job (typically, based on fair allocations). The next step to GRASS is to weigh the impact of speculating a running task with scheduling a new task of *any* job. Answering this question will not only involve comparing across jobs but also revisit existing fairness based schedulers.

A Modeling and Analyzing Speculation

In this section we introduce the model and analysis that led to the guidelines described in §3.2. The model focuses on one job that has T tasks[5] and S slots out of a total capacity normalized to 1. Let the initial job size be x and the remaining amount of work in the job at time t be $x(t)$. We use $W = T/S$ to denote the (fractional) number of waves necessary to complete the job, and throughout we assume $W \geq 1$.

We focus our analysis on the rate at which work is completed, which we denote by $\mu(t; x, S, T)$ or $\mu(t)$ for short. Note that by focusing on the service rate we are ignoring ordering of the tasks and focusing primarily on the impact of speculation.

In our analysis we begin with proactive speculation, and then move to reactive speculation. This progression is natural since the analysis of proactive speculation serves as a stepping stone to the design of reactive speculation policies. Further, in the case of proactive speculation we can precisely specify the optimal policy, whereas in the case of reactive speculation, we must resort to numerical optimization.

A.1 Proactive speculation

We start by considering a general class of proactive policies that launch $k(x(t))$ speculative copies of tasks when the job has remaining size $x(t)$. We propose the following *approximate model* for $\mu(t)$ in this case.

$$\left[\left(\frac{x(t)}{x}\right)\left(\frac{T}{S}\right)k(x(t))\right]^{1}_{\frac{k(x(t))}{S}} \cdot \left(\frac{E[\tau]}{k(x(t))E\left[\min(\tau_1, \ldots, \tau_{k(x(t))}\right]}\right) \quad (1)$$

where τ is a random task size. Note that we assume task sizes are i.i.d.

To understand this approximate model, note that the first term approximates the completion rate of work and the second term approximates the "blow up factor," i.e., the ratio of the expected work completed without duplications to the amount of work done with duplications. To understand the first term, note that $(x(t)/x)T$ is the fractional number of tasks that remain to be completed at time t, and thus there are $(x(t)/x)Tk(x(t))$ tasks available to schedule at time t including speculative copies. Recalling that the capacity of a slot is $1/S$, that the maximum capacity that can be allocated is 1, and that the minimum number of slots is the number of copies $k(x(t))$,

[5] For approximation jobs T should be interpreted as the number of tasks that are completed before the deadline or error limit is reached.

we obtain the first term in (1). The second term computes the "blow up factor," which is the the expected amount of work done per task without speculation ($E[\tau]$) divide by the expected work done per task with speculation ($k(x(t))E[\min(\tau_1, \tau_2, \ldots, \tau_{k(x(t))})]$, since $k(x(t))$ copies are created and they are stopped when the first copy completes. Perhaps the most important aspect of this approximation is the fact that task sizes are i.i.d. in this manner, and this is what leads both to stragglers and to the benefits of replication. While this is certainly simplistic, the value of the model is highlights be the usefulness of the guidelines that follow from our analysis.

Given the model in (1), the question is: What proactive speculation policy minimizes job duration? As discussed in §3.2, the distribution of task sizes shows considerable evidence of a Pareto-tail, and so we focus our analysis on this setting. The following theorem follows from first deriving the response time of a job given the model for $\mu(t)$ in (1), and then deriving the $k(x(t))$ that minimizes the response time. Each of these steps requires significant, but not difficult, analysis, which we omit due to space.

Theorem 1 *When task sizes are Pareto(x_m,β), the proactive speculation policy that minimizes the completion time of the job is*

$$k(x(t)) = \begin{cases} \sigma, & \frac{x(t)}{x} T\sigma \geq S \\ S/(\frac{x(t)}{x}T), & S > \frac{x(t)}{x} T\sigma \text{ and } \frac{x(t)}{x}T \geq 1; \\ S, & 1 > \frac{x(t)}{x}T. \end{cases} \quad (2)$$

where $\sigma = \max(2/\beta, 1)$.

This theorem leads to Guidelines 1 and 2. Specifically, the first line corresponds to the "early waves" and the second and third lines correspond to the "last wave". During the "early waves" the optimal policy may or may not speculate, depending on the task size distribution – speculation happens only when $\beta < 2$, which is when task sizes have infinite variance. In contrast, during the "last wave", regardless of the task size distribution, the optimal policy speculates to ensure all slots are used.

A.2 Reactive speculation

We now turn to reactive speculation policies, which launches copies of a task only after it has completed ω work. Both GS and RAS are examples of such policies and can be translated into choices for ω as shown in §3.2.

Our analysis of proactive policies provides important insight into the design of reactive policies. In particular, during early waves the the optimal proactive policy runs at most two copies of each task, and so we limit our reactive policies to this level of speculation. Additionally, the previous analysis highlights that during the last wave the it is best to speculate aggressively in order to use up the full capacity, and thus it is best to speculated immediately without waiting ω time. This yields the following approximation for $\mu(t)$:

$$\begin{cases} \frac{E[\tau_1]}{E[\tau_1 | 0 \leq \tau_1 < \omega] \Pr(0 \leq \tau_1 < \omega) + (2E[Z - \omega | \tau_1 \geq \omega] + \omega) \Pr(\tau_1 > \omega)}, \\ \quad \text{when } \frac{x(t)}{x} T(\Pr(0 \leq \tau_1 < \omega) + 2\Pr(\tau_1 \geq \omega)) \geq S. \\ \\ \text{optimal proactive speculation (from (1))}, \\ \quad \text{when } \frac{x(t)}{x} T(\Pr(0 \leq \tau_1 < \omega) + 2\Pr(\tau_1 \geq \omega)) < S. \end{cases} \quad (3)$$

τ_1, τ_2 are random task sizes and $Z = \min(\tau_1, \tau_2 + \omega)$.

Again, the first line in (3) approximates the service rate during the early waves of the job, while the second line approximates the service rate during the final wave of the job. To understand the first line, note that during early waves there are enough tasks to use capacity 1 (in expectation) as long as $\frac{x(t)}{x} T(\Pr(0 \leq \tau < \omega) + 2\Pr(T \geq \omega)) \geq S$. Thus, all that remains is the "blow up factor." As before, the numerator is the expected amount of work per task without speculation ($E[\tau]$) and the denominator is the expected amount of work per task with reactive speculation. This is $E[\tau | \tau < \omega]$ if the initial copy finishes before ω, and $2E[Z - \omega | \tau > \omega] + \omega$ if the initial copy takes longer than ω.

Within this model, our design problem can now be reduced to finding ω *that minimizes the response time of the job*. The complicated form of (3) makes it difficult to understand the optimal ω analytically, and thus we use numerical calculations. Figure 4 presents a numerical optimization by comparing GS and RAS to other reactive policies. It leads go Guideline 3, which highlights that GS is near optimal if the number of waves in the job is < 2, while RAS is near-optimal if the number of waves in the job is ≥ 2. Note that the results in Figure 4 are for Pareto task sizes with $\beta = 1.259$, but the finding is robust for $\beta \in (1, 2)$.

Acknowledgments

We thank our shepherd Nina Taft and the anonymous reviewers for their suggestions to improve this work. We also thank Rohan Gandhi for his feedback on our early drafts. This research was partially funded by research grant NSF CNS-1319820, NSF CISE Expeditions award CCF-1139158, the DARPA XData Award FA8750-12-2-0331, and gifts from Qualcomm, Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

References

[1] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, E. Harris, and B. Saha. Reining in the Outliers in Map-Reduce Clusters Using Mantri. In *USENIX OSDI*, 2010.

[2] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *USENIX OSDI*, 2008.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective Straggler Mitigation: Attack of the Clones. In *USENIX NSDI*, 2013.

[4] S.Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*. ACM, 2013.

[5] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *USENIX NSDI*, 2010.

[6] Interactive Big Data analysis using approximate answers, 2013. http://tinyurl.com/k5favda.

[7] J. Liu, K. Shih, W. Lin, R. Bettati, and J. Chung. Imprecise Computations. *Proceedings of the IEEE*, 1994.

[8] S. Lohr. *Sampling: design and analysis*. Thomson, 2009.

[9] J. Hellerstin, P. Haas, and H. Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[10] M. Garofalais and P. Gibbons. Approximate Query Processing: Taming the Terabytes. In *VLDB*, 2001.

[11] M. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2012.

[12] L. Kleinrock. *Queueing systems, volume II: computer applications*. John Wiley & Sons New York, 1976.

[13] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-real-time Environment. *Journal of the ACM (JACM)*, 1973.

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

[15] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX NSDI*, 2012.

[16] E. Bortnikov, A. Frank, E. Hillel, S. Rao. Predicting Execution Bottlenecks in Map-Reduce Clusters. In *USENIX HotCloud*, 2012.

[17] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *USENIX NSDI*, 2012.

[18] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The Case for Tiny Tasks in Compute Clusters. In *USENIX HotOS*, 2013.

[19] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A Study of Skew in MapReduce Applications. In *Open Cirrus Summit*, 2011.

[20] J. Dean. Achieving Rapid Response Times in Large Online Services. In *Berkeley AMPLab Cloud Seminar*, 2012.

[21] S. Resnick. *Heavy-tail phenomena: probabilistic and statistical modeling*. Springer, 2007.

[22] J. C. Gittins. Bandit Processes and Dynamic Allocation Indices. *Journal of the Royal Statistical Society. Series B (Methodological)*, 1979.

[23] I. Sonin. A Generalized Gittins Index for a Markov Chain and Its Recursive Calculation. *Statistics & Probability Letters*, 2008.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *ACM Eurosys*, 2007.

[25] W. Baek and T. Chilimbi. Green: a Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *ACM Sigplan Notices*, 2010.

[26] Hive. http://wiki.apache.org/hadoop/Hive.

[27] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[28] M. Tokic and G. Palm. Value-difference Based Exploration: Adaptive Control between Epsilon-greedy and Softmax. In *KI 2011: Advances in Artificial Intelligence*. Springer, 2011.

[29] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[30] A. Baratloo, M. Karaul, Z. Kedem, and P. Wycko. Charlotte: Metacomputing on the Web. In *9th Conference on Parallel and Distributed Computing Systems*, 1996.

[31] E. Korpela D. Anderson, J. Cobb. SETI@home: An Experiment in Public-Resource Computing. In *Comm. ACM*, 2002.

[32] M. Rinard and P. Diniz. Commutativity Analysis: a New Analysis Framework for Parallelizing Compilers. In *ACM PLDI*, 1996.

[33] D. Paranhos, W. Cirne, and F. Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In *Euro-Par*, 2003.

[34] G. Ghare and S. Leutenegger. Improving Speedup and Response Times by Replicating Parallel Programs on a SNOW. In *JSSPP*, 2004.

[35] W. Cirne, D. Paranhos, F. Brasileiro, L. Goes, and W. Voorsluys. On the Efficacy, Efficiency and Emergent Behavior of Task Replication in Large Distributed Systems. In *Parallel Computing*, 2007.

Bringing Gesture Recognition To All Devices

Bryce Kellogg[†], Vamsi Talla[†], and Shyamnath Gollakota
University of Washington
[†]Co-primary Student Authors

Abstract

Existing gesture-recognition systems consume significant power and computational resources that limit how they may be used in low-end devices. We introduce AllSee, the first gesture-recognition system that can operate on a range of computing devices including those with no batteries. AllSee consumes three to four orders of magnitude lower power than state-of-the-art systems and can enable always-on gesture recognition for smartphones and tablets. It extracts gesture information from existing wireless signals (e.g., TV transmissions), but does not incur the power and computational overheads of prior wireless approaches. We build AllSee prototypes that can recognize gestures on RFID tags and power-harvesting sensors. We also integrate our hardware with an off-the-shelf Nexus S phone and demonstrate gesture recognition in through-the-pocket scenarios. Our results show that AllSee achieves classification accuracies as high as 97% over a set of eight gestures.

1 Introduction

There is growing interest in using human gestures as a means of interaction with computing devices. The success of Xbox Kinect has led to the development of novel gesture-recognition approaches including those based on infrared [4], electric-field sensing [6], and more recently, wireless signals (e.g., Wi-Fi) [32].

Existing approaches, however, are limited in how they may be used for low-end devices. For example, the "air gesture" system on Samsung Galaxy S4 drains the battery when run continuously [2]. Similarly, wireless approaches [18, 32] consume significant power and computational resources that limit their applicability to plugged-in devices such as Wi-Fi routers.

This is due to two main reasons: First, existing approaches need power-hungry sensing hardware — always-on cameras drain the battery; wireless receivers use power-intensive analog components such as oscillators and high-speed ADCs. Second, they require rich signal processing capabilities — computing optical flows, FFTs, frequency-time Doppler profiles, etc., is computationally expensive; performing these operations while maintaining fast response times consumes power.

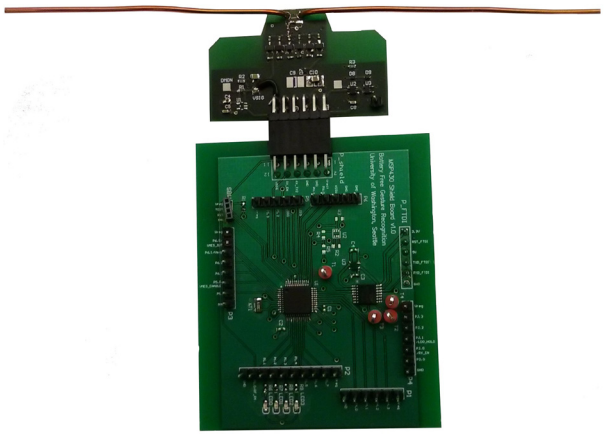


Figure 1: **AllSee Prototype.** It has two main components: a pluggable receiver that extracts the amplitude of wireless signals (e.g., TV and RFID transmissions), and our classification logic implemented on a microcontroller. It also comes with LEDs and a UART interface.

We introduce AllSee, a novel gesture-recognition system that can be used for computing devices, no matter how low-end and power-constrained they are. Such a capability can enable a number of interactive applications for Internet-of-things, where sensor deployments enable smart homes and cities [14, 24]; in fact, we show that AllSee can operate on battery-free devices that run off of harvested RF energy. AllSee can also be useful for small form-factor consumer electronics without the need for conventional input modalities like keypads. Finally, AllSee consumes three to four orders of magnitude lower power than existing systems; thus, it can also enable always-on gesture recognition on mobile devices such as

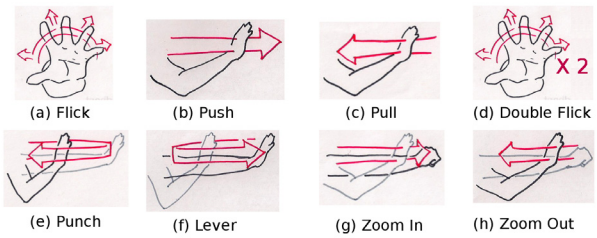


Figure 2: **Gesture Sketches.** AllSee can detect and classify these eight gestures using TV and RFID transmissions with average classification accuracies of 94.4% and 97% respectively.

smartphones and tablets.

AllSee's approach is to extract gesture information from ambient RF signals (e.g., TV transmissions) that already exist around us. AllSee can also work with signals from dedicated RF sources like RFID readers. AllSee eliminates the need for power-hungry wireless hardware components (e.g., oscillators) by using low-power analog operations to extract just the signal amplitude. To keep the computational complexity low, we avoid prior Doppler-based approaches that require computing FFTs and frequency-time Doppler profiles.[1] Instead, we introduce a novel design that extracts gesture information from the amplitude of the received signal.

At a high level, we use the insight that motion at a location farther from the receiver results in smaller wireless signal changes than from a close-by location. This is because the reflections from a farther location experience higher attenuation and hence have lower energy at the receiver. Now, consider the push and pull gestures shown in Fig. 2(b) and Fig. 2(c). As the user moves her arm *toward* the receiver, the wireless changes induced by the gesture increase with time, as the arm gets closer to the receiver. On the other hand, as the user moves her arm *away* from the receiver, the changes decrease with time. Thus, the receiver can distinguish between a pull and a push gesture even without access to the Doppler information. In §3, we explore this time-domain analysis further and extract deterministic rules to detect and classify the eight gestures in Fig. 2. Further in §3.3, we show that a subset of these rules can be encoded in the analog domain using passive components such as capacitors and resistors, further reducing our power consumption.

To demonstrate the feasibility of AllSee, we built multiple prototypes (one of which is shown in Fig. 1) that achieve gesture recognition on a range of devices. The first prototype is a modified RFID tag that extracts gesture information from signals of an RFID reader. The second prototype is a battery-free device that uses existing 725 MHz TV signals as a source of both power and gesture information. The third prototype encodes our gesture-detection rules using analog components, to further reduce the power consumption. We also integrate our prototypes with an off-the-shelf Nexus S phone and demonstrate gesture recognition in through-the-pocket scenarios; this enables the user to gesture at the phone in a pocket, to say change volume or mute the phone.

We evaluated our prototypes with up to five users using the gestures shown in Fig. 2. Our findings are as follows:

- AllSee classifies the eight gestures shown in Fig. 2 with an average accuracy of 97% and 94.4% on our RFID- and TV-based prototypes respectively; the accuracy is 92.5% with our phone prototype in through-the-pocket scenarios. This is promising, given that the accuracy for random guesses is 12.5%.
- AllSee achieves the above accuracies for distances of up to 2.5 feet from the device, while using 28.91 μW. This is in contrast to a state-of-the-art low-power gesture recognition system that consumes about 66 mW and has a range of upto 15 centimeters [31].
- The rate of false positive events—gesture detection in the absence of the target human—is 0.083 events per hour over a 24-hour period. AllSee achieves this by using a repetitive flick gesture to gain access to the system.
- AllSee's response time (the time between the gesture's end and its recognition) is less than 80 μs.
- Our analog gesture-encoding prototype that uses capacitors and resistors to detect a subset of gestures, consumes as little as 5.85 μW.

Contributions. We make the following contributions:

- We introduce the first gesture-recognition system that can run on battery-free devices. Our approach also enables always-on gesture recognition on mobile devices.
- We present computationally-light algorithms to extract gesture information from time-domain wireless signals.
- We show how to encode gestures in the analog domain using components such as capacitors and resistors.
- We build prototypes to demonstrate gesture recognition for multiple devices including RFID tags and power-harvesting battery-free devices. We integrate our prototypes with an off-the-shelf smartphone and use AllSee to classify gestures while the phone is in a pocket.

Our current implementation is limited to locations that have signals from either TV towers or RFID readers. We believe, however, that the techniques developed in this paper can be extended to leverage cellular and Wi-Fi transmissions, making AllSee more ubiquitous.

[1] As we show in §3.2.1, AllSee's receiver does not have phase information. Hence prior Doppler frequency solutions are in fact not applicable.

2 Related Work

Our work is related to prior art from both wireless and gesture-recognition systems.

(a) Wireless Systems: Prior work has shown the feasibility of using wireless signals for coarse motion detection (e.g., running [21] and walking forward and backward [18]). Further, our recent work on WiSee has shown how to extract gesture information from wireless signals [32]. These systems require power-hungry ultra-wideband transceivers [20, 33], interference-nulling hardware [18], or multiple antennas [18, 32]. Further, they require receivers with power-consuming analog components such as oscillators and high-speed ADCs and impose significant computational requirements such as 1024-point FFT and frequency-time Doppler profile computations [32]. We also note that these systems were implemented on USRPs, each of which consumes about 13.8 W [16]. In contrast, AllSee enables gesture recognition with orders of magnitude lower power.

AllSee is also related to work on low-power ultra-wideband radar sensors [1, 26, 27] that perform proximity detection. AllSee builds on this work, but goes beyond motion and velocity detection and designs the first wireless gesture-recognition system for power-constrained devices. We also show how to encode gestures using analog components with as little as 5.85 μW.

Finally, prior work [25, 29] has leveraged backscattered signals from RFID tags for activity recognition on a powered RFID reader. These systems, however, require quite a bit of computational processing and are designed to operate on powered RFID readers. In contrast, we enable gesture recognition on power-constrained devices.

(b) Gesture-Recognition Systems: Prior gesture-recognition systems can be primarily classified into vision-based, infrared-based, electric-field sensing, ultrasonic, and wearable approaches. Xbox Kinect [17], Leap Motion [9], PointGrab [12], and CrunchFish [5] use advances in cameras and computer vision to enable gesture recognition. Xbox Kinect uses the 3D sensor by PrimeSense which consumes 2.25 W [13], while PointGrab and CrunchFish run on mobile devices and consume as much power as the embedded camera.

Samsung Galaxy S4 introduced an "air gesture" feature that uses infrared cameras to enable gesture recognition. It is, however, not recommended to keep the gesture recognition system ON as it can drain the battery [2]. Further, it is known to be sensitive to lighting conditions [3] and does not work in through-the-pocket scenarios. GestIC [6], which we believe is the state-of-the-art system, uses electric-field sensing to enable gesture recognition using about 66 mW in the processing mode [31]. However, it requires the user's hand to be within 15 centimeters of the screen and also does not work in through-the-pocket scenarios. Further, it requires a relatively large surface area for sensing the electric fields. AllSee on the other hand achieves gesture recognition with three to four orders of magnitude lower power, works on devices with smaller form factors and in through-the-pocket scenarios.

Ultrasonic systems such as SoundWave [28] transmit ultrasound waves and analyze them for gesture recognition. These systems require active ultrasound transmissions and expensive operations such as 1024-point FFTs and Doppler profile computations. In contrast, AllSee leverages existing wireless signals (e.g., TV) and thus does not require active transmissions; this reduces the power consumption to the microwatts range. We note that, in principle, the time-domain analysis developed in this paper can be applied to ultrasonic systems to reduce their computational complexity.

Finally, prior work on inertial sensing and other on-body gesture recognition systems require instrumenting the human body with sensing devices [11, 22, 23]. In contrast, we focus on gesture recognition without requiring such instrumentation.

3 AllSee

AllSee is an ultra-low power wireless gesture-recognition sensor that consumes three to four orders of magnitude lower power than state-of-the-art systems. It uses ambient wireless signals (e.g., TV, cellular, and Wi-Fi) to extract gesture information. In this paper, we focus on demonstrating the feasibility of our designs using TV (and RFID) transmissions.

Designing such a wireless system is challenging for three main reasons: First, traditional radio receivers use power-intensive components such as oscillators that provide magnitude and phase information; the latter allows for gesture-recognition using Doppler frequency analysis. In contrast, AllSee uses passive components that significantly reduce the power consumption but provide only magnitude information. Thus, we need to develop algorithms and designs that can extract gesture information without relying on Doppler frequencies. Second, our designs should work on power-constrained devices and hence should be highly power-efficient and require minimal computational resources. Finally, gesture recognition is interactive in nature and hence requires short response times; this means that our algorithms and hardware should introduce minimal delays.

The rest of this section describes how AllSee addresses these challenges. We first describe AllSee's receiver design that extracts amplitude information using passive hardware components, and then present our algorithm to perform gesture classification using only this

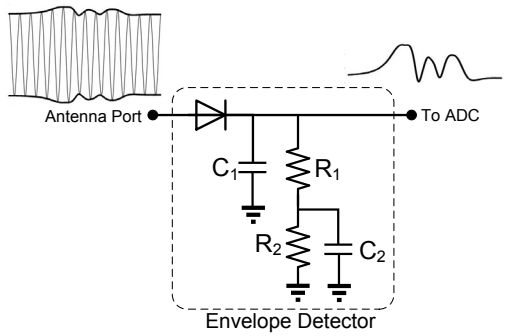


Figure 3: **AllSee's receiver circuit.** AllSee uses an envelope detector to extract the amplitude information. Using different values for the analog components, the receiver can work with both TV and RFID transmissions.

magnitude information.

3.1 AllSee's Receiver Design

We ask the following question: how can we extract amplitude information without using power-intensive components such as oscillators? Oscillators generate the carrier frequency (e.g., the 725 MHz TV carrier), and are typically used at receivers to remove the carrier frequency (down-conversion). At a high level, AllSee removes the carrier frequency by using an analog envelope detector. The rest of this section first describes how this works for constant-wave transmissions (e.g., RFID) and then extends it to work with fast-changing ambient TV transmissions.

(a) With Constant-Wave Transmissions: AllSee uses the envelope detector shown in Fig. 3 to remove the carrier frequency and extract amplitude information. As shown in the figure, the envelope detector tracks the envelope of the received signal, while eliminating the carrier frequency. The key point to note is that the envelope detector circuit is designed using passive analog components (diodes, resistors, and capacitors) and hence is ultra-low power in nature. The operating principle behind the design is similar to that used in RFID tags: To a first-order approximation, diodes act as switches that allow current to flow in the forward direction but not in the reverse; capacitors are charge-storage elements; and resistors regulate current flow. The diode in the above circuit provides charge to the capacitor C_1, whenever the input voltage is greater than the voltage at the capacitor. On the other hand, when the input is lower than the voltage at the capacitor, the diode does not provide charge and the resistors R_1 and R_2 slowly dissipate the energy stored on the capacitor, lowering its voltage. The rate of voltage drop is determined by the product $C_1 * (R_1 + R_2)$. Thus, by choosing appropriate values of R_1, R_2 and C_1, we can pick the rate at which the signal's envelope is tracked. Effectively, the above circuit acts as a low-pass filter, smoothing out the carrier in the constant-wave transmissions; the additional capacitor C_2 aids with this filtering.

Note that the envelope detector does not remove the amplitude variations caused by human gestures. This is because the circuit is tuned to track the variations caused by human motion which happen at a rate orders of magnitude lower than the carrier frequency.

(b) With Fast-Changing TV Transmissions: The key problem with ambient signals is that they have information encoded in them and hence have fast-varying amplitudes. For example, ATSC TV transmissions encode information using 8VSB modulation, which changes the instantaneous amplitude of the signal. In principle, the receiver could decode the TV transmissions and estimate the channel parameters to extract the amplitude changes that are due to human gestures. This is, however, infeasible on a power-constrained device. Thus, the challenge is to distinguish between the encoded TV information and the changes in the received signal due to human gestures, while operating on power-constrained devices.

Our key insight is that amplitude changes due to human gestures happen at a much lower rate than the changes inherent to TV transmissions. Our design leverages this difference in the rates to separate the two effects. Specifically, TV signals encode information at a rate of 6 MHz, but human gestures occur at a maximum rate of tens of Hertz. AllSee uses the envelope detector in Fig. 3 to distinguish between these rates. Specifically, we set the time constant of the envelope detector to be much greater than $\frac{1}{6MHz}$. This ensures that the encoded TV data is filtered out, leaving only the amplitude changes that are due to human gestures.

3.2 AllSee's Classification Logic

AllSee extracts gesture information from the signal output by the envelope detector. In this section, we first explain why prior approaches to wireless gesture recognition do not apply in our case. We then describe AllSee's algorithm to classify gestures.

3.2.1 Why Are Prior Approaches Not Applicable?

Prior approaches leverage wireless Doppler shifts for gesture classification. For example, the reflections from a user moving her hand toward the wireless receiver creates a positive Doppler shift. On the other hand, when the user moves her hand away from the receiver, it creates a negative Doppler shift. By using the sign of the Doppler shift, prior work [32] distinguishes between these gestures.

The problem is that the output of the envelope detector does not have phase information, which makes it difficult to apply the above approach. To understand this in more detail, let us consider a basic scenario where an RF source transmits a sinusoid with a frequency of f. Thus, the transmitted signal is given by:

$$sinftsinf_ct$$

where f_c is the transmitter's center frequency. Now, say that the user moves her arm towards the receiver and creates a Doppler shift, f_d. The receiver now receives the following signal [36]:

$$sinftsinf_ct + sinftsin(f_c + f_d)t$$

That is, the received signal is a linear combination of the direct signal from the RF source and the Doppler-shifted multi-path reflection from the user's arm. For simplicity, we assume that the user's reflection has the same signal strength as the direct signal, but the analysis would be similar in the general case. Now we can simplify the above equation to:

$$sinft(sinf_ct + sin(f_c + f_d)t) \quad (1)$$

$$= \quad 2sinftcos\tfrac{f_dt}{2}sin(f_c + \tfrac{f_d}{2})t \quad (2)$$

Traditional receivers use oscillators tuned to the center frequency f_c; hence they can extract the Doppler frequency f_d from the last sinusoid term in the above equation. AllSee, however, uses a low-power envelope-detector circuit that by nature is not as frequency-selective as an oscillator [19]. Specifically, the envelope detector tracks the envelope of the fastest-changing sinusoid in the received signal. Thus, in Eq. 2 the envelope detector considers the last sinusoid, $sin(f_c + \frac{f_d}{2})t$, as the effective transmitted signal and removes it. So, the output of the envelope detector is,

$$2sinftcos\tfrac{f_dt}{2}$$

$$= \quad sin(f + \tfrac{f_d}{2})t + sin(f - \tfrac{f_d}{2})t$$

Now if the receiver computes an FFT of the above signal, centered at f, it sees energy in both the positive and negative frequencies. This holds true independent of whether the user performs the push or the pull action. As a result, the receiver cannot distinguish between these two gestures using Doppler information.

Note that, from Eq. 2 the Doppler shift does not affect the transmitted signal $sinft$. Hence, replacing the transmitted sinusoid with any other signal does not change the above analysis.

3.2.2 AllSee's Time-Domain Analysis

AllSee leverages both the structure of the magnitude changes as well as the timing information to classify gestures. To see this, consider the push and pull gestures. As the user moves her arm towards the receiver, the changes in magnitude increase, as the arm gets closer to the receiver. This is because the reflections from the user's arm undergo lower attenuations as the arm gets closer. When the user moves her arm away from the receiver, the changes in the magnitude decrease with time. Thus, the changes in the time-domain signal can be uniquely mapped to the push and the pull gestures as shown in Figs. 4(b) and (c).

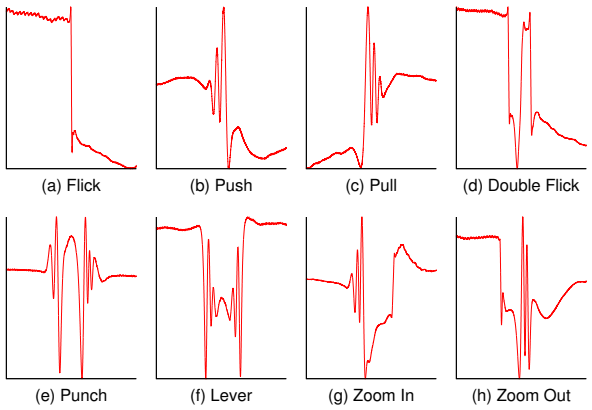


Figure 4: **Changes created on the envelope detector's output.** The amplitude changes have a unique correspondence to the gestures in Fig. 2.

AllSee also leverages timing information to classify gestures. Specifically, the wireless changes created by the flick gesture, as shown in Fig. 4(a), occurs for a shorter period of time compared to a push or a pull gesture. Using this timing information one can distinguish between these three gestures. Fig. 4 plots the amplitude changes created by our gestures as a function of time.

In the rest of this section, we describe in more detail how AllSee identifies and classifies these gestures. Specifically, AllSee uses a low-rate 10-bit ADC operating at 200 Hz to digitally process the signal output by the envelope detector. AllSee's time-domain classification algorithm has three main steps: (1) Signal conditioning to remove location dependence, (2) Segmentation to identify a time-domain segment that corresponds to a gesture, and (3) Classification to determine the most likely gesture amongst a set of gestures.

(1) Signal Conditioning: Our goal is to extract a deterministic location-independent mapping between gestures and amplitude changes. Note that the actual amplitudes vary with the user's position. For example, when the user performs the push gesture, the initial and final amplitudes depend on where the user starts and ends the push action. AllSee removes these location dependencies by performing a moving average over a time window (set to 320 ms in our implementation) and subtracting the average from

each digital sample returned by the ADC; thus, effectively normalizing the received signal.

(2) Segmentation: AllSee uses amplitude changes to detects the start and end of a gesture. Specifically, it computes a derivative of the received signal, i.e., the difference between the current and the previous sample. When this difference rises above a threshold (set to 17.5 mV in our implementation), AllSee detects the beginning of a gesture. Similarly when this difference falls below the same threshold, we detect the end of the gesture. The use of the derivative operation to detect the beginning and end of a gesture works because, as shown in Fig. 4, the changes caused by a gesture tend to be high. This results in large differences between adjacent samples, which we can use for segmentation.

We note the following: First, in comparison to ambient human motion such as walking and running, the changes between adjacent samples tend to be higher during intentional gestures close to the device. Thus, the above segmentation procedure helps reduce the false positive rate. Second, in some of our experiments, the difference between adjacent samples prematurely dropped below the threshold, even before the end of the gesture. It rises back up soon afterward, creating multiple close-by segments. To avoid this being detected as multiple gestures, we combine any two segments that occur within 75 milliseconds into a single segment.

(3) Gesture Classification: In principle, one could run signal-processing algorithms such as dynamic time warping to distinguish between the signals in Fig. 4. However, this is not desirable since it increases the computational complexity. Instead, AllSee uses simple rules that have minimal complexity to distinguish between gestures. For example, to classify between the push and pull gestures, we use the following rule: if the maximum changes in the signal occurs closer to the start of a gesture segment, it is a pull action; otherwise, it is a push action. In Alg. 1 we describe the rules for all eight of our gestures. We note that the algorithm is a set of if-then-else statements; in the worst case, these require only 56 instructions on an MSP430 microcontroller [10].

Algorithm 1 Gesture Classification

```
while Vsig − Vsig_prev < THRESHOLD do
    LOWPOWERSLEEP()
end while
[length, maxIndex] ← GETGESTURE()
g0 ← CLASSIFYSUBGESTURE(length, maxIndex)

[length, maxIndex] ← GETGESTURE()
g1 ← CLASSIFYSUBGESTURE(length, maxIndex)

if (g0 = FLICK and g1 = FLICK) then return D_FLICK
else if (g0 = FLICK and g1 = PULL) then return Z_OUT
else if (g0 = PUSH and g1 = FLICK) then return Z_IN
else if (g0 = PUSH and g1 = PULL) then return PUNCH
else if (g0 = PULL and g1 = PUSH) then return LEVER
else if (g0 = FLICK and g1 = NULL then return FLICK
else if (g0 = PUSH and g1 = NULL) then return PUSH
else if (g0 = PULL and g1 = NULL) then return PULL
end if

function CLASSIFYSUBGESTURE(length, maxIndex)
    if (length < FLICKLENGTH) then return FLICK
    else if (maxIndex < length/2) then return PUSH
    else if (maxIndex ≥ length/2) then return PULL
    end if
end function
```

3.3 Analog Gesture Decoding

In this section, we ask if we can further reduce the power consumption of the above design. As we see later in §5, the main factor that contribute to power consumption is the ADC. Specifically, we require an ADC with a resolution of eight to ten bits for the classification algorithm in §3.2 to work with high accuracy. So our goal is to eliminate the need for such high-resolution ADCs.

Our idea is to encode gesture information directly using analog components such as capacitors and resistors. Such an approach could reduce the need to processing the signals in the digital domain and hence avoids high-resolution ADCs. To show the feasibility of this idea, we design a circuit, shown in Fig. 5, that can distinguish between the punch and the flick gestures from Fig. 2. The circuit has four main components: an envelope detector to remove the carrier frequency, a second envelope detector that tracks time-domain changes caused by the gestures at a slow rate, an averaging circuit that computes the slow-moving average of the second envelope detector, and finally a low-power comparator that outputs bits.

Fig. 5 annotates the signals for the two gestures at each stage of the circuit. After the first envelope detector, the signals no longer have the carrier frequency. The second envelope detector tracks the signal at a much lower rate, and hence the punch signal looks like an increase and then a decrease in the amplitude levels; this corresponds to starting the arm at an initial state and then bringing it back to the same state. The flick signal, on the other hand, is a transition between two reflection states: one where the fingers are closed to another where the fingers are wide open.

The averaging circuit and the comparator allow us to distinguish between these two signals. Specifically, the averaging circuit further averages these signals to create the red signals shown in the figure. Now the comparator takes these signals and their average values as inputs, and outputs a '1' bit whenever the signal is greater than the

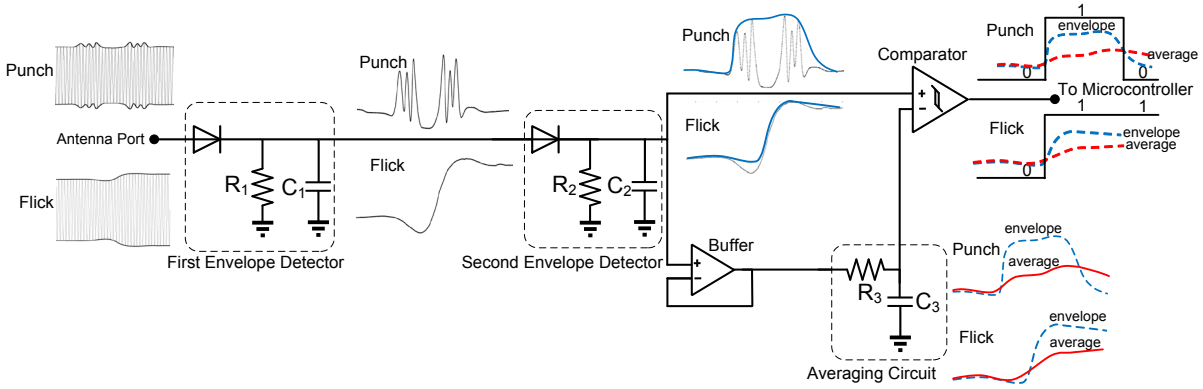


Figure 5: **AllSee's analog gesture encoding.** The circuit has four main components: an envelope detector to remove the carrier frequency, a second envelope detector that tracks changes caused by the gestures at a slow rate, an averaging circuit and finally a low-power comparator that outputs bits. The buffer ensures that the envelope detector and the averaging circuit do not affect each other. At each stage, input (dashed lines) and output (solid lines) waveforms corresponding to the punch and flick gestures are annotated.

average and a '0' bit otherwise. Thus, the comparator outputs unique set of bit patterns for the two gestures (010 and 011 in the figure). Thus, we can classify these gestures with almost no computational complexity.

We note three main points: First, the comparator is essentially a one-bit ADC; it has minimal resolution and hence consumes the least amount of power. Second, the parameters in our circuit are chosen to account for the timing information inherent in our specific gestures. Thus, it is unlikely that random human motions would trigger the same bit patterns. Third, while the above circuit only classifies the flick and the punch gestures, in principle, one can use higher order capacitor and resistor networks to encode all the eight gestures. This however is not in the scope of this paper.

4 Hardware Design

Fig. 6 shows the general hardware design of an AllSee device. It has two core components: an AllSee gesture receiver, and the computational logic that detects and classifies human gestures. Our implementation performs the digital logic operations on a low-power microcontroller. The AllSee gesture receiver primarily is the design in §3 that extracts amplitude information. However, it could also incorporate the analog gesture encoding mechanism described in §3.3.

The figure also shows optional components: a transmitter and receiver for communications, an RF energy harvester and the power management circuit to extract power from RF signals of either TV towers or RFID readers. These components are essential in devices such as RFID tags and ambient RF-powered devices, but are not necessary when AllSee is used in battery-powered mobile devices such as smartphones. We note that our design could, in principle, be incorporated into sensor hubs that are becoming popular on mobile devices.

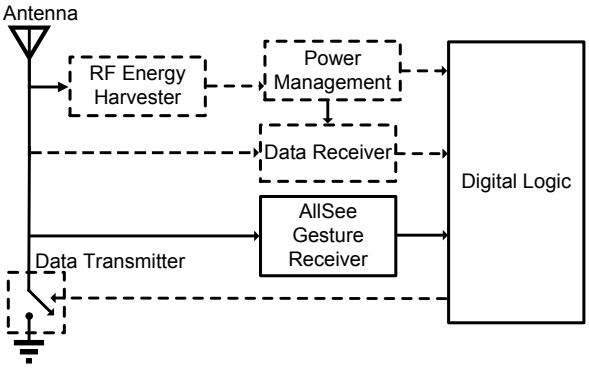


Figure 6: **AllSee's Hardware Design.** It consists of two main components: the wireless receiver for gesture recognition and our classification logic. The other components such as transmitter and receiver for communications, and energy harvester are optional.

5 Prototype Implementation

AllSee prototypes are implemented on two-layer printed circuit boards (PCBs) using off-the-shelf commercial circuit components. The PCBs were designed using the Altium design software and manufactured by Sunstone Circuits. The capacitor and resistor values R_1, R_2, C_1 and C_2 shown in Fig. 3 are set to 150 $k\Omega$, 10 $M\Omega$, 27 nF and 0.2 μF respectively. Our pluggable gesture recognition component consists of a low-power microcontroller (MSP430F5310 by Texas Instruments [10]) and an in-

Table 1: **AllSee's Average Power Consumption**

	ADC-based	Analog-based
No Gestures	26.96 μ W	4.57 μ W
15 Gestures/minute	28.91 μ W	5.85 μ W

terface to plug in our wireless receivers. It also features a UART interface to send data to a computer for debugging purposes as well as low-power LEDs. The output from the wireless receivers is sampled by an ADC at a frequency of 200 Hz (i.e., generating a digital sample every 5 ms). In most of our experiments, AllSee uses 10 bits of resolution at the ADC; in §6.1, however, we use other resolutions and sampling rates to understand their effects on classification accuracy.

To minimize power consumption, the microcontroller sleeps most of the time. The ADC wakes up the microcontroller to deliver digital samples every 5 ms. The microcontroller processes these samples before going back to sleep mode. The maximum time spent by the microcontroller processing a digital sample is 280 μs.

We also build a prototype for the analog gesture encoding system described in §3.3 by incorporating additional components into our wireless receivers. Specifically, we use an ultra-low power comparator, TS881 [15], and implement the buffer using an ultra-low power operational amplifier (ISL28194 [8]). The output of the comparator is fed to the digital input-output pin of the microcontroller. The capacitor and resistor values R_1, R_2, R_3, C_1, C_2 and C_3 shown in Fig. 5 are set to 470 $k\Omega$, 56 $M\Omega$, 20 $k\Omega$, 0.47 μF, 22 μF and 100 μF respectively. The RF energy harvester, transmitter, receiver and power-management circuit we use in our prototypes are similar to those in previous work [30, 34, 35].

Table 1 shows the total power consumption of our prototypes. For the ADC-based prototype, the 10-bit ADC continuously sampling at 200 Hz consumes 23.867 μW. The micro-controller consumes 3.09 μW for signal conditioning and gesture segmentation and 1.95 μW for gesture classification (as described in §3.2.2); the average power consumption is 26.96 μW when no gestures are present and 28.91 μW when classifying 15 gestures (including the starting gesture) per minute. In the analog-based prototype, the hardware components, the buffer and the comparator consume a total of 0.97 μW. The micro-controller consumes 3.6 μW in sleep mode (i.e., no bit transitions at the comparator's output). The average power consumption for the analog-based system is 4.57 μW when no gestures are present and 5.85 μW when classifying 15 gestures per minute.[2]

[2]Note that our prototypes use off-the shelf components and a general purpose micro-processor. One can further reduce the power consumption by using application specific integrated circuits.

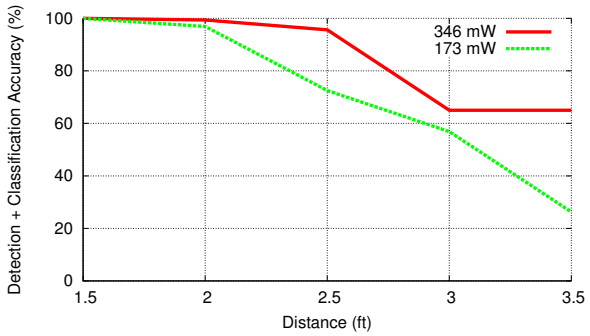


Figure 7: **Effect of distance and transmit power level.** The plots show results for two transmit power levels at the RFID reader. We operate in conservative settings – commercial RFID readers operate at up to 1 W.

6 Evaluation

We first present micro-benchmarks to understand the effect of various parameters on our system. We then evaluate different system aspects including classification accuracy, response time, and false-positive rate.

6.1 Micro-Benchmarks

We evaluate the key aspects that affect classification accuracy: (1) the user's distance from the prototype and the transmit power level, and (2) the bit-resolution and sampling rate of the ADC. Since it is easier to run controlled benchmark experiments with RFID readers than with TV towers, in this section we use our RFID prototype in the presence of an RFID reader.

Effect of distance and transmit power level: We run experiments to understand the effects of these parameters on classification accuracy. Specifically, we run our USRP-based RFID reader at two different transmit power levels: 346 mW and 173 mW. Note that commercial RFID readers can go upto 1 W; thus, we are operating in more conservative settings. For each of the power levels, we place our RFID-prototype in the decoding range of the reader. We then have the user stand at different distances from our prototype and perform our eight gestures, 20 times each, without fully blocking the signal from the RFID reader. Note that the users were not trained to orient themselves in a particular direction. At each of these distances, we compute the average classification accuracy across all eight gestures.

Fig. 7 shows classification accuracy as a function of the user's distance from our prototype. The plots show the following:

• As the distance between the user and the device increases, the classification accuracy decreases. This is expected since the strength of the wireless reflections from

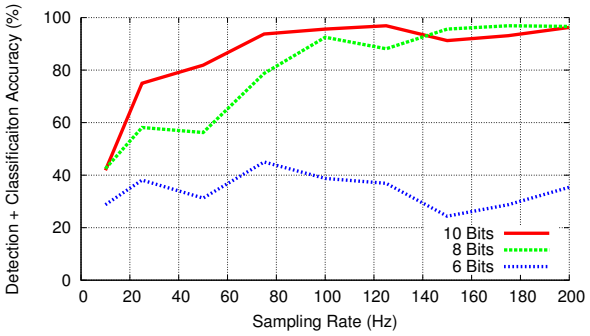


Figure 8: **Effect of ADC parameters.** The plot shows the accuracies at three different ADC bit resolutions.

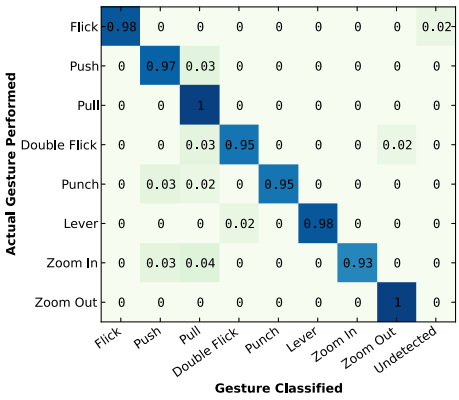


Figure 9: **Confusion matrix for our RFID prototype.** The average classification accuracy across all the gestures is 97%.

the user's body decreases as the user moves away from the device. This makes it harder to detect minute gestures such as the flick motion, bringing down the classification accuracy. We note, however, that the accuracies are greater than 90% at distances of 2.5 feet and 2 feet respectively for the two power levels. Such distances are sufficient for most gesture-recognition applications on mobile devices and sensors.

• As the transmit power level decreases, the classification accuracy decreases. This is because our wireless receiver has a minimum sensitivity below which it cannot accurately track changes from human gestures. We note, however, that the power levels we use in this experiment are lower than the 1 W power used by a commercial RFID reader. Further, our prototype RFID tags have about 11 dBm lower sensitivity than commercial tags. So while the trends we see here would hold for commercial tags, one can expect that the system would work at larger distances between the user and the prototype.

Effect of ADC parameters: The power consumption increases with the sampling rate and the bit resolution at the ADC. To empirically evaluate this effect, we run experiments in the presence of an RFID reader with a transmit power of 346 mW. We compute the classification accuracies for different ADC sampling frequencies and 6-bit, 8-bit and 10-bit resolutions. The user performs the eight gestures 20 times each, for each combination of sampling rates and resolutions.

Fig. 8 shows the detection and classification accuracies for the eight gestures shown in Fig. 2 as a function of the sampling rate. The different curves correspond to different bit resolutions. The figure shows that the accuracy typically increases with the ADC's bit resolution. The accuracy is also lower at low sampling rates; this is expected because as the sampling rates decrease, we lose timing information about the gestures.

6.2 AllSee's Classification Accuracy

We evaluate AllSee's accuracy in classifying gestures with our RFID-based and TV-based prototypes using the ADC design in §3.2.

Evaluating Our RFID-Based Prototype: We first evaluate the classification accuracy of gesture recognition using RFID signals.

Experiments: We run experiments in locations spanning two lab spaces in the UW CSE building. To check if AllSee works in the presence of multi-path reflections from nearby objects, one of these locations has strong reflective surfaces such as walls and objects (metallic cupboards, desks) close to our prototype hardware. In each of these locations, our prototype is placed in positions that are in the decoding range (about 1 meter) of an USRP-based RFID reader with a 346 mW transmit power. In our experiments, users perform the gestures in Fig. 2 at a distance of 1.5 to 2.5 feet away from the prototype. Before each experiment, users were shown how to perform all of the gestures. We ran the experiments with five users who volunteered to perform gestures; one of the five users is a co-author of this paper. Each gesture is performed a total of 20 times. The gestures are detected and classified on the microcontroller on our hardware prototype using the algorithm described in Alg. 1.

Results: Fig. 9 plots the confusion matrix where each row denotes the actual gesture performed by the user and each column the gestures it was classified into. The last column counts the fraction of gestures that were not detected at the receiver. Each element in the matrix corresponds to the fraction of gestures in the row that were classified as the gesture in the column; the fraction is computed across all the locations and the users. The table shows the following:

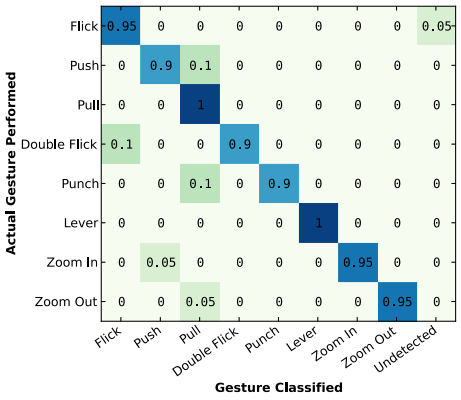


Figure 10: **Confusion matrix for our TV prototype.** The average classification accuracy across all the gestures is 94.4%. The lower accuracy is due to lower RF frequency of TV transmissions.

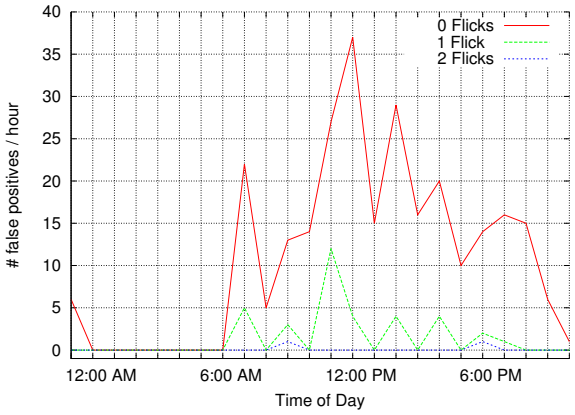


Figure 11: **False Positive Rate from a 24-Hour trace.** The figure plots the false positive rate when we use a flick gesture as a starting sequence.

• The average accuracy is 97% with a standard deviation of 2.51% when classifying among our eight gestures. This is in comparison to a random guess, which has an accuracy of 12.5% for eight gestures. This shows that one can extract rich information about gestures from time-domain wireless signals. We note that all the detection and classification operations were performed on our hardware prototype. This demonstrates the feasibility of achieving gesture recognition on battery-free devices.

• The flick motion was the most likely gesture to be undetected. This is because the flick gesture involves only finger movements; wireless reflections from the fingers have a much lower energy than those from the whole arm. Thus, the receiver has a higher probability of not detecting these gestures. Finally, there are small variations in the signals, due to differences in gesture articulation across users. These variation however are small and do not necessitate per-user tuning.

Evaluating Our TV-Based Prototype: Next we evaluate our accuracy using ambient TV signals.

Experiments: We run experiments using our battery-free prototype that harvests TV signals for power. We use the receiver design from our prior work on ambient backscatter devices [30] that can currently operate up to 10 Km away from a TV tower with power levels ranging between -24 dBm and -8 dBm. AllSee operates at similar distances and power levels from the TV tower. In principle, we can increase the distance and power sensitivity by using ASIC designs; this, however, is not in the scope of this paper. Our receiver prototype is tuned to harvest power and extract gesture information from TV signals in the 50 MHz band centered at 725 MHz. The users stand in a random location 1.5 to 2.5 feet away from the AllSee receiver and randomly perform all our eight gestures 20 times each. As above, the microcontroller on our prototype detects and classifies the gestures. We extract this information and compute the classification accuracy for the gestures.

Results: Fig. 10 shows the confusion matrix for our TV-based prototype. The figure shows similar trends to the results with our RFID-based prototype. The classification accuracies with our TV-based prototype, however, are lower than those with RFID signals. The main reason for this is the lower transmission frequency of TV signals. Specifically, lower transmission frequencies (higher wavelengths) require larger displacements to have a similar change in the received signals. Since RFID transmissions occur at 915 MHz, small displacements (e.g., a flick gesture) create large changes in the wireless signal. In contrast, since the TV transmissions occur at 725 MHz, the extent of wireless changes is smaller, making it harder to detect such gestures.

6.3 AllSee's False Positive Rate

To avoid random human motion near the device from being classified as the target gestures, AllSee uses a unique gesture sequence (a repetitive flick gesture) at the beginning to detect the target human. In this section, we evaluate the effectiveness of such an approach.

Experiments: Since our prototypes have the range of a few feet, we stress-test our system by placing them next to a participant's desk. Specifically, we run experiments in our lab over a 24-hour period during which the participant continues to perform activities including typing, eating, and moving around in the chair. Our prototype's location is such that five other lab occupants have to get as close as a foot to enter or leave their workspace.

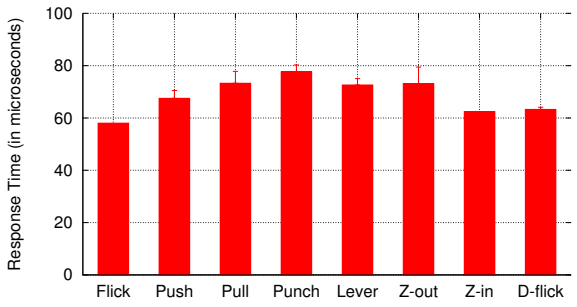


Figure 12: **Response time for various gestures.** The maximum response time is less than 80 μs across all the gestures. Z-in, Z-out, and D-flick refer to the zoom in, zoom out, and double flick gestures.

Results: Fig. 11 plots the average number of false detection events per hour as a function of time. The results show that when the receiver does not use a starting sequence (zero repetitions), the number of false positive events is about 11.1 per hour over the 24-hour period. We note that this is surprisingly low despite running experiments next to the user. This is because, as explained in §3.2, our segmentation algorithm is designed to only account for the large instantaneous changes in the received amplitude, that occur with intentional gestures; this reduces the probability of ambient motion resulting in the expected segment lengths. The results also show that the average number of false-positive events reduces to 1.46 per hour when a single flick action is used as a starting sequence. This is because the flick action creates a very short burst of amplitude changes that rarely occur with typical human activities. Note that as the number of flick repetitions increase, the false-positive rate reduces. Specifically, with two repetitions, this average rate reduces to 0.083 events per hour. This is again because a repetitive flick motion creates periodic short bursts of amplitude changes, which are unlikely to occur with typical activities. Further, since we expect these bursts to have a specific range of periodicities, random mechanical and environmental variations are unlikely to be confused for the repetitive flick sequence.

6.4 AllSee's Response Time

Short response times are important for the interactive nature of gesture recognition. Here, we evaluate AllSee's response time, i.e., the time between the completion of a gesture and its classification by our prototype. Recall that the microcontroller runs instructions at 1 MHz and the ADC operates at 200 Hz, waking up the microcontroller every 5 ms to deliver the digital samples. A 1 MHz microcontroller can run up to $10,000$ instructions in 10 ms. The number of instructions in Alg. 1 is significantly smaller and hence 10 ms is an upper bound on AllSee's response time.

Experiments: To compute the exact response time, we measure the time difference between when the user finishes performing the gesture and when the microcontroller outputs the classified gesture. We program the microcontroller to toggle two output pins: first pin when it receives a data sample from the ADC and the second at the end of gesture classification. We observe the two output pins using an oscilloscope. In the absence of gestures, the first pin toggles periodically at 5 ms (sampling rate of ADC) and the second pin is steady. In the presence of a gesture, however, the second pin toggles immediately after classification. We compute the response time by measuring the time difference between the second pin's toggle and the periodic toggle on the first pin right before it. During the evaluation, the user performs our eight gestures 20 times and we compute the response time for each gesture averaged across the 20 repetitions.

Results: Fig. 12 shows the average measured response times for all the gestures. The plot shows the following:

- The maximum response time across all the gestures is less than 80 μs. This demonstrates that AllSee's gesture recognition algorithm requires negligible computation that can be performed even on an MSP430 with limited memory and computational capabilities.
- The variance of the response time, within the repetitions of the same gesture, is between 2–3 μs. This is because across experiments, the number of instructions that need to be run per gesture remains constant and deterministic. The only variability comes from the operational frequency of the microcontroller which is 1 MHz and hence has a resolution of 1 μs.

Table 2: **Classification With Analog Gesture Encoding**

	Classification Rate
Punch Gesture	25/25
Flick Gesture	23/25

6.5 Evaluating Analog Gesture Encoding

Next, we evaluate our prototype for analog gesture encoding in the presence of RFID signals. The user stands at a distance of two feet away from our prototype tag and randomly performs the flick and punch gestures, 25 times each. Our prototype detects these gestures using analog components such as capacitors and resistors as described in §3.3. We extract the results and compute the classification rates for the two gestures.

Table 2 shows the classification results. They show that while the punch gesture was always correctly detected and classified, the flick gesture was misclassified

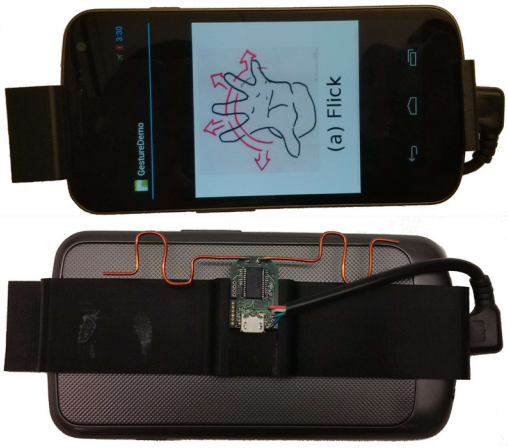


Figure 13: **Phone Prototype.** We interface a Galaxy Nexus with our miniaturized prototype mounted on a 3D printed phone case. We achieve 92.5% accuracy for the gestures in Fig. 2, in through-the-pocket scenarios.

in 2 out of the 25 trials. This is because the flick action involves finger motion that has less effect on the wireless signals than the arm motion in the punch action. Note that the average accuracy across the two gestures is about 96%. These results show the feasibility of our analog gesture encoding approach.

6.6 Through-the-Pocket Gestures

The ability to change the music or mute the phone while it is in a pocket, purse, or jacket is a useful capability. In this section, we integrate our miniaturized hardware prototype with a Samsung Galaxy Nexus smartphone [7] and demonstrate gesture recognition in such scenarios. We designed a 3-D printed case to mount our prototype on the phone. Further, as shown in Fig. 13, we modified the antenna for a better form factor and connect our prototype to the phone via a USB OTG cable.

We evaluate our smartphone prototype by placing the device in the pocket of a jacket that the user is wearing. The user then performs the eight gestures in Fig. 2 on the same x-y plane as the phone, 20 times each. Our results show that the mean accuracy across gestures is about 92.5%, which is encouraging. We note two main points: First, in comparison to the previous scenarios, the classification accuracy here is a bit lower. This is because, in these experiments, our device is hidden behind the jacket fabric and hence experiences higher signal attenuation. Second, our proof-of-concept prototype is limited to scenarios where the user is stationary and does not walk/run while performing the gestures. In principle, one can leverage other low-power sensors such as accelerometers on the phone to detect these scenarios; this however is not in the scope of this paper.

7 Discussion

We describe how one may augment AllSee's current design to make it more ubiquitous and robust.

Leveraging cellular and Wi-Fi signals: Our current prototypes work with TV and RFID transmissions. However, building AllSee prototypes that work with other RF signals such as Wi-Fi and cellular transmissions can enable greater ubiquity. We believe that the techniques developed in this paper can provide the framework for such designs. For instance, one could design specific envelope detectors that focus on human gestures while eliminating uncorrelated changes caused by the burstiness in these systems. Further, to address the issue of non-continuous transmissions, one can leverage the periodically transmitted beacons or pilot symbols in Wi-Fi and cellular systems to extract gesture information.

Reducing the gesture recognition range: One of the advantages of using wireless signals is that they enable far-field gesture recognition. However, one could reduce the sensitivity of our wireless receivers to reduce their range. This could be beneficial in certain applications where proximity is used as a proxy for access control.

Integration with other power sources: Our current design uses existing signals (e.g., TV and RFID transmissions) as a source of both power as well as gesture information. In principle, however, one can use AllSee to enable gesture recognition on other harvesting devices where we extract gesture information from wireless signals but use solar or mechanical energy for harvesting.

8 Conclusion

We introduce AllSee, a novel gesture recognition system that consumes three to four orders of magnitude lower power than the state-of-the-art systems today. AllSee can operate on battery-free devices such as ambient RF powered devices and RFID tags; it also enables always-on gesture recognition on mobile devices such as smartphones and tablets. We build prototypes and demonstrate that our system can detect and classify a set of eight gestures with classification accuracies as high as 97%. We believe that this is a promising result and hope that the techniques developed in this paper would take us closer to the vision of ubiquitous gesture interaction.

Acknowledgements: We thank the members of the UW Networks and Wireless group, David Wetherall, Ben Ransford, our shepherd Lili Qiu, and the anonymous NSDI reviewers for their helpful comments and Lilian de Greef for help with the gesture sketches in Fig. 2. This research is funded in part by a Google Faculty Research Award and a Washington Research Foundation gift.

References

[1] Advantaca. http://www.advantaca.com/AboutAdvantaca.htm.

[2] Air Gesture on Samsung S4 drains battery. http://www.1techportal.com/2013/05/maximize-your-galaxy-s4s-battery-in-five-ways/.

[3] Air Gesture on Samsung S4 works well under very well lit conditions. http://touchlessgeneration.com/discover-touchless/testing-of-air-gestures-on-the-galaxy-s4/#.UkSVWIY3uSo.

[4] AN580: Ifrared gesture recognition by Silicon Labs. http://www.silabs.com/Support%20Documents/TechnicalDocs/AN580.pdf.

[5] CrunchFish. http://crunchfish.com/.

[6] GestIC: electrical near field (E-field) 3D Tracking and Gesture Controller by Microchip. http://www.microchip.com/pagehandler/en-us/technology/gestic/home.html.

[7] Google Nexus S. http://www.android.com/devices/detail/nexus-s.

[8] ISL28194 op amp datasheet by Intersil. http://www.intersil.com/content/dam/Intersil/documents/fn62/fn6236.pdf.

[9] Leap Motion. https://www.leapmotion.com/.

[10] MSP430F5310 micro-controller by Texas Instruments. http://www.ti.com/product/msp430f5310.

[11] Myo by Thalmic Labs. https://www.thalmic.com/en/myo/.

[12] Point Grab. http://www.pointgrab.com/.

[13] The PrimeSense 3D Awareness Sensor. http://www.primesense.com/wp-content/uploads/2012/12/PrimeSense-3D-Sensor-A4-Lo.pdf.

[14] Smart Things. http://www.smartthings.com/.

[15] TS 881 comparator datasheet by STMicroelectronics. http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/DM00057901.pdf.

[16] USRPN200/N210 by Ettus Research. https://www.ettus.com/content/files/07495_Ettus_N200-210_DS_Flyer_HR.pdf.

[17] Xbox Kinect. http://www.xbox.com/en-US/kinect.

[18] F. Adib and D. Katabi. Seeing Through Walls Using WiFi! In *SIGCOMM*, 2013.

[19] R. Barnett, G. Balachandran, S. Lazar, B. Kramer, G. Konnail, S. Rajasekhar, and V. Drobny. A passive uhf rfid transponder for epc gen 2 with -14dbm sensitivity in 0.13 μm cmos. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 582–623, 2007.

[20] G. Charvat, L. Kempel, E. Rothwell, C. Coleman, and E. Mokole. A Through-dielectric Radar Imaging System. In *Trans. Antennas and Propagation, 2010*.

[21] K. Chetty, G. Smith, and K. Woodbridge. Through-the-wall Sensing of Personnel Using Passive Bistatic WiFi Radar at Standoff Distances. In *Trans. Geoscience and Remote Sensing, 2012*.

[22] G. Cohn, S. Gupta, T.-J. Lee, D. Morris, J. R. Smith, M. S. Reynolds, D. S. Tan, and S. N. Patel. An ultra-low-power human body motion sensor using static electric field sensing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12. ACM, 2012.

[23] G. Cohn, D. Morris, S. Patel, and D. Tan. Humantenna: using the body as an antenna for real-time whole-body interaction. In *CHI'12*.

[24] G. Cohn, E. Stuntebeck, J. Pandey, B. Otis, G. D. Abowd, and S. N. Patel. Snupi: sensor nodes utilizing powerline infrastructure. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*, Ubicomp '10. ACM, 2010.

[25] A. Czeskis, K. Koscher, J. R. Smith, and T. Kohno. Rfids and secret handshakes: defending against ghost-and-leech attacks and unauthorized reads with context-aware communications. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08. ACM, 2008.

[26] P. Dutta, A. Arora, and S. Bibyk. Towards radar-enabled sensor networks. In *Information Processing in Sensor Networks, 2006. IPSN 2006. The Fifth International Conference on*, pages 467–474, 2006.

[27] P. Dutta and A. K. Arora. Integrating micropower radar and motes. IEEE Conf. on Ultra Wideband Systems and Technologies, Baltimore, MD, 2002.

[28] S. Gupta, D. Morris, S. Patel, and D. Tan. Soundwave: using the doppler effect to sense gestures. In *HCI 2012*.

[29] L. Kriara, M. Alsup, G. Corbellini, M. Trotter, J. D. Griffin, and S. Mangold. RFID Shakables: Pairing Radio-Frequency Identification Tags with the Help of Gesture Recognition. In *ACM CoNEXT*, 2013.

[30] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient Backscatter: Wireless Communication Out of Thin Air. In *ACM SIGCOMM*, 2013.

[31] Microchip. GestIC: Single-Zone 3D Tracking and Gesture Controller Data Sheet, 2012-2013.

[32] Q. Pu, S. Gupta, S. Gollakota, and S. Patel. Whole-Home Gesture Recognition Using Wireless Signals. In *MOBICOM*, 2013.

[33] T. Ralston, G. Charvat, and J. Peabody. Real-time through-wall imaging using an ultrawideband MIMO phased array radar system. In *Array, 2010.*

[34] A. Sample and J. R. Smith. Experimental results with two wireless power transfer systems. In *Radio and Wireless Symposium, 2009. RWS '09. IEEE*, pages 16 –18, jan. 2009.

[35] A. Sample, D. Yeager, P. Powledge, A. Mamishev, and J. Smith. Design of an rfid-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, November 2008.

[36] D. Tse and P. Viswanath. *Fundamentals of wireless communication*. Cambridge University Press, New York, NY, USA, 2005.

3D Tracking via Body Radio Reflections

Fadel Adib Zachary Kabelac Dina Katabi Robert C. Miller
Massachusetts Institute of Technology

Abstract – This paper introduces WiTrack, a system that tracks the 3D motion of a user from the radio signals reflected off her body. It works even if the person is occluded from the WiTrack device or in a different room. WiTrack does not require the user to carry any wireless device, yet its accuracy exceeds current RF localization systems, which require the user to hold a transceiver. Empirical measurements with a WiTrack prototype show that, on average, it localizes the center of a human body to within a median of 10 to 13 cm in the x and y dimensions, and 21 cm in the z dimension. It also provides coarse tracking of body parts, identifying the direction of a pointing hand with a median of 11.2°. WiTrack bridges a gap between RF-based localization systems which locate a user through walls and occlusions, and human-computer interaction systems like Kinect, which can track a user without instrumenting her body, but require the user to stay within the direct line of sight of the device.

1 Introduction

Recent years have witnessed a surge in motion tracking and localization systems. Multiple advances have been made both in terms of accuracy and robustness. In particular, RF localization using WiFi and other communication devices has reached sub-meter accuracy and demonstrated its ability to deal with occlusions and non-line of sight scenarios [31, 18]. Yet these systems require the user to carry a wireless device in order to be localized. In contrast, systems like Kinect and depth imaging have revolutionized the field of human-computer interaction by enabling 3D motion tracking without instrumenting the body of the user. However, Kinect and imaging systems require a user to stay within the device's line-of-sight and cannot track her across rooms. We envision that if an RF system can perform 3D motion tracking without requiring the user to wear a radio, it will motivate the integration of such a technology in systems like Kinect to expand their reach beyond direct line of sight and enable through-wall human-computer interaction.

Motivated by this vision, this paper introduces WiTrack, a system that tracks the 3D motion of a user using radio reflections that bounce off her body. It works through walls and occlusions, but does not require the user to carry any wireless device. WiTrack can also provide coarse tracking of a body part. In particular, the user may lift her hand and point at objects in the environment; the device detects the direction of the hand motion, enabling the user to identify objects of interest.

WiTrack has one antenna for transmission and three antennas for receiving. At a high level, WiTrack's motion tracking works as follows. The device transmits a radio signal and uses its reflections to estimate the time it takes the signal to travel from the transmitting antenna to the reflecting object and back to each of the receiving antennas. WiTrack then uses its knowledge of the position of the antennas to create a geometric reference model, which maps the round trip delays observed by the receive antennas to a 3D position of the reflecting body.

Transforming this high-level idea into a practical system, however, requires addressing multiple challenges. First, measuring the time of flight is difficult since RF signals travel very fast – at the speed of light. To distinguish between two locations that are closer than one foot apart, one needs to measure differences in reflection time on the order of hundreds of picoseconds, which is quite challenging. To address this problem, we leverage a technique called FMCW (frequency modulated carrier wave) which maps differences in time to shifts in the carrier frequency; such frequency shifts are easy to measure in radio systems by looking at the spectrum of the received signal.

A second challenge stems from multipath effects, which create errors in mapping the delay of a reflection to the distance from the target. WiTrack has to deal with two types of multipath effects. Some multipath effects are due to the transmitted signal being reflected off walls and furniture. Others are caused by the signal first reflecting off the human body then reflecting off other objects. This is further complicated by the fact that in non-line-of-sight settings, the strongest signal is not the one directly bouncing off the human body. Rather it is the signal that avoids the occluding object by bouncing off some side walls. WiTrack eliminates reflections from walls and furniture by noting that their distance (and time of flight) does not change over time. Hence, they can be eliminated by subtracting consecutive frames of the signals. Reflections that involve a combination of a human and some static object are more complex and are addressed through filters that account for practical constraints on the continuity of human motion and its speed in indoor settings.

We have built a prototype of WiTrack and evaluated it empirically. Since off-the-shelf radios do not perform FMCW, we built an analog FMCW radio frontend, which operates as a daughterboard for the USRP software radio. In our evaluation, we use the VICON motion capture system [6] to report the ground truth location. VICON can achieve sub-centimeter accuracy but requires instrument-

ing the human body with infrared markers and positioning an array of infrared cameras on the ceiling. Since VICON cannot operate in non-line-of-sight, the human moves in the VICON room while our device is placed outside the room and tracks the motion across the wall. Our evaluation considers three applications, each of them uses the developed 3D tracking primitive in a different way.

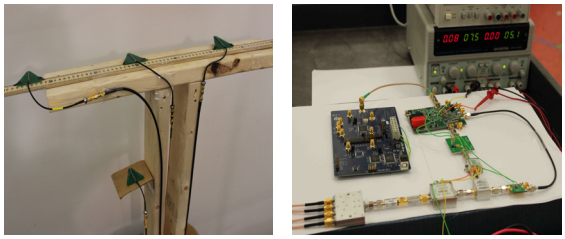
In the first application, we consider 3D tracking of human motion through a wall. The objective of such an application is to augment virtual reality and gaming systems to work in non-line-of-sight and across rooms. We compute the tracking error as the difference between the location reported by our device and the actual location of the body center as reported by VICON. Our results show that WiTrack localizes the center of the human body to within 10 to 13 cm in the x and y dimensions, and 21 cm in the z dimension. This high accuracy stems from WiTrack's ability to eliminate errors due to multipath and the combined performance of FMCW and our geometric mapping algorithm. The results also show that even the 90^{th} percentile of the measurements stays within one foot along the x/y-axis and two feet along the z-axis.

In the second application, we consider elderly fall detection. Current solutions to this problem include inertial sensors which old people tend to forget to wear [15], or cameras which infringe on privacy, particularly in bedrooms and bathrooms [20]. In contrast, WiTrack does not require the user to wear any device and protects her privacy much better than a camera. However, simply looking at the change in elevation cannot allow us to distinguish a fall from sitting on the floor. Thus, WiTrack identifies a fall as a *fast* change in the elevation that reaches the ground level. In a population of 11 users and over 133 experiments, WiTrack distinguishes a fall from standing, walking, sitting on a chair and sitting on the floor with an accuracy of 96.9% (the F-measure is 94.34%).

In the third application, we consider a user who desires to control appliances by pointing at them (e.g., the user can turn her monitor on or turn the lights off by simply pointing at these objects.) We consider a gesture in which the user lifts her arm, points at an appliance, and drops her arm. By comparing the position of the arm over time, WiTrack can identify the pointing direction. Our prototype estimates the pointing direction with a median of 11.2 degrees and a 90^{th} percentile of 37.9 degrees.

Our results also show that the prototype operates in realtime, and outputs the 3D location within 75 ms from the time the antennas receive the signal. Further, it operates at a fairly low-power, transmitting only 0.75 milliwatts. However, our current prototype can track a single person, and requires the person to move to obtain an initial estimate of his location.

Contributions: This paper introduces the first device that can achieve centimeter-scale accuracy in tracking the 3D motion of a human based on radio reflections off her body. The paper presents new algorithms for eliminating errors due to multipath and performing accurate 3D tracking, both of a whole body and a body part. The paper also presents a prototype implementation that includes a low-power FMCW radio frontend and realtime processing, delivering accurate 3D motion tracking to within a median of 10 to 20 centimeters.



(a) **Antenna "T" Setup** (b) **FMCW Signal Generation**

Figure 1—WiTrack's Setup and Signal Generation. (a) shows WiTrack's directional antennas (dimension of each antenna: $5cm \times 5cm$) arranged in a "T": the transmit antenna is placed at the crossing point of the T, whereas the receive antennas are on the edges. (b) shows the hardware we built to generate FMCW signals.

Our results demonstrate that WiTrack can expand the space of human-computer interfaces and enable interaction across walls, and occluded spaces. We believe that WiTrack also expands the role that wireless computer networks may play in the future to enable them to provide a variety of services: Communication is definitely a major service, but other services may include motion tracking, through-wall human-computer interaction, and a gesture based interface for controlling appliances and interacting with the environment.

2 WiTrack Overview

WiTrack is a wireless system that performs 3D motion tracking in both line-of-sight and through wall scenarios. It can also provide coarse tracking of body parts, like an arm movement. WiTrack uses multiple directional antennas: one antenna is used for transmitting, and three antennas for receiving. In its default setup, the antennas are arranged in a "T" shape, as shown in Fig. 1(a). In its current version WiTrack tracks one moving body at any time. Other people may be around but should be either behind the antenna beam or they should be approximately static.[1]

WiTrack operates by transmitting an RF signal and capturing its reflections off a human body. It tracks the motion by processing the signals from its received antennas using the following three steps:

1. *Time-of-Flight (TOF) Estimation:* WiTrack first measures the time it takes for its signal to travel from its transmit antenna to the reflecting body, and then back to each of its receive antennas. We call this time the

[1] Small moving objects which do not have significant reflections (e.g., a plastic fan) create some noise but do not prevent WiTrack's 3D tracking.

TOF (time-of-flight). WiTrack obtains an initial measurement of the TOF using FMCW transmission technique; it then cleans this estimate to eliminate multi-path effects and abrupt jumps due to noise.

2. *3D Localization:* Once it obtains the TOF as perceived from each of its receiving antennas, WiTrack leverages the geometric placement of its antennas to localize the moving body in 3D.
3. *Fall Detection and Pointing:* WiTrack builds on the 3D localization primitive to enable new functionalities. Specifically, WiTrack can detect a fall by monitoring fast changes in the elevation of a human and the final elevation after the change. WiTrack can also differentiate an arm motion from a whole body motion; it can track the motion of raising one's arm, localize the initial and final position of the arm, and determine the direction in which the arm is pointing.

3 Time-of-Flight Estimation

The first step for WiTrack is to measure the TOF from its transmit antenna to each of its receive antennas and clean this estimate from the effect of multi-path.

3.1 Obtaining Time-of-Flight Estimates

A straightforward approach for estimating the time of flight is to transmit a very short pulse and measure the delay between the transmitted pulse and its received echo. Such a design requires sampling the signal at sub-nanosecond intervals – i.e, it requires high speed analog-to-digital converters (ADCs) that operate at multi-GS/s. Such ADCs are high power, expensive, and have low bit resolution, making this approach unattractive in practice.

Instead, WiTrack measures the TOF by leveraging a technique called Frequency-Modulated Carrier Waves (FMCW). We explain FMCW at a high level, and refer the reader to [19] for a more detailed explanation. FMCW transmits a narrowband signal (e.g., a few KHz) whose carrier frequency changes linearly with time. To identify the distance from a reflector, FMWC compares the carrier frequency of the reflected signal to that of the transmitted signal. Since the carrier frequency is changing linearly in time, delays in the reflected signals translate into frequency shifts in comparison to the transmitted wave. Therefore, by comparing the frequency difference between the transmitted signal and the received signal, one can discover the time delay that the signal incurred, which corresponds to the TOF of that signal.

Fig. 2 illustrates this concept. The green line is the carrier frequency of the transmitted signal which sweeps linearly with time. The red line is the carrier frequency of the reflected signal as a function of time. The time shift between the two is the time-of-flight (TOF) for that reflector. The frequency shift Δf between the transmitted and received signals is a function of both the slope of the sweep and the TOF, i.e.:

$$TOF = \Delta f / slope \tag{1}$$

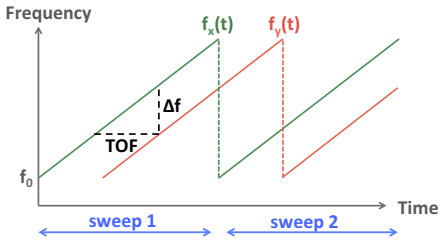


Figure 2—FMCW operation. The transmitted signal has a carrier frequency $f_x(t)$ that is repeatedly swept in time. Because the received signal is time-shifted with respect to the transmitted signal, its carrier frequency $f_y(t)$ is frequency-shifted with respect to $f_x(t)$.

Though the above description is for a single reflector, it can be easily generalized to an environment with many reflectors. In this case, the transmitted signal would still consist of a single carrier wave that is linearly swept in time. However, because wireless reflections add up linearly over the medium, the received signal is a linear combination of multiple reflections, each of them shifted by some Δf that corresponds to its own TOF. Hence one can extract all of these TOFs by taking a fourier transform (i.e, an FFT) of the received baseband signal.[2]

In comparison to transmitting a very short pulse and measuring its sub-nanosecond delay in the time domain, FMCW does not require high speed ADCs because at any point in time, the received baseband signal is narrowband.

FMCW Resolution: It is important to note that the resolution of an FMCW system is a function of the total bandwidth that the carrier frequency sweeps [19]. The resolution is defined by the ability to distinguish between two nearby locations, which depends on the ability to distinguish their TOFs, which itself depends on the resolution in distinguishing frequency shifts Δf. The resolution of identifying frequency shifts is equal to the size of one bin of the FFT. The FFT is typically taken over a duration of one sweep of the carrier frequency (denoted by T_{sweep}) and hence the size of one FFT bin is $1/T_{sweep}$. Since the minimum measurable frequency shift is $\Delta f_{min} = 1/T_{sweep}$, the minimum measurable change in location is:

$$Resolution = C\frac{TOF_{min}}{2} = C\frac{\Delta f_{min}}{2 \times slope}, \tag{2}$$

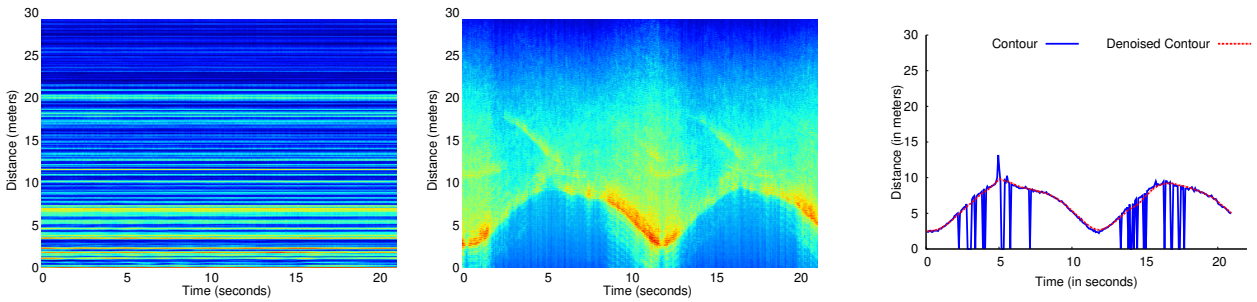
where C is the speed of light and the factor 2 accounts for the fact that the reflected signal traverses the path back and forth.

The *slope*, however, is equal to the total swept bandwidth B divided by the sweep time T_{sweep}. Hence after substituting for the slope in the above equation we get:

$$Resolution = \frac{C}{2B} \tag{3}$$

Since C is very large, obtaining high resolution requires a large B, i.e., the system has to take a narrowband signal

[2]The baseband signal is the received signal after mixing it by the transmitted carrier. The mixing shifts the spectrum of the received signal by the transmitted carrier frequency.



(a) **Spectrogram** (b) **After Background Subtraction** (c) **Contour Tracking**

Figure 3—Obtaining the Time-of-Flight (TOF) Estimates. WiTrack takes an FFT of the received signal in baseband over every sweep period to generate the spectrogram in (a). Then, by subtracting out a given frame from the frame that precedes it, WiTrack eliminates static multipath as in (b). The blue plot in (c) shows how WiTrack can address dynamic multipath by tracking the bottom contour of (b), and then denoise the signal (red plot) to obtain a clean TOF estimate.

and sweep its carrier frequency across a wide bandwidth of multiple GHz.

In our design we chose the following parameter for our FMCW. We have built an FMCW system that sweeps a total bandwidth of 1.69 GHz from 5.56 GHz to 7.25 GHz, and transmits at 0.75 milliwatt. The choice of this bandwidth has been dictated by the FCC regulations for civilian use of spectrum [9]. Specifically, it is the largest contiguous bandwidth below 10 GHz which is available for civilian use at low power.

Based on Eq. 3, our sweep bandwidth allows us to obtain a distance resolution of 8.8 cm. Hence the average error in mapping TOF to distance in 1D is about 4.4 cm. Note that the above derivation neglects the impact of noise, and hence provides a lower bound on the achievable resolution. In practice, the system's resolution is affected by the noise level. It also depends on the geometric model that maps TOFs to 3D locations.

3.2 Addressing Static Multi-path

The next step in WiTrack's operation is to distinguish a human's reflections from reflections off other objects in the environment, like furniture and walls. Recall from the previous section that every reflector in the environment contributes a component to the overall received signal, and that component has a frequency shift that is linearly related to the time-of-flight of the reflection based on Eq. 1. Typically, reflections from walls and furniture are much stronger than reflections from a human, especially if the human is behind a wall. Unless these reflections are removed, they would mask the signal coming from the human and prevent sensing her motion. This behavior is called the "Flash Effect".

To remove reflections from all of these static objects (walls, furniture), we leverage the fact that since these reflectors are static, their distance to the WiTrack device does not change over time, and therefore their induced frequency shift stays constant over time. Fig. 3(a) plots the spectrogram of the received signal as a function of time, for one of the receive antennas of WiTrack. In particular, we take the FFT of the received signal every sweep window, and compute the power in each frequency as a function of time. Note that there is a linear relation between frequency shifts and the traveled distances as follows:

$$distance = C \times TOF = C \times \frac{\Delta f}{slope}. \quad (4)$$

Thus, instead of plotting the power in each frequency as a function of time, we can use the above equation to plot the power reflected from each distance as a function of time, as shown in Fig. 3(a). The color code of the plot corresponds to a heat-map of the power in the reflected signal. Strong reflectors are indicated by red and orange colors, weaker reflectors are indicated by yellow and green, and the absence of a reflector is indicated by blue at the corresponding frequency. The figure indicates the presence of very strong static reflectors in the environment. Specifically, it has many horizontal stripes; each of these stripes signifies the presence of a reflector at the corresponding round-trip distance. Because these stripes are horizontal, their corresponding reflectors are stationary over time. Hence, we eliminate the power from these static reflectors by simply subtracting the output of the FFT in a given sweep from the FFT of the signal in the previous sweep. This process is called background subtraction because it eliminates all the static reflectors in the background.

Fig. 3(b) is the result of applying background subtraction to Fig. 3(a). The figure shows that all static reflectors corresponding to the horizontal lines have been eliminated. This makes it easier to see the much weaker reflections from a moving human. Specifically, we see that the distance of the dominant reflector (the red color signal) is varying with time, indicating that the reflector is moving.

3.3 Addressing Dynamic Multi-path

By eliminating all reflections from static objects, WiTrack is left only with reflections from a moving human (see Fig. 3(b)). These reflections include both signals that bounce off the human body to the receive antennas, and those that bounce off the human then bounce off other

objects in the environment before reaching WiTrack's antennas. We refer to these indirect reflections as dynamic multi-path. It is quite possible that a human reflection that arrives along an indirect path, bouncing off a side wall, is stronger than her direct reflection (which could be severely attenuated after traversing a wall) because the former might be able to avoid occlusion.

Our idea for eliminating dynamic multi-path is based on the observation that, at any point in time, the direct signal reflected from the human to our device has travelled a shorter path than indirect reflections. Because distance is directly related to TOF, and hence to frequency, this means that the direct signal reflected from the human would result in the smallest frequency shift among all strong reflectors after background subtraction.

We can track the reflection that traveled the shortest path by tracing the bottom contour of all strong reflectors in Fig. 3(b). The bottom contour can be defined as the closest local maximum to our device. To determine the first local maximum that is caused by human motion, we must be able to distinguish it from a local maximum due to a noise peak. We achieve this distinguishability by averaging the spectrogram across multiple sweeps. In our implementation, we average over five consecutive sweeps, which together span a duration of 12.5 ms. For all practical purposes, a human can be considered as static over this time duration; therefore, the spectrogram would be consistent over this duration. Averaging allows us to boost the power of a reflection from a human while diluting the peaks that are due to noise. This is because the human reflections are consistent and hence add up coherently, whereas the noise is random and hence adds up incoherently. After averaging, we can determine the first local maximum that is substantially above the noise floor and declare it as the direct path to the moving human.

The blue plot in Fig. 3(c) shows the output of WiTrack's contour tracking of the signal in Fig. 3(b). In practice, this approach has proved to be more robust than tracking the dominant frequency in each sweep of the spectrogram. This is because, unlike the contour which tracks the closest path between a human body and WiTrack's antennas, the point of maximum reflection may abruptly shift due to different indirect paths in the environment or even randomness in the movement of different parts of the human body as a person performs different activities.

3.4 Dealing with Noise

After obtaining the bottom contour of the spectrogram of the signal from each receive antenna, WiTrack leverages common knowledge about human motion to mitigate the effect of noise and improve its tracking accuracy. Specifically, by performing the following optimizations, we obtain the red plot in Fig. 3(c):

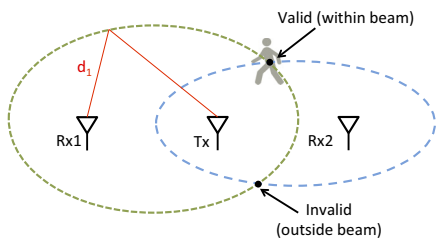
- *Outlier Rejection:* WiTrack rejects impractical jumps in distance estimates that correspond to unnatural human motion over a very short period of time. For example, in Fig. 3(c) , the distance from the reflector (the blue line) repeatedly jumps by more than 5 meters over a span of few milliseconds. Such changes in distance are not possible over such small intervals of time, and hence WiTrack rejects such outliers.
- *Interpolation:* WiTrack uses its tracking history to localize a person when she stops moving. In particular, if a person walks around in a room then sits on a chair and remains static, the background-subtracted signal would not register any strong reflector. In such scenarios, we assume that the person is still in the same position and interpolate the latest location estimate throughout the period during which we do not observe any motion, enabling us to track the location of a subject even after she stops moving.
- *Filtering:* Because human motion is continuous, the variation in a reflector's distance to each receive antenna should stay smooth over time. Thus, WiTrack uses a Kalman Filter to smooth the distance estimates.

4 Localizing in 3D

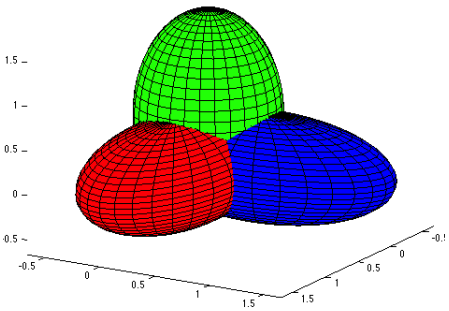
After contour tracking and de-noising of the estimate, WiTrack obtains a clean estimate of the distance travelled by the signal from the transmit antenna to the human reflector, and back to one of the receive antennas. Let us call this estimate the round trip distance. At any time, there are three such round trip distances that correspond to the three receive antennas. The goal of this section is to use these three estimates to identify the 3D position of the human, for each time instance.

To do so, WiTrack leverages its knowledge of the placement of the antennas. Recall that the antennas are placed in a T, as in Fig.1(a) where the y-axis is a horizontal line orthogonal to the plane of the T and the z-axis is along its vertical line. WiTrack uses this reference frame to track the 3D location of a moving target.

Let us focus on identifying the location at a particular time t_i. Also for clarity, let us first assume that we would like to localize the person in the 2D plane defined by the x and y axes. Consider the transmit antenna and the first receive antenna. WiTrack knows the round trip distance from the transmit antenna to the person and back to the first receive antenna. The region of feasible 2D locations for the target need to satisfy this constraint; hence, they fall on the periphery of an ellipse, whose foci are collocated with the Tx and Rx1 antennas and its major axis is equal to the round trip distance. Now consider the second receive antenna. WiTrack knows the round trip distance from the Tx to the person and back to Rx2. Similarly, the feasible solutions to this constraint in 2D are on the periphery of another ellipse whose foci are collocated with the Tx and Rx2 antennas and its major axis is equal to the round trip distance to Rx2. Since the correct location is on both ellipses, it is one of the intersection points, as



(a) **2D Localization**



(b) **3D Localization**

Figure 4—WiTrack's Localization Algorithm. The TOF estimate from a receive antenna defines an ellipse whose foci are the transmit antenna and the receive antenna. (a) shows that WiTrack can uniquely localize a person using the intersection of two ellipses. (b) shows that in 3D, the problem translates into an intersection of three ellipsoids.

shown in Fig. 4(a). In fact, since our antennas are directional, only one of the two intersection points is feasible, which is the one that yields a location in the direction of the antennas beams.

It is straightforward to generalize the argument to localizing in 3D. Specifically, in a 3D space, the round-trip distance defines an ellipsoid whose two foci are the transmit antenna and one of the receive antennas. In this setting, the intersection of two ellipsoids would define an arc in the 3D space, and hence is insufficient to pinpoint the 3D location of a person. However, by adding a third directional antenna, we obtain a unique solution in 3D that is within the beam of all the directional antennas as shown in Fig. 4(b). Therefore, our algorithm can localize a person in 3D by using three directional receive antennas.

Finally we note two points:

- The T-shape placement for the antennas is chosen because we assume the user wants to localize motion behind a wall, in which case all the antennas would have to be arranged in one plane facing the wall. We place one antenna below to help determine elevation, while the others are on the same level.
- While the minimum number of Rx antennas necessary to resolve a 3D location is three, adding more antennas would result in more constraints. This would allow us to over-constrain the solution and hence add extra robustness to noise.

5 Beyond 3D Tracking

In this section, we build on WiTrack's 3D localization primitive to enable two additional capabilities: estimating a pointing direction from the corresponding arm movement, and detecting a fall.

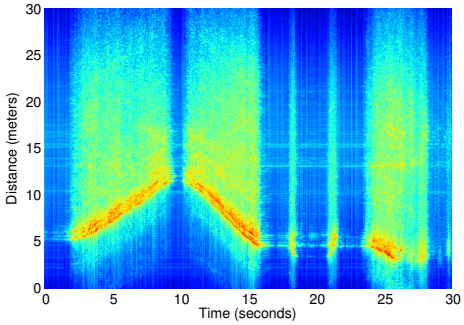


Figure 5—Gestures. The figure shows a human moving then stopping and pointing with her arm. The small bright regions around $t = 18$s and $t = 21$s correspond to the arm lifting and dropping motions.

5.1 Estimation of Pointing Angle

We explain how WiTrack provides coarse estimation of body part motion. We consider the following motion: the user starts from a state where her arm is rested next to her body. She raises the arm in a direction of her choice with the intention of pointing toward a device or appliance, and then drops her hand to the first position. The user may move around and at a random time perform the pointing gesture. We require, however, that the user be standing (i.e., not walking) when performing the pointing gesture. The goal is to detect the pointing direction.

To track such a pointing gesture, WiTrack needs to distinguish between the movement of the entire body and the motion of an arm. To achieve this goal, we leverage the fact that the reflection surface of an arm is much smaller than the reflection surface of an entire human body. We estimate the size of the reflection surface from the spectrogram of the received signal at each of the antennas. Fig. 5 illustrates the difference between the spectrogram of a whole body motion and that of an arm pointing, as captured by one of WiTrack's receiving antennas. In the figure the human was moving then stopped and performed the pointing gesture. The two bright spots around $t = 18$s and $t = 21$s refer to the arm being lifted and dropped respectively. The figure shows that the signal variance along the vertical axis is significantly larger when the reflector is the entire human body than when it is just an arm motion (note the bright yellow as opposed to the cyan color). If the reflector is large, its parts have slightly different positions from each other; hence, at any point in time the variance of its reflection along the y-axis is larger than that of an arm movement. WiTrack uses this spatial variance to detect body part motion from a whole body motion.

Once we detect it is a body part, WiTrack tries to estimate the direction of the motion to identify the pointing direction, which involves the following steps:

1. *Segmentation:* The goal of segmentation is to determine the start and end of a pointing gesture. Fig. 5 shows how WiTrack segments the round trip distance spectrogram obtained from each receive antenna. In

our pointing experiments, we ask the user to remain static for a second before performing the pointing gesture. Thus, we are able to detect the start of a pointing gesture since it is always preceded by a period of absence of motion. Similarly, after a person raises her arm in a pointing direction, we ask her to wait for a second before resting her arm back to its initial position. Because WiTrack performs a frequency sweep every 2.5 ms, we can easily distinguish the silence at the start and end of a gesture.

2. *Denoising:* As is the case for a whole body motion, the contour of the segmented spectrogram is denoised and interpolated (see §3.4) to obtain a clean estimate of the round trip distance of the arm motion as a function of time, for each receive antenna.
3. *Determining the Pointing direction:* We perform robust regression on the location estimates of the moving hand, and we use the start and end points of the regression from all of the antennas to solve for the initial and final position of the hand. WiTrack estimates the direction of pointing as the direction from the initial state to the final extended state of the hand. Since the user drops her hand after pointing, WiTrack repeats the above steps for this drop motion obtaining a second estimate of the pointing direction. Then, WiTrack estimates the pointing direction as the middle direction between the two.[3] Being able to leverage the approximate mirroring effect between the arm lifting and arm dropping motions adds significant robustness to the estimation of the pointing angle.

We envision that an application of the estimation of pointing direction can be to enable a user to control household appliances by simply pointing at them. Given a list of instrumented devices and their locations, WiTrack would track the user's hand motion, determine the direction in which she points, and command the device to change its mode (e.g., turn on or off the lights, or control our blinds).

Finally, to demonstrate the pointing gesture within the context of an application, we created a setup where the user can control the operation mode of a device or appliance by pointing at it. Based on the current 3D position of the user and the direction of her hand, WiTrack automatically identifies the desired appliance from a small set of appliances that we instrumented (lamp, computer screen, automatic shades). Our instrumentation is a basic mode change (turn on or turn off). WiTrack issues a command via Insteon home drivers [2] to control the devices. We envision that this setup can evolve to support a larger set of functionalities and be integrated within a home automation systems [16].

5.2 Fall Detection

Our objective is to automatically distinguish a fall from other activities including sitting on the ground, sitting on a chair and walking. To do so, we build on WiTrack's elevation tracking along the z dimension. Note that simply checking the person's elevation is not sufficient to distinguish falls from sitting on the floor. To detect a fall, WiTrack requires two conditions to be met: First, the person's elevation along the z axis must change significantly (by more than one third of its value), and the final value for her elevation must be close to the ground level. The second condition is the change in elevation has to occur within a very short period to reflect that people fall quicker than they sit.

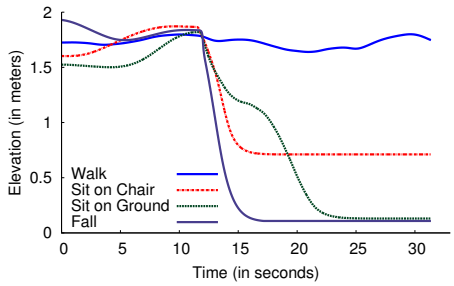


Figure 6—Fall Detection. WiTrack automatically detects falls by monitoring the absolute value and the change in elevation.

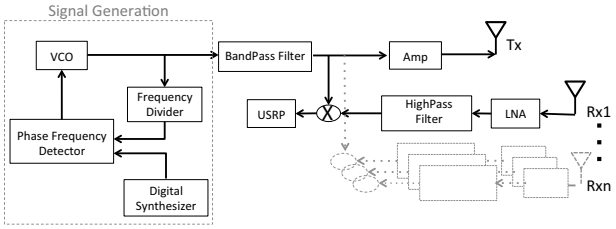


Figure 7—Schematic of the Front End Design. WiTrack's front end consists of an FMCW signal generation component, and a receive chain that is connected to a USRP.

Fig. 6 plots WiTrack's estimate of the elevation along the z dimension for four activities: a person walking, sitting on a chair, sitting on the ground, and (simulated) falling on the ground.[4] The figure confirms that walking and sitting on a chair can be identified from falling and sitting on the floor based on elevation because the final elevation is far from $z = 0$. However, to distinguish a fall on the ground from a sitting on the ground, one has to exploit that during a fall the person changes her elevation faster than when she voluntarily sits on the floor.

6 Implementation

FMCW Radio Front-End Hardware: We have built an FMCW front-end that operates as a daughterboard for the USRP software radio [5]. Below, we describe our design, which is illustrated in the schematic of Fig. 7.

The first step of our front end design is the generation of an FMCW signal, which consists of a narrowband signal whose carrier frequency is linearly swept over a large bandwidth. This signal can be obtained by using

[3] by zooming on Fig. 5 the reader can see how the arm lifting and dropping motions approximately mirror each other's tilt.

[4] The fall was performed in a padded room as detailed in §8.5.

a voltage-controlled oscillator (VCO). Because the output frequency of a VCO is a linear function of its input voltage, we can generate our desired frequency sweep by feeding a voltage sweep as an input to the VCO. However, small errors in the input voltage can create large nonlinearities in the output sweep.

To obtain a highly linear sweep, we use a feedback mechanism. Specifically, we use a phase frequency detector to compare the output frequency of the VCO with a highly accurate reference signal, and use the offset between the two to control the VCO. Note that even though the reference signal needs to be highly accurate, it does not need to span the same bandwidth as our desired output signal. In particular, rather than directly comparing the output of the VCO to the reference signal, we first use a frequency divider. This allows us to use a reference signal that sweeps from 136.5–181.25 MHz to generate an FMCW signal that sweeps from 5.46–7.25 GHz. This FMCW signal is transmitted over the air using WA5VJB directional antennas [7] after filtering and amplification.

At the receive chain, the transmitted signal is captured using WA5VJB directional antennas and passed through a low-noise amplifier and a high-pass filter to improve its SNR. Recall from §3 that an FMCW receiver determines the TOF by measuring the frequency offset between the transmitted and the received signal. This offset can be obtained by downconverting (mixing) the received signal with the transmitted signal. The output of the mixer is then fed to the LFRX-LF daughterboard on USRP2 which samples it at 1 MHz and passes the digitized samples to the UHD driver.

Real-time Software Processing: The implemented prototype performs real-time 3D motion tracking as described in §3, §4 and §5. Tracking is implemented directly in the UHD driver of the USRP software radio. The signal from each receiving antenna is transformed to the Frequency domain using an FFT whose size matches the FMCW sweep period of 2.5ms. To improve resilience to noise, every five consecutive sweeps are averaged creating one FFT frame. Background subtraction is performed by subtracting the averaged FFT frame from the frame that precedes it. The spectrogram is processed for contour tracking by identifying for each time instance the smallest local frequency maximum that is significantly higher than the noise level. Outlier rejection is performed by declaring that the contour should not jump significantly between two successive FFT frames (because a person cannot move much in 12.5ms). The output is smoothed with a Kalman filter.

To locate a person, instead of solving a system of ellipsoid equations in real-time, we leverage that the location of the antennas does not change and is known a priori. Thus, before running our experiments, we use MATLAB's symbolic library to find a symbolic representation of the solutions (x, y, z) as a function of symbolic TOF to each of the receiving antennas. This means that the ellipsoid equations need to be solved only once (for any fixed antenna positioning), independent of the location of the tracked person. After it obtains the 3D location of a person, WiTrack uses python's matplotlib library to output this location in real-time.

Software processing has a total delay less than 75 ms between when the signal is received an a corresponding 3D location is output.

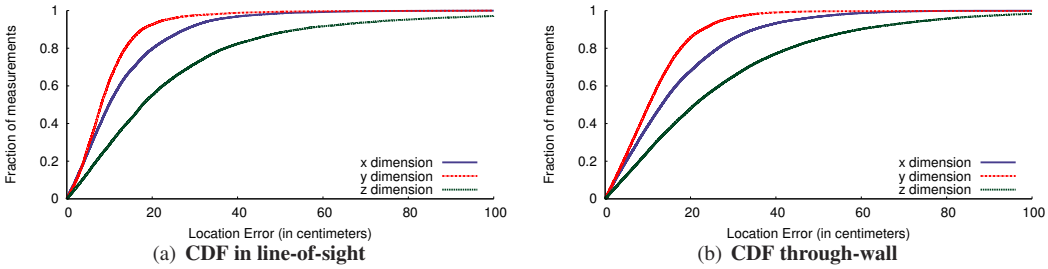
7 Evaluation

We empirically evaluate the performance of the WiTrack prototype by conducting experiments in our lab building with 11 human users.

(a) Ground Truth: We determine WiTrack's localization accuracy by testing it against the VICON motion capture system. The VICON is a multi-hundred-thousand dollar system used in filmmaking and video game development to track the human motion and map it to a 3D character animation model. It uses calibrated infrared cameras and records motion by instrumenting the tracked body with infrared-reflective markers. The VICON system has a sub-centimeter accuracy and hence we use it to determine the ground truth location. To track a moving person with the VICON, she is asked to wear a jacket and a hat, which are instrumented with eight infrared markers. To track a subject's hand, she is asked to wear a glove that is also instrumented with six markers. The VICON tracks the infrared markers on the subject's body and fits them to a 3D human model to identify the subject's location.

The VICON system has a built-in capability that can track the center of any object using the infrared-reflective markers that are placed on that object. This allows us to determine the center position of a human subject who is wearing the instrumented jacket and hat. WiTrack however computes the 3D location of the body surface where the signal reflects. In order to compare WiTrack's measurements to those obtained by the VICON, we need to have an estimate of the depth of the center with respect to the body surface. Thus, we use the VICON to run offline measurements with the person standing and having infrared markers around her body at the same height as the WiTrack transmit antenna (about the waist). We use the VICON to measure the average depth of the center from surface for each person. To compare the 3D location computed by the two systems, we first compensate for the average distance between the center and surface for that person and then take the Euclidean distance.

(b) Device Setup: WiTrack is placed behind the wall of the VICON room. The device uses one transmit antenna and three receive antennas. The transmit antenna and two receive antennas are lined up parallel to the wall, and a third receive antenna is placed below the transmit antenna. The distance between the transmit antenna and each re-



(a) **CDF in line-of-sight** (b) **CDF through-wall**

Figure 8—Performance of WiTrack's 3D Tracking. (a) and (b) show the CDF of the location error for WiTrack in line-of-sight and through-wall scenarios respectively.

ceive antenna is 1m, unless otherwise noted.

(c) Human Subjects: The experiments are performed with eleven human subjects: two females and nine males. The subjects are of different heights and builds, and span an age range of 22 to 56 years. In each experiment, the subject is asked to move at will in the VICON room; he/she is tracked using both the VICON system and WiTrack. Note that WiTrack tracks the subject through the wall, from an adjacent room, while the VICON has to be within direct line of sight from the subject.

8 Performance Results

8.1 Accuracy of 3D Tracking

We first focus on the developed 3D tracking primitive and evaluate its accuracy across all three dimensions.

We run 100 experiments each lasting for 1 minute, during which a human subject moves at will in the VICON room. The VICON room has no windows. It has 6-inch hollow walls supported by steel frames with sheet rock on top, which is a standard setup for office buildings. The WiTrack prototype is placed outside the room with all transmit and receive antennas facing one of the walls of the VICON room. Recall that WiTrack's antennas are directional; hence, this setting means that the radio beam is directed toward the wall of the VICON room. In each experiment, we ask the subject to wear the jacket and hat that were instrumented with VICON markers and move inside the VICON-instrumented room. The subject's location is tacked by both the VICON system and WiTrack.

We note that the VICON IR cameras are set to accurately track the target only when she moves in a 6×5 m^2 area in the room. Their accuracy degrades outside that area. Since VICON provides the ground truth in our experiment, we ask the target to stay within the 6×5 m^2 area where the IR cameras are focused. This area is about 2.5m away from the wall. Thus, the minimum separation between WiTrack and the human subject in these experiments is 3 m and the maximum separation is about 9 m.

We perform a total of 100 experiments for this evaluation, each lasting for one minute. Since each FMCW sweep lasts for 2.5ms and we average 5 sweeps to obtain for each TOF measurement, we collect a total of about 480,000 location readings from these 100 experiments.

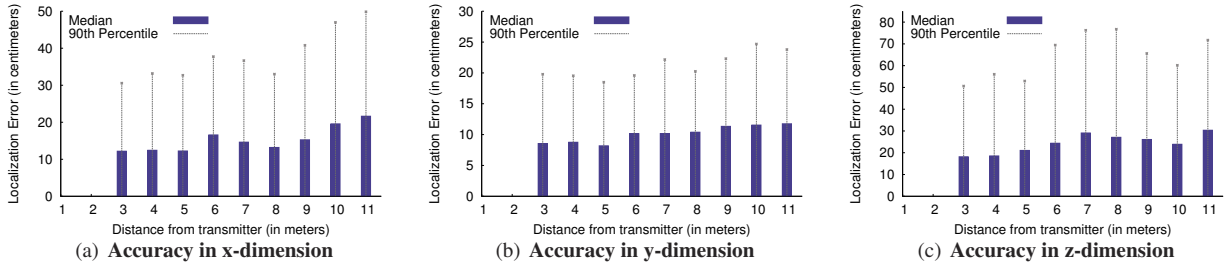
To show that WiTrack works correctly both in line of sight and through a wall, we repeat the above 100 experiments with one modification, namely we move the WiTrack device inside the room and set it next to the wall from the inside.

Fig. 8(a) and Fig. 8(b) plot the CDFs of the location error along the x, y, and z coordinates. The figure reveals the following findings:

- WiTrack's median location error for the line-of-sight experiments is 9.9 cm, 8.6 cm, and 17.7 cm along the x, y, and z dimensions respectively. In comparison, the median location error in the through-wall experiments is 13.1 cm, 10.25 cm, and 21.0 cm along the x, y, and z dimensions. As expected the location accuracy in line-of-sight is higher than when the device is behind a wall due to the extra attenuation and the reduced SNR. In both cases, however, the median error is fairly small. This is due to the use of an FMCW radio which ensures a highly accurate TOF estimate, and the ability to prevent errors due to multipath and noise, allowing the system to stay accurate as it moves from TOF to a 3D location estimate of the human body.
- Interestingly, the accuracy in the y dimension is better than the accuracy in the x dimension. This difference is because the x and y dimensions are not equal from the perspective of WiTrack's antennas. Recall that in the xy-plane, WiTrack's antennas are all along the x-axis. As a result, the two ellipses in the xy-plane, shown in Fig. 8, both have their major radius along x and minor radius along y. Hence, the same error in TOF produces a bigger component when projected along the x axis than along the y axis.
- The accuracy along the z-dimension is worse than the accuracy along the x and y dimensions. This is the result of the human body being larger along the z dimension than along x or y.

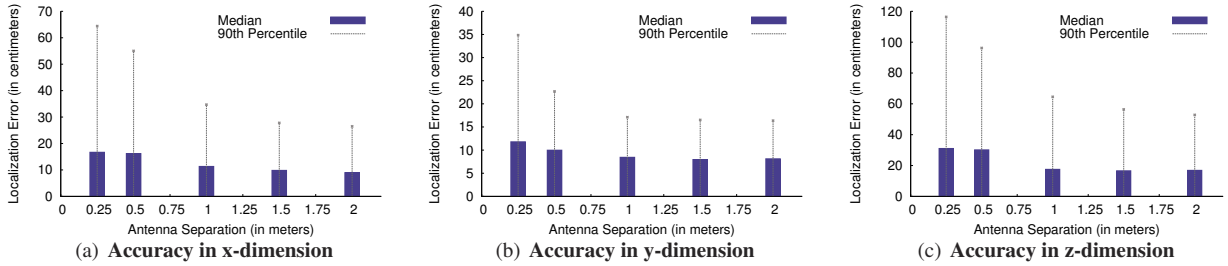
8.2 Accuracy Versus Distance

We are interested in evaluating WiTrack's accuracy as the person gets further away from the device. Thus, we repeat the above through-wall experiments. As mentioned above, VICON requires the human to move in a certain space that is in line of sight of the IR cameras. Thus, to increase the distance from WiTrack to the human we move WiTrack away in the hallway next to the VICON



(a) **Accuracy in x-dimension** (b) **Accuracy in y-dimension** (c) **Accuracy in z-dimension**

Figure 9—3D Localization Accuracy Versus Distance to Device. (a)-(c) show the location error along the x, y, and z dimensions as a function of how far the subject is from WiTrack. The median and 90th percentile errors increase as the distance from the device to the person increases.



(a) **Accuracy in x-dimension** (b) **Accuracy in y-dimension** (c) **Accuracy in z-dimension**

Figure 10—3D Localization Accuracy Versus Size of Device. (a)-(c) show the median and 90th percentile location errors as a function of the antenna separation. Along all three dimensions, a larger separation leads to a decrease in the location error.

room. Again, we collect 100 experiments, each spanning one minute for a total of 480,000 location measurements.

Fig. 9 plots WiTrack's localization error as a function of its distance to the subject. The distance to the subject is determined using the VICON ground-truth coordinates, and rounded to the nearest meter. The figure shows the median and 90th percentile of the estimation error for the x, y, and z coordinates.

The figure shows that the median accuracy changes by 5 to 10 cm for distances that are 3 to 11 m away from the device. As expected, the further the human moves from the device, the larger the estimation error. This increase in error with distance is expected since as the distance gets larger the signal gets more attenuated. However, a second reason stems from the geometry of the ellipsoid-based localization model. Given the equations of the ellipsoid, the TOF multiplied by the speed of light is equal to the major axis of the ellipsoid/ellipse that describes the user's location, and the antenna separation is the distance between the foci. For a fixed antenna separation, as the distance/TOF increases the ellipsoid's surface increases, increasing the overall space of potential locations.

The figure also shows that the accuracy is best along the y dimension, then the x, and finally the z, which is due to the reasons discussed in the previous section.

8.3 Accuracy Versus Antenna Separation

Our default setting places the receive antennas 1 m away from the transmit antenna. In this section, we examine the impact of antenna separation on performance.

We evaluate five different configurations. In all of these configurations, the transmit antenna is at an equal distance from all receive antennas, and is placed at the crossing point of a "T" whereas the receive antennas are placed at the edges. We vary the distance between the transmit antenna and each of the receive antennas from 25 cm to 2 m. We run 100 one-minute experiments, 20 for each antenna setting. All experiments are run through a wall. In each experiment, we ask the human subject to move at will inside the VICON room, as we record her location using both the VICON system and WiTrack.

Fig. 10 shows WiTrack's localization accuracy as a function of antenna separation. The figure shows that even if one brings the antennas to within 25cm of each other, the median location error stays less than 17 cm, 12 cm, and 31 cm for the x, y, and z dimensions, and 90th percentile becomes 64cm, 35cm, and 116cm respectively. While this is higher than the previous results where the antennas were separated by 1 m, it is still comparable to state of the art localization using a WiFi transmitter (in our case, the user does not need to carry any wireless device).

The plots show that as the antenna separation increases, the localization accuracy improves along all three dimensions x, y, and z. This behavior is expected, because the further the receive antennas are from each other, the larger the spatial diversity between them. Because of the geometric nature of the algorithm, a spatially diverse setup would lead to a smaller intersection curve between any pair of ellipsoids.[5] For this reason, in a larger setup, the

[5]For simplicity, consider the 2D case with 1 Tx and 2 Rx antennas. Because of the system's resolution, each ellipse has some fuzzy region about it (i.e., a thickness of $+/\varepsilon$, where ε is determined by the resolution). Thus, the intersection of two ellipses is a region rather than a single point. This region becomes larger when the Rx antennas are closer to each other, and the larger the region, the larger the ambiguity in localization. In the extreme case where the two receive antennas are co-located,

same noise variance in the TOF estimates would be confined to a smaller curve, thus, minimizing estimate error.

Mathematically, for any TOF, the antenna separation is the distance between the foci of the ellipsoid that defines the person's location. Hence for any given TOF, increasing the antenna separation increases the distance between the foci while keeping the ellipsoid's major radius constant. Hence the ellipsoid gets more squashed and its circumference becomes smaller, reducing the region of potential solutions.

8.4 Accuracy of Estimating Pointing Direction

In the experiments in this section, the human subjects wear a glove that is instrumented with infrared reflexive markers, and are asked to stand in a given location inside the VICON room and point in a direction of their choice. Each pointing gesture consists of raising the subject's hand in the direction of her choice, followed by the subject returning her hand to its original resting position. Across our experiments, we ask the human subjects to stand in random different locations in the VICON room and perform the pointing gesture. We determine the direction in which the subject pointed by using both the VICON recordings and WiTrack's estimates (see §5.1).

Fig. 11 plots a CDF of the error between the angle as determined by WiTrack and the ground truth angle based on the VICON measurements. The figure shows that the median orientation error is 11.2 degrees, and the 90^{th} percentile is 37.9 degrees. These results suggest that WiTrack provides an enabling primitive to track pointing gestures. We used this capability to enable controlling different household appliances like shades, lamps and computer screens by sending commands to these different appliances over Insteon drivers.

8.5 Fall Detection

We test the fall detection algorithm described in §5.2 by asking different participants to perform four different activities: walk, sit on a chair, sit on the floor, and simulate a fall. The floor of the VICON room is already padded. We add extra padding to ensure no injury can be caused by simulated falls. We perform 132 experiments in total, 33 for each activity. We log the data files from each of these experiments and process them offline with our fall detection algorithm. We obtain the following results:

- None of the walking or sitting on a chair activities are classified as falls.
- One of the sitting on the floor experiments was classified as a fall.
- Two out of 33 simulated falls were not detected (they were misclassified as sitting on the ground).

Thus, the precision of the fall detection algorithm is 96.9% (since out of the 32 detected falls only 31 are true falls) , and the recall is 93.9% (since out of 33 true falls we detected 31). This yields an F-measure of 94.4%.

the two ellipses perfectly overlap and the ambiguity region is large.

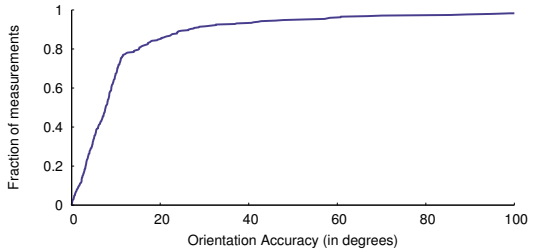


Figure 11—Orientation Accuracy. The CDF of the orientation accuracy shows that the median orientation error is 11.2 degrees, and the 90^{th} percentile error is 37.9 degrees.

9 Related Work

Indoor wireless localization: WiTrack builds on recent advances in RF-based localization [31, 18, 28, 11]. These systems localize a wireless device using RSSI [11, 22], fine-grained-OFDM channel information [25], antenna arrays [31, 18], or RFID backscatter [28, 27]. In contrast, WiTrack localizes a human using body radio reflections.

Some past works in radio tomography use a network of tens or hundred sensors to track a person even if she does not carry any wireless device [29, 30]. These works measure the RSSI for each of the resulting n^2 links between their sensors, and attribute the variation of RSSI on a link to a human crossing that link. Other works on device-free localization rely on RSSI fingerprints [32, 24], which are generated in a training phase by asking a person to stand in different locations throughout the area of interest. In the testing phase, they localize a person by mapping the resulting RSSI to the closest fingerprint. While WiTrack shares the objective of tracking a person's motion without instrumenting her body, it differs in both technology and accuracy. Specifically, WiTrack does not require prior training and uses a few antennas that generate FMCW signals and measure the time-of-flight of the signal reflections to infer location of a human. Its technique extends to 3D, and its 2D accuracy is more than 5× higher than the state of the art RSSI-based systems [33, 24].

See through-wall & gesture recognition using WiFi: WiTrack is motivated by recent research that used WiFi signals to detect users through walls and identify some of their gestures [10, 21, 13]. Similar to these systems, WiTrack captures and interprets radio reflections off a human body. WiTrack, however, differs from these systems both in capability and technology. Specifically, these systems rely on the Doppler shift of WiFi signals. Hence, they can distinguish only between getting closer or getting further away, but cannot identify the location of the person.[6] In contrast, WiTrack measures the time of flight and, hence, can identify the exact location of a person. Among these past systems, WiVi [10] focuses on track-

[6]The gestures recognized by WiVi and WiSee are sequences of getting closer or getting further away, which translate into positive and negative Doppler shifts. The work in [13] provides a distance estimate with an accuracy of about 30 meters.

ing through dense walls such as concrete by leveraging interference nulling to eliminate the wall's reflection. In contrast, WiTrack focuses on accurate 3D motion tracking that operates through *interior* walls (which are less dense than concrete)[7], pinpointing the exact location of a user at any point in time.

FMCW Radar: WiTrack builds on past work on FMCW radar, including work that used FMCW for see-through-wall that is targeted for the military [23, 12]. WiTrack however differs along multiple dimensions. First, FMCW radios in past work were high-power and heavy (needed to be mounted on a truck). Their tracking capabilities hinge on using large antenna arrays that can achieve a narrow beam, which enables tracking a moving target. In contrast, we present a light weight, low-power FMCW radio that complies with the FCC regulations for consumer devices. We are able to perform accurate tracking with a low-power, relatively cheap FMCW prototype because of two innovations: first, a geometric localization algorithm that combines multiple measurements from different antenna locations and fits them within a geometric reference to pinpoint an accurate 3D location, and second, novel techniques that enable rejecting errors that are due to both static and dynamic multi-path in indoor environments. Further, WiTrack extends its techniques to tracking the motion of body parts, e.g., tracking a hand as it points in a particular direction.

Motion tracking in user interfaces: Finally, WiTrack is related to an emerging body of motion-tracking user interfaces. These include devices that the person needs to hold (such as the Nintendo Wii [4]) or wear (e.g., on-body sensors such as wristbands [1, 14, 17]). They also include vision and infrared-based systems, like Xbox Kinect [8] and Leap Motion [3], which can track a person's movement without requiring her to hold or wear any transmitter or receiver but require the user to maintain a line-of-sight path to their sensors. Similar to these systems, WiTrack enables more natural human-computer interaction. However, in comparison to these systems, WiTrack does not require the user to hold/wear any device or to maintain a line-of-sight path to its sensors; it can track a user and her gestures in non-line-of-sight and across different rooms.

10 Limitations & Conclusion

3D motion tracking based purely on RF reflections off a human body is a challenging technical problem. We believe WiTrack has taken an important step toward addressing this problem. However, the current version of WiTrack still has limitations:

Tracking one person: Our current design can track only one person at any point in time. This does not mean that WiTrack requires only one person to be present in the environment. Other people can be around, but they have to be behind the directional antennas. We believe that this limitation is not fundamental to the design of WiTrack and can be addressed as the research evolves. Consider for example, the case of two moving humans. In this case, each antenna has to identify two concurrent TOFs (one for each person), and hence two ellipsoids. To eliminate the ambiguity, one may use more antennas which add more constraints to the system.

Requiring motion: A second limitation stems from the fact that WiTrack needs the user to move in order to locate her. This is because WiTrack receives reflections from all static objects in the environment; hence, it cannot distinguish the static user from a piece of furniture. To eliminate these static reflectors, WiTrack subtracts consecutive FMCW sweeps. Unfortunately, that eliminates the reflections of the static user as well. Future research may address this issue by having WiTrack go through a training period where the device is first presented with the space without any user so that it may learn the TOFs of the static objects. Naturally, this would require retraining every time the static objects are moved in the environment.

Distinguishing between body parts: Currently WiTrack can provide coarse tracking of the motion of one body part. The tracked part has to be relatively large like an arm or a leg. WiTrack however does not know which body part has moved, e.g., it cannot tell whether it is an arm or a leg. In our experiments, the users were pointing with their arms. Extending this basic capability to tracking more general movements of body parts will likely require incorporating complex models of human motion. In particular, Kinect's ability to track body parts is the result of the combination of 3D motion tracking using infrared with complex vision algorithms and advanced models of human motion [26]. An interesting venue for research is to investigate how WiTrack may be combined with these techniques to produce a highly accurate motion tracking system that operates across walls and occlusions.

While there is scope for many improvements, we believe WiTrack advances the state of the art in 3D motion tracking by enabling through wall operation without requiring any instrumentation of the user body. Furthermore, its fall detection and pointing estimation primitives enable innovative applications.

Acknowledgments: We thank Jue Wang, Haitham Hassanieh, Ezz Hamad, Deepak Vasisht, Mary McDavitt, Diego Cifuentes, Swarun Kumar, Omid Abari, and Jouya Jadidian for participating in our experiments. We also thank Nate Kushman, Lixin Shi, the reviewers, and our shepherd, Kyle Jamieson, for their insightful comments. This research is supported by NSF. We thank members of the MIT Center for Wireless Networks and Mobile Computing: Amazon.com, Cisco, Google, Intel, Mediatek, Microsoft, ST Microelectronics, and Telefonica for their interest and support.

[7] To enable WiTrack to track through thicker walls such as concrete (as in WiVi), one may add a filter to remove the wall's reflection.

REFERENCES

[1] Fitbit Flex. **http://www.fitbit.com/flex**.

[2] Insteon ApplianceLinc. **http://www.insteon.com**. Insteon.

[3] Leap Motion. **https://www.leapmotion.com**.

[4] Nintendo Wii. **http://www.nintendo.com/wii**.

[5] USRP N210. **http://www.ettus.com**. Ettus Inc.

[6] VICON T-Series. **http://www.vicon.com**.

[7] WA5VJB antenna. **http://www.wa5vjb.com**. Kent Electronics.

[8] X-box Kinect. **http://www.xbox.com**. Microsoft.

[9] *Understanding the FCC Regulations for Low-power, Non-licensed Transmitters*. Office of Engineering and Technology Federal Communications Commission, 1993.

[10] F. Adib and D. Katabi. See through walls with Wi-Fi! In *ACM SIGCOMM*, 2013.

[11] P. Bahl and V. Padmanabhan. RADAR: an in-building RF-based user location and tracking system. In *IEEE INFOCOM*, 2000.

[12] G. Charvat, L. Kempel, E. Rothwell, C. Coleman, and E. Mokole. A through-dielectric radar imaging system. *IEEE Trans. Antennas and Propagation*, 2010.

[13] K. Chetty, G. Smith, and K. Woodbridge. Through-the-wall sensing of personnel using passive bistatic wifi radar at standoff distances. *IEEE Trans. Geoscience and Remote Sensing*, 2012.

[14] G. Cohn, D. Morris, S. Patel, and D. Tan. Humantenna: using the body as an antenna for real-time whole-body interaction. In *ACM CHI*, 2012.

[15] J. Dai, X. Bai, Z. Yang, Z. Shen, and D. Xuan. Perfalld: A pervasive fall detection system using mobile phones. In *IEEE PERCOM*, 2010.

[16] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and V. Bahl. An operating system for the home. In *Usenix NSDI*, 2012.

[17] C. Harrison, D. Tan, and D. Morris. Skinput: appropriating the body as an input surface. In *ACM CHI*, 2010.

[18] K. Joshi, S. Hong, and S. Katti. Pinpoint: Localizing interfering radios. In *Usenix NSDI*, 2013.

[19] B. R. Mahafza. *Radar systems analysis and design using MATLAB*. Chapman & Hall, 2013.

[20] N. Noury, A. Fleury, P. Rumeau, A. Bourke, G. Laighin, V. Rialle, and J. Lundy. Fall detection-principles and methods. In *IEEE EBMS*, 2007.

[21] Q. Pu, S. Jiang, S. Gollakota, and S. Patel. Whole-home gesture recognition using wireless signals. In *ACM MobiCom*, 2013.

[22] A. Rai, K. K. Chintalapudi, V. N. Padmanabhan, and R. Sen. Zee: zero-effort crowdsourcing for indoor localization. In *ACM MobiCom*, 2012.

[23] T. Ralston, G. Charvat, and J. Peabody. Real-time through-wall imaging using an ultrawideband multiple-input multiple-output (MIMO) phased array radar system. In *IEEE ARRAY*, 2010.

[24] M. Seifeldin, A. Saeed, A. Kosba, A. El-Keyi, and M. Youssef. Nuzzer: A large-scale device-free passive localization system for wireless environments. *IEEE Transactions on Mobile Computing*, 2013.

[25] S. Sen, B. Radunovic, R. R. Choudhury, and T. Minka. Spot localization using phy layer information. In *ACM MobiSys*, 2012.

[26] J. Shotton, T. Sharp, A. Kipman, A. Fitzgibbon, M. Finocchio, A. Blake, M. Cook, and R. Moore. Real-time human pose recognition in parts from single depth images. *Communications of the ACM*, 2013.

[27] J. Wang, F. Adib, R. Knepper, D. Katabi, and D. Rus. RF-Compass: Robot Object Manipulation Using RFIDs. In *ACM MobiCom*, 2013.

[28] J. Wang and D. Katabi. Dude, where's my card? RFID positioning that works with multipath and non-line of sight. In *ACM SIGCOMM*, 2013.

[29] J. Wilson and N. Patwari. Radio tomographic imaging with wireless networks. In *IEEE Transactions on Mobile Computing*, 2010.

[30] J. Wilson and N. Patwari. See-through walls: Motion tracking using variance-based radio tomography networks. In *IEEE Transactions on Mobile Computing*, 2011.

[31] J. Xiong and K. Jamieson. ArrayTrack: a fine-grained indoor location system. In *Usenix NSDI*, 2013.

[32] M. Youssef, M. Mah, and A. Agrawala. Challenges: device-free passive localization for wireless environments. In *ACM MobiCom*, 2007.

[33] Y. Zhao, N. Patwari, J. M. Phillips, and S. Venkatasubramanian. Radio tomographic imaging and tracking of stationary and moving people via kernel distance. In *ACM ISPN*, 2013.

Epsilon: A Visible Light Based Positioning System

Liqun Li[1], Pan Hu[3], Chunyi Peng[2], Guobin Shen[1], Feng Zhao[1]
[1]*Microsoft Research, Beijing, China*
[2]*Department of Computer Science and Engineering, Ohio State University*
[3]*School of Computer Science, University of Massachusetts, Amherst*

Abstract

Exploiting the increasingly wide use of Light-emitting Diode (LED) lighting, in this paper, we study the problem of using visible LED lights for *accurate* localization. The basic idea is to leverage the existing lighting infrastructure and apply trilateration to localize any devices with light sensing capability (e.g., a smartphone), using LED lamps as anchors. Through the design of *Epsilon*, we identify and tackle several technique challenges. In particular, we establish and experimentally verify the optical channel model for localization. We adopt BFSK and channel hopping to enable reliable location beaconing from multiple, uncoordinated light sources over the shared optical medium. We handle realistic situations towards robust localization, for example, we exploit user involvement to resolve the ambiguity in case of insufficient LED anchors. We have implemented the Epsilon system and evaluated it with a small scale hardware testbed as well as moderate-size simulations. Experimental results confirmed the effectiveness of Epsilon: the 90th percentile accuracies are 0.4m, 0.7m and 0.8m for three typical office environments. Even in the extreme situation with a single light, the 90th percentile accuracy is 1.1m. We believe that visible light based localization is promising to significantly improve the positioning accuracy, despite few open problems in practice.

1 Introduction

We have been witnessing ever increasing roll-out of location-based services, for which accurate location provisioning is a key. GPS has largely solved the problem for outdoor scenarios. However, accurate localization remains a grand challenge for indoor environments. WiFi-based indoor localization has attracted lots of research attentions, for the advantage of ease-use and low deployment cost by leveraging existing WiFi infrastructure [3, 6, 23]. However, they usually deliver an accuracy of up to few meters (refer to §8.4), suffering from wireless channel dynamics, fading, interference and environmental noises.

In this paper, we propose *Epsilon*, a novel sub-meter localization system exploiting visible Light-emitting Diode (LED) lighting infrastructure. Such work is inspired by two observations. The first is the ever increasingly widespread of LED lighting [14]. LED offers a new and revolutionary lighting technology with the potential for longer lifetime, energy saving, quality improvement, and environment preservation. The second is its unique dual-paradigm feature, i.e., illumination as well as communication. It is attributed to the LED's ability of instantaneous on/off, which allows LEDs to be dimmed via Pulse Width Modulation (PWM) and thus to carry digital information in the visible light carrier, i.e., visible light communication (VLC) [9, 11, 16].

Inspired by these favorable facts, Epsilon is designed to *provide high-accuracy positioning in a low(zero)-cost and easy-to-use fashion*. It has three-fold implications. First, it reuses the existing lighting system for the localization purpose and can be gradually enabled. Second, Epsilon does not rely on any centralized localization service (e.g., a localization database in the WiFi-based solutions). Ideally, the system would be capable of "plug-and-play". It facilitates receiver-side localization so that a device (e.g., a smartphone) can infer its position at a minimum interaction (passive listening, here) with the lighting infrastructure. Last but not least, Epsilon is able to yield high accuracy (sub-meter) localization. In fact, it is promising to achieve unprecedented accuracy by leveraging two advantages of the lighting system rather than other infrastructure-based systems (e.g., WiFi-based). (1) The deployment of illumination lights is much (over one order of magnitude) denser than that of WiFi access points (APs). For example, in our office floor, there are about 21 APs whereas over 300 light sources are deployed to cover the same space. (2) Light sources, unlike WiFi radio signals, are always visible. It

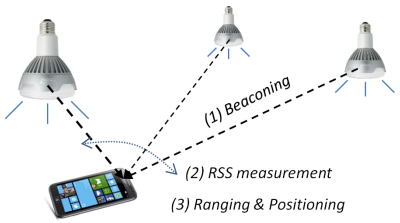


Figure 1: Conceptual design of Epsilon.

exposes a unique opportunity to involve the user in loop for some challenging scenarios.

We design Epsilon to exploit illumination infrastructure for localization purpose. The basic idea is trilateration[1] using visible LED light sources as anchors. In Epsilon (as shown in Figure 1), each bulb, in addition to its major lighting role, also serves as a location landmark. It broadcasts, via the light carrier, location beacons carrying information, i.e., the position of the bulb and its duty cycle, to facilitate receiver side localization. A receiver (e.g., a mobile phone) employs a light sensor to retrieve the beacon information, and measures the received signal strengths (RSSs) from multiple bulbs and computes the distances to each bulb through the optical channel model. Finally, it estimates its location based on the received beacon information and distance measurements from all light sources. Note that the beacons are transmitted via a certain optical channel which is thus free from interferences from ambient light such as sunlight and fluorescent light (more details in §4.2).

Though the basic idea sounds straightforward, it is non-trivial to realize a fast and highly accurate light-based localization system, due to the following three technical challenges. Along with them, we briefly describe the Epsilon solution and our contributions accordingly.

- It is nontrivial to accurately measure the distances between a receiver and surrounding light sources. Many factors such as irradiation angles, phone orientation and light emission power, affect the measurement accuracy. To this end, We establish and experimentally verify a precise optical channel model for localization purpose. We identify all the factors that affect the measurement and their extents, so that we can precisely relate the distance to the RSS (§4).
- It is challenging to obtain a reliable information of each LED bulb (e.g., ID, location, and optical channel parameters), especially among multiple, uncoordinated light sources via the *shared* optical medium. Reliable information transmission over the shared medium, in general, belongs to the VLC paradigm that is yet to come. Therefore, we must design a scheme that does not depend on the existence of a deployed VLC network. Our focus is more on avoiding interferences among a large amount of ad-hoc deployed light sources without any explicit coordination, rather than achieving high throughput. Specifically, we adopt binary frequency shift keying (BFSK) modulation scheme, and mitigate possible collisions through channelization and hopping (§5).
- There are practical challenges to provide robust localization in real situations. In some cases of dense light deployment, given too many observations, how can we precisely localize the receiver by making a full use of all distance measurements which might even interferes or conflicts with each other? In the contrary situations with sparse light deployment, how can a receiver work with few measurements that are even insufficient to uniquely locate the receiver. To this end, we develop precise localization through multilateration techniques, as well as handling practical challenges with the help of simple user interactions (§6).

Though Epsilon is based on existing techniques like BFSK, channel hopping, and intensity modelling, integrating them effectively is non-trivial, and has never been examined before. We have implemented the Epsilon system. To preliminarily evaluate the performance of Epsilon, we build a small hardware testbed by designing and assembling five LED bulbs. We also made a light sensor board that connects to mobile phone through the audio jack. We evaluated Epsilon in typical office environments, including a conference room, a cubicle area, and a corridor, representing various environmental complexities and light layouts. The experimental results confirmed that using visible light yields high localization accuracy: the 90th percentile accuracy reaches 0.4m, 0.7m, and 0.8m for the three environments, respectively. Even in scenarios with a single light source, the 90th percentile accuracy is 1.1m. Although recent work has investigated the idea that exploits LED lighting for indoor localization [13, 15, 22, 24], this is the first piece of work from academia that actually designed, implemented and evaluated a *real* working system, to the best of our knowledge.

2 LED Background

Light Emitting Diode: LED is a simple semiconducting device. We envision that LED lighting will become the mainstream lighting technology in the near future for its several advantages. First, LED bulbs are much more energy efficient (2×) in comparison with the conventional compact fluorescent light (CFL) bulbs. Its lighting efficiency is almost constant (drop by less than 10% after 70,000 working hours [14]) throughout the whole lifes-

[1]The key difference between trilateration and triangulation is the way of determining the location. The former uses the distances to a few anchors while the latter uses the corresponding relative angles.

pan. Second, the lifetime is also much longer lifespan (6×). Third, LED bulbs are free of mercury and thus more environmentally friendly. One drawback of current commodity LED bulbs is its higher production cost, which however is still a win considering the savings on the energy expense.

Instantaneous On/Off: As a semiconductor device, LED possesses a feature – instantaneous on and off. In other words, a LED bulb can be toggled within few microseconds. Our measurements using an oscilloscope show that the rising and falling edges of an ordinary LED are about $4\mu s$. Due to such property, Pulse Width Modulation (PWM) is the most widely used approach to dim a LED bulb, i.e., frequently turning on/off the LED. In PWM, the brightness is determined by the duty cycle. Figure 2 shows two examples with 60% and 20% duty cycles, respectively.

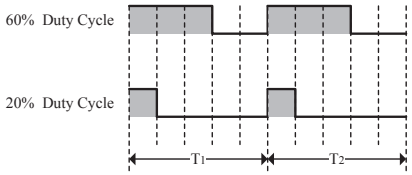


Figure 2: Illustration of pulse width modulation.

Visible Light Communication (VLC): The instantaneous On/Off feature turns a LED lamp into an effective transmitter for VLC. LED bulbs can use various modulation schemes, such as on-off keying (OOK), variable pulse-position modulation (VPPM), and color shift keying (CSK), to embed digital information in its light. VLC has been studied for years [9, 11] and was recently standardized in IEEE 802.15.7 [16]. One special mandatory requirement of VLC is to avoid the *flickering problem*, which is caused by the periodic changes in the instantaneous brightness. It is reported that low-frequency (less than 120Hz [17] or 160Hz [9]) flickers make people feel uncomfortable or even sick. Although there is no widely accepted criterion for the safe flicker frequency, it is generally thought that a frequency higher than 200 Hz is safe.

3 Epsilon Overview

We present *Epsilon* – a visible light based localization system. Figure 3 plots the overall system architecture of Epsilon. It consists of two parts, one on the LED bulb and the other on the receiving device such as a smartphone. Each part consists of several functional modules that collaboratively fulfil the three key technical components of Epsilon, as briefly described below.

Light Beaconing: Each LED bulb broadcasts location

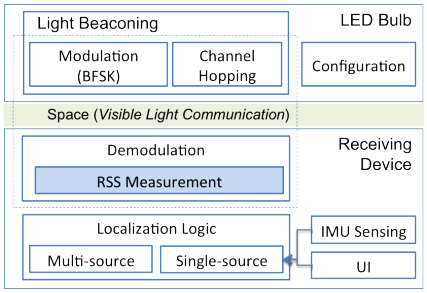


Figure 3: The system architecture of Epsilon.

beacons to the receiver. This is jointly achieved by the modulation module at the LED side and the demodulation module at the receiver side. We adopt binary frequency shift keying (BFSK) modulation to encode the messages. In precaution of possible collisions when multiple, uncoordinated light sources co-exist, we channelize the overall usable spectrum and design a distributed channel hopping logic at the LED bulb. Each beacon is transmitted at a certain optical channel, and thus is interference free from ambient light such as sunlight and fluorescent light.

Distance Estimation: We need to estimate the distances from the receiver to observed light sources, in order for trilateration. The receiver decodes light beacons from multiple light sources, and measure their RSSs, simultaneously. The RSS as well as the information embedded in the light beacon are used to infer the distance from the receiver to a particular light source.

Localization: We design different approaches to localize the receiver, depending on the number of perceived light sources. If over three light sources are perceived, we locate the receiver via trilateration/multilateration which involves an optimization process that maximally respects all distance constraints. Otherwise, we involve the user in loop, and design a process to locate the receiver by fusing the measurements of light and IMU sensors (accelerometer, magnetometer, and gyroscope).

These three design components echo the aforementioned challenges, respectively. Next we will elaborate the details for each of them in the following sections. We start with distance estimation because it is a critical enabler to accurate localization of Epsilon.

4 RSS vs. Distance

To achieve high accuracy trilateration, we need to precisely measure the distances from the receiver to observed LEDs. To this end, we establish a model that can precisely relate the received light signal strength to the distance of a light source.

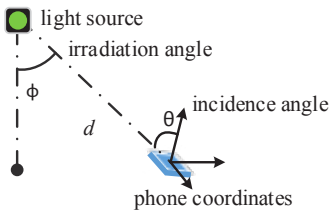


Figure 4: The irradiation angle ϕ; the distance between the light source and the sensor d; and the incidence angle θ. Note that the incidence angle is between the ray and the z axis of the phone coordinate system.

4.1 Optical Channel Model

For an optical wireless link, the received energy over one channel can be described as

$$P_r = P_t \cdot H(d) \cdot G_r, \tag{1}$$

where P_t is the transmission power over a certain channel of the light source. $H(d)$ is the channel gain that is related to the actual sender-receiver distance d. G_r is the receiver gain which can be calibrated once for good. In fact, the channel gain H is not merely a function of the distance, but also depends on the irradiation angle ϕ and the incidence angle θ, as depicted in Figure 4. Intuitively, the longer the distance, or the larger the incidence or the irradiation angle, the lower the received energy.

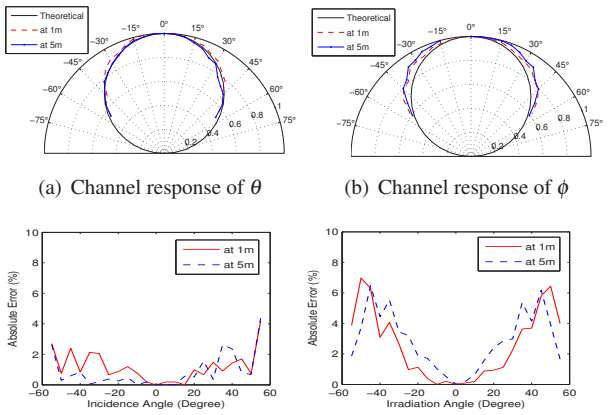
The radiant intensity of a LED chip is usually assumed to follow a Lambertian radiation pattern [4]. Then, the channel gain can be generally modelled by Eq. (2)[2]

$$H(0) = A \cdot g(\phi) \cdot \left[\frac{m+1}{2\pi}\right] \cdot \cos^m \phi \cdot \frac{\cos\theta}{d^2} \tag{2}$$

where A is the area of the sensor detector and $g(\phi)$ is called the optical concentrator which is a constant if the incidence angle falls in the field of view (FoV) of the sensor detector [11]. m is called the Lambertian order which equals 0 for an ideal point light source. For typical LED bulbs with limited illumination range like $\pm 60^\circ$, we have $m = 1$ [4]. The accuracy of distance inference is directly affected by the way of RSS measurement and the precision of the channel gain model. We verify them through a sender-receiver pair, where the sender is a LED bulb and the receiver is a light sensor.

Incidence angle and irradiation angle: We first examine the received energy versus the incidence angle θ and the irradiation angle ϕ. According to the channel model in Eq. (2), the received energy follows the cosine of θ and ϕ, which is shown in black solid curve (Theoretical) in Figure 5(a) and 5(b). We measure the observed channel response for $\theta \in [-60^\circ, 60^\circ]$ and $\phi \in [-60^\circ, 60^\circ]$, at

[2]The parameter '0' in $H(0)$ is an abbreviation, referring to all affecting parameters.



(a) Channel response of θ (b) Channel response of ϕ

(c) Relative error of θ (d) Relative error of ϕ

Figure 5: Normalized channel responses and relative errors of the incidence angle θ and irradiation angle ϕ measured at 1m and 5m distances.

distances of 1m and 5m from a LED bulb, respectively. The normalized measurement results are also plotted in Figure 5(a) and 5(b) in red dashed and blue dotted curves, respectively. To quantify the relative error between our model-based estimation and actual measurement, we define the *error ratio* (denoted as r) as

$$r = \frac{|rss_{measured} - rss_{model}|}{rss_{model}} \tag{3}$$

where $rss_{measured}$ and rss_{model} are the RSSs from measurements and model-based derivations. The error ratios regarding to both angles are shown in Figure 5(c) and 5(d), respectively. In Figure 5, we can see that the real measurements fit the model very well. The error ratio is mostly below 5% when the angles (θ and ϕ) are within $\pm 60^\circ$. Once they exceed $\pm 60^\circ$, the error ratio grows significantly, though the absolute error still stays relatively low. It is caused by the physical limitations in the FoV of ordinary LED chipsets and the light sensor.

LED-receiver distance: According to Eq. (2), the received energy falls off against the the distance d, following an inverse-square law. We verify this by fixing the incidence and irradiation angles to 0° and vary the distance from 1m to 5m with the step length of 0.25m. The measured channel responses are shown with scatters in Figure 6. We then fit the scatters with function C/d^2 where C accounts for the constant coefficients in Eq. (2). From Figure 6, we could see the overall fitting error is very small, with root-mean-square error (RMSE) being 1.85e-4 and $C = 0.0018$. Therefore, the inverse square model accurately characterizes the relation between distance and RSS.

In our experiments, we found that the constant C will be different when using the same light sensor with differ-

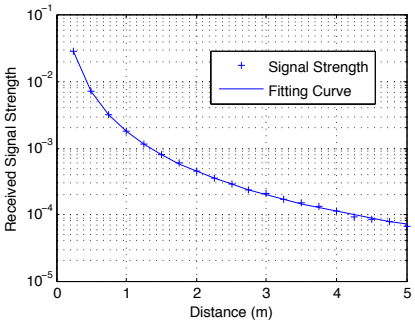


Figure 6: The received energy versus the distance from 1m to 5m with step length 0.25m. Both θ and ϕ are fixed to 0°. The duty cycle of the PWM is 50%.

ent light sources. We thus conducted additional experiments using a cross-validation approach, that is, trying different combinations of LEDs and light sensors. We found the coefficient C can be factored into C_L and C_s (i.e., $C = C_L \cdot C_s$), where C_L is per-LED constant and relates only to its maximum power, and C_s is a per-sensor constant and related to its receiver gain. Both C_L and C_s are constant and can be measured once for good, e.g., preferably by the manufacturer, and stored in the LED and the sensor device. Since we are exploiting multiple light sources to locate a device, thus for sake of clear presentation, we will leave out C_s and use C to indicate C_L in formulations throughout the rest of the paper.

Putting all insights we gain so far together, with known C, θ and ϕ, the distance can then be precisely derived with the measured RSS.

4.2 Emission Power from a LED

We now address how to model the emission power from a LED light source. The light signal from a LED is a 0-1 pulse wave as shown in Figure 2. Suppose the period is T with pulse time τ. The Fourier series expansion for this pulse wave is

$$f(t) = \frac{\tau}{T} + \sum_{n=1}^{\infty} \frac{2}{n\pi} \sin\left(\frac{\pi n \tau}{T}\right) \cos\left(\frac{2\pi n}{T} t\right) \qquad (4)$$

The equation above indicates that the emission power of the LED spreads over the baseband (the first AC component) and all the harmonics. Thus, it is infeasible to measure the overall received energy. Fortunately, for sake of localization, we only measure the portion of energy over the baseband optical channel which already validates the channel model in Eq. (2).

The RSS hereafter is thus defined as the magnitude of the baseband frequency component. Actually, the transmitted energy at the light source is proportional to the coefficient of the first AC component in Eq. (4), i.e., $P_t \propto \frac{2}{\pi} \sin(\pi\tau/T)$. Note that P_t is not affected by the actual baseband frequency, but interestingly, it is a function of duty cycle τ/T of the PWM. This insight indicates that the light source also needs to convey the duty cycle information in its beacon for the receiver to correctly model the transmission power. In conclusion, the RSS measured at a receiver is calculated as follows:

$$P_r = C \cdot \sin(\frac{\tau}{T}\pi) \cdot \frac{\cos\theta \cdot \cos\phi}{d^2} \qquad (5)$$

where C and τ/T are the per-LED constant related to its maximum emission power and the current duty cycle of the LED, both included in its beacon.

Interference from ambient light: Note that the measured power from a LED is the portion within a certain frequency range. Most ambient lights are with only DC component (e.g., sunlight) or energy at a fixed frequency (e.g., incandescent and fluorescent bulbs at 100 Hz or 120 Hz). Therefore, the RSS at a receiver in our system will not be affected by these ambient light sources, by simply avoiding these colliding frequencies. In § 8, we actually evaluate our system with the co-existence of various kinds of ambient light sources.

5 Beaconing Over Visible Light

The PMW-based dimming mechanism of LED enables communication with visible light. We now present our design to achieve reliable location beaconing.

5.1 Communication with BFSK

Many modulation schemes were proposed in the VLC field, such as OOK, VPPM, and CSK. They all can be adopted to carry location beacons in the light carrier. However, they require either sophisticated decoding logic or special hardware, and also special mechanisms to avoid the flickering problem. In Epsilon, we use the binary frequency shift keying (BFSK) for its simplicity and the natural prevention of flicker – there is no flicker issue for the carrier frequencies over 200 Hz.

In BFSK, symbol 0 and 1 are represented by two frequencies f_0 and f_1, each frequency lasting for a certain duration (termed *symbol length*). The receiver demodulates the incoming BFSK signal by transforming (FFT) the sensed light signals in a decoding window, whose length equals to the symbol length, to the frequency domain, and performing a binary decision on the major frequency component. The transform is carried out in a sliding fashion: each time the window advances by a fraction of the symbol length. More details on modulation/demodulation could be found in our previous work [10], which is omitted here due to the space limitation.

5.2 Channelization and Hopping

The major challenge of reliable beaconing is the collision problem caused by multiple, uncoordinated, and unsynchronized light sources over shared light medium. It is extremely difficult to coordinate among the light sources. First, no light lamps are equipped with extra sensors to find their neighbors. Moreover, the actual deployment of light sources (e.g., usually attached to ceiling) makes it difficult for the light sources to see/sense each other. This is different from most wireless radios where each transmitter also serves as the receiver. Consequently, time division multiple access is not feasible in our scenario as they require synchronization or a carrier sensing mechanism among senders.

We choose to channelize the whole available spectrum into multiple disjoint and even spaced sub-carriers. In Epsilon, each LED bulb is configurable, thus it tends to think of manually assign a static channel for each LED. Unfortunately, this is infeasible due to the unknown coverage of each light source and how multiple sources' coverages may intersect with each other. Even though we can see the coverage, it is unlikely to adjust the power to avoid interference, as the power control should serve primary lighting function. Therefore, we adopt random channel hopping to avoid persistent collision among light sources. The timeline is divided into slots (called a *hopping period*) with equal length. Each light source randomly picks one channel in each hopping period, transmits a beacon, and then hops to another channel. As long as the number of channels is large enough in comparison with the number of contending LEDs, random hopping actually handles the problem of collision effectively. We formulate and analyse the random channel hopping scheme in the next section.

Another unsolved problem is to select the communication band. Intuitively, the overall usable spectrum is jointly determined by a few factors such as the minimum frequency to prevent flicker and the On/Off speed of the LED bulb. We will discuss more about spectrum selection in § 7.

5.3 Minimizing the Waiting Time

Collisions may still occur under random channel hopping. Note that when a collision happens, the receiver needs to wait for additional hopping periods to correctly receive the beacon.[3] Regarding to good indoor localization user experience, short *waiting time* is highly desired, which is directly related to the time to correctly receive all beacons from all sources.

[3]Sometimes, the receiver can still decode one beacon from the collided signals due to the *capture effect* [19]. However, it will affect the RSS measurement and hence adverse to distance estimation.

Suppose one light sensor observes M light sources and the number of channels is N. The waiting time t_w can be formulated as

$$t_w(M) = k(M) \cdot \tau \tag{6}$$

where k is the number of hopping periods and τ is the length of each hopping period. Note that both the t_w and k are functions of M. Given the fixed overall spectrum, we have $\tau \propto N$ as increasing the number of channels implies narrower channels, hence lower data rates (i.e., $\tau \uparrow$). Nonetheless, increasing N reduces the collision probability, and less hopping periods will be waited (i.e., $k \downarrow$). Based on this formulation, we discuss how to obtain the optimal N below.

Note that, without clock synchronization among light sources, the hopping periods of different light sources are likely misaligned. Thus, one light source may partially collide with another, leading to corrupted beacons from the two colliding sources. Assume the channel selection during each period is independent and uniformly distributed in $[1,N]$. The probability that, in k consecutive hopping periods, one light source does not collide with any other light sources for at least one hopping period (which guarantees correct decoding at the receiver) is

$$p = 1 - \left[1 - \left(1 - \frac{1}{N}\right)^{2(M-1)}\right]^k \tag{7}$$

In Eq. (7), M is a constant which is determined by the environment. Once N is given, we can derive the minimum k so that $p \geq P_0$, where P_0 is the success rate the system desires:

$$k(M) = \arg\min_k p \geq P_0 \tag{8}$$

Once M and P_0 are given, we can find the optimal N that minimizes the waiting time by combining Eq. (6), (7), and (8). For instance, the optimal number of channels for $M = 3$ is 7, and the corresponding number of hopping periods is 3.

In practice, the density of light sources varies from place to place. Therefore, it is unable to find a globally optimal N for all situations. Empirically, we may select the N which minimizes the maximum waiting time for typical settings such that $M \in M_{typical}$, i.e.,

$$N = \arg\min_k \max_{M \in M_{typical}} t_w(M) \tag{9}$$

where $M_{typical}$ represents typical numbers of light sources that are observed by the receiver. In our system, P_0 and $M_{typical}$ are set to 90% and $[3,10]$ respectively, where correspondingly $N = 30$ and $k = 3$, i.e., the communication band is divided into 30 channels and the

receiver waits for 3 hopping periods. During the waiting time, the receiver obtains multiple beacons from the same light source. We select the one with the lowest signal strength for further processing, which is less likely corrupted by other beacons. The actually waiting time depends on the bandwidth of each channel which is further discussed in §7.

6 The Localization Algorithm

The localization core is to use trilateration to calculate the receiver's position from distance measurements to multiple light sources. We first address the normal cases with sufficient light sources and then the challenging cases with insufficient sources.

6.1 Localization with Trilateration

Given n light sources, we can apply Eq. (5) to establish a series of constraints on measured RSSs and distances, as well as the angles. Those are,

$$\begin{cases} P_{r1} = C_1 \cdot \sin(\alpha_1 \pi) \cdot \frac{\cos\theta_1 \cdot \cos\phi_1}{d_1^2} \\ P_{r2} = C_2 \cdot \sin(\alpha_2 \pi) \cdot \frac{\cos\theta_2 \cdot \cos\phi_2}{d_2^2} \\ \cdots \\ P_{rn} = C_n \cdot \sin(\alpha_n \pi) \cdot \frac{\cos\theta_n \cdot \cos\phi_n}{d_n^2} \end{cases}, \quad (10)$$

where α_i and C_i are obtained from the i^{th} beacon.

Let the 3D coordinates of the receiver and the light source are $\langle x_0, y_0, z_0 \rangle$ and $\langle x, y, z \rangle$, respectively. Their distance is $d = \sqrt{(x_0-x)^2 + (y_0-y)^2 + (z_0-z)^2}$. For simplicity, we assume all the light sources facing downward (it is usually the most common case with lights on the ceiling), we further have $\cos\phi = |z - z_0|/d$. In case that the receiver's light sensor faces squarely upward toward the ceiling, we have $\theta = \phi$. Therefore, only three unknowns remain, namely x_0, y_0, z_0. Later, we discuss the general case with arbitrary light deployment or phone orientation, which only requires extra calibration, local angle detection, or more distance measurements.

With four or more light sources, we may uniquely determine all unknowns. The localization is an optimization process trying to minimize the linear mean square (LMS) error, which is actually a well-studied topic [18]. Here, we use Newton's Method for the optimization. The goal is to minimize the sum of absolute error between the left and right side of each equation in Eq. (10). In our implementation, we generate the initial values for each unknown randomly and ran the optimization process multiple times to avoid local minima. Note that if only three light exist, the optimization may end up with two optimums, one of which is actually fake (above the ceiling due to even symmetry property of the cosine function in the model). In practice, light sources may deployed at similar height and hence such ambiguous solutions can be filtered out by common sense (e.g., the device is unlikely higher than the ceiling). As a result, three light sources are typically the minimum required number in our system.

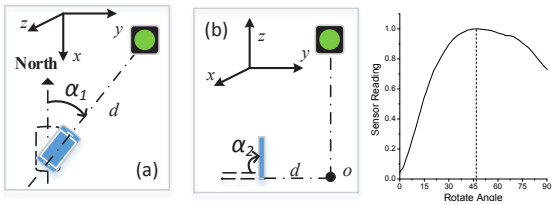


Figure 7: Localization with only one light source. (a) keep the phone in the horizontal plane and rotate it for an angle of α_1 until pointing to the light source (top view); (b) pitch the phone for an angle of α_2 where $\alpha_2 \geq$ the incidence/irradiation angle (side view).

In real usage, the receiver (hence the light sensor) may be in arbitrary orientation. This will complicate the problem. Intuitively, we can leverage the equipped orientation sensors (e.g., inertial measurement unit (IMU) on the phone) to measure the device's attitude and transform back to the horizontal attitude. In case not all the light sources face downward, their angles can be pre-obtained via calibration and delivered to the receiver via beacons. More measurements can help to solve the general localization problem by introducing more constraints. Even the light source and the receiver is not perfectly facing down or up, slight imperfection actually has little impact to the location accuracy, as their impact to the distance estimation is via a cosine function that changes slowly near 0. We evaluate this in §8.3.

6.2 Involving the User

In real situations, we may end up with insufficient number (e.g., one or two) of light sources that cannot uniquely locate the device. For instance, there might be only a single lamp in a room; in a long corridor or tunnel, a serial of lamps are usually deployed with a long distance between neighboring lamps, where the device can sense only one (or two) light sources in most of the time. While using the lamp position (coverage-based method) already fulfils rough position estimation, we discuss the option of involving the user if a higher location accuracy is desired. Note that this is an advantage of visible light than RF signals (e.g., WiFi, FM, and Geomagnetism) which are not perceptible by human.

Figure 7 illustrates the procedure of the user gestures. It contains two steps. The first step is *exactly* the same as we use compass to find direction. That is, the user holds the phone horizontally and rotates the phone (around device's Z-axis) as shown in Figure 7(a), until the phone is

oriented to one light source. The second step is to gradually pitch the phone. In the meantime, a continuous measurement is performed to record the RSSs from the light source at different phone pitch angles (Figure 7(b)).

The procedure above basically uses inertial sensors to measure the irradiation and the incidence angles. In the first step, it measures the orientation angle, α_1, between a virtual line connecting the phone and the light source and the North from the compass. We also record the RSS, as P_{r1}, at the point when the phone is pointing to the light source. The user then continues with the second step by pitching the phone from the horizontal attitude to the roughly vertical attitude for an angle of α_2 while keeping the phone screen facing the light source. Note that α_2 should be larger than the incidence/irradiation angle and it can be easily fulfiled as long as it passes the point at which the phone screen faces squarely towards the light. The system logs the reading of the light sensor as well as the instantaneous pitching angle (around device's X-axis) that is captured by the gyroscope. An instance of the logged light sensor trace is given in Figure 7. We can see that the sensor readings increase until a peak point and then decrease. This is caused by the changing incidence angle. Thus, the peak point is the instant that the device faces squarely to the light. The corresponding pitched angle from the beginning to the peak point is the desired incidence angle θ when the phone was placed horizontally.

After the two steps, we obtain the incidence angle θ, RSS P_{r1}, and the angle α_1. With the model in Eq. (5), the former two measured parameters ensure all the possible device positions are in a 2-D horizontal circle around the light source. Then, we can use α_1 to finally determine only one location in the circle.

7 System Implementation

Hardware Design: Note that Epsilon is still a pioneer work exploiting LED for localization, there is thus no off-the-shelf product that supports programming and VLC. We designed a small LED lamp, as shown in Figure 8, with a commercial LED (Model: Cree T6) [8] with 10W marked power and peripheral control circuit to adjust the beaconing content. The modifications to the LED is easily met in practice. As the commodity LED bulbs already employ PWM for dimming purpose, we only need to add the capability of varying the frequency for BFSK. For the receiver design, modern mobile phones ship with light sensors. However, it turns out that the OS restricts the sampling rate (e.g., Motorola XT910 $\sim$ 5Hz, Samsung Galaxy SIII $\sim$ 100 Hz). While we envision that we can modify the driver in the future, we currently design a small light sensor board, that merely consists of a

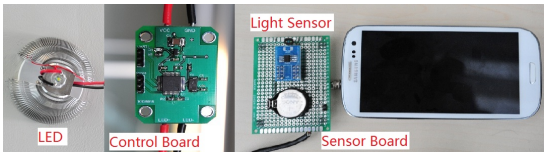


Figure 8: The hardware design of Epsilon.

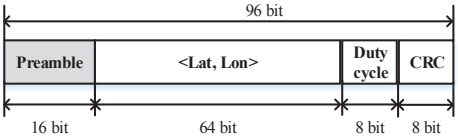


Figure 9: Beacon frame used in Epsilon.

light sensor, an amplifier, and a small battery. We connect the board to the phone through the audio jack. The sampling of light sensor is performed using the ADC for microphone. As will be shown later, the audio ADC imposes certain design constraints regarding to the usable communication band.

Configuration and Frame Design: In our system, we embed the coordinates of each light source in its beacon. Having a back-end service for mapping the ID of each LED to its physic location is an alternative solution, which however relies on the network connection. Therefore, we insist on making each LED bulb self-contained. Another practical issue is that we need to configure the location for each LED. We rely on the profile (e.g., blueprint map) from building management to configure the position of each bulb. Each beacon payload consists three parts: preamble, location information, and the duty cycle, as shown in Figure 9. The preamble consists of 2 bits of zeros to facilitate RSS measurement. The location information is a 64-bit latitude and longitude tuple. We use a 8-bit number to represent the duty cycle, which corresponds to $\sim 0.4\%$ dimming adjustment granularity. We also adopted a 8-bit CRC to check the integrity of the contents.

Communication Band Selection: As discussed in §5.3, we desire wider communication band for less waiting time. Suppose the band used for communication is $[f_l, f_h]$ where f_l and f_h represent the lower and upper boundaries, respectively. There are actually several constrains in determining the two boundaries.

1. f_l should be high enough to avoid the flickering problem, i.e., $f_l \geq 200$Hz. f_h cannot exceed the minimum of the LED On/Off speed and the light sensor response speed as discussed in §5.2, which is 118.2 kHz.

2. $f_h < 2f_l$. Based on Eq. (4), the pulse wave carrier results in harmonics while the transmitted energy spreads across all harmonic frequencies $f = 2\pi n/T$.

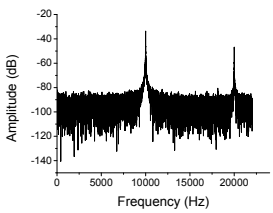


Figure 10: Power spectrum of 10kHz light carrier

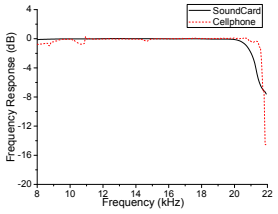


Figure 11: Frequency response of audio ADC.

Figure 10 plots one such example, where the light carrier frequency is 10kHz. Energy peaks can be observed at 10kHz and 20kHz (30kHz, 40kHz, etc. are omitted in the plot). To avoid harmonic interference, f_h should be lower than $2 \times f_l$.

3. Constrained by the sampling rate (up to 44.1 kHz) by audio ADC, f_h should not exceed 22.05 kHz, according to the Nyquist theorem.

With above constraints, we choose to use the band from 10 kHz to 19 kHz in our implementation. We divide the band into 30 channels, each with 300Hz bandwidth and the corresponding data rate at each channel is 120 bps. For each beacon, one hopping period is about 0.7s. The overall waiting time is thus around 2.1s.

Frequency Response of Audio ADC: The sensed light signal is affected by the audio ADC circuit of the mobile phone (we disabled the auto gain control). Figure 11 shows the frequency response we measured using a high end sound card (AVID M-Audio C600 [2]) and a smartphone (Samsung Galaxy S III). The frequency response of the sound card is perfectly flat from 10 kHz to 20 kHz, so that the receiver gain in Eq. (1) can be viewed as a constant. For the phone, there are small fluctuations near 10.5 kHz and 14.5 kHz due to hardware issues. The fluctuations are all below 1 dB which is thus small enough to tolerate.

8 System Evaluation

We first evaluate Epsilon with a small-scale hardware-based testbed, and then moderate-scale model-based simulations. Our hardware-based evaluations focus on the localization accuracy, while the simulations covering other performance aspects such as robustness with respect to light source selection and imperfect incidence angles. We designed and assembled 5 LED lamps, and evaluated Epsilon under three typical office environments: a conference room, a cubicle area, and a corridor. The environments and the deployed LEDs are shown in Figure 12. They represent different environmental complexities and reflection characteristics. The areas are 5m×8m, 2m×12m, and 3.5m×6.5m, respectively. For each area, we place the phone at 60 positions and run multiple tests at each position.

Methods for Comparison: We compare Epsilon with two intuitive methods in our experiments:

- Coverage Method: it locates a receiver to the position of the light source that the receiver sees the highest RSS.
- Weighted Average: it locates a receiver as the weighted average of the locations of the sensed light sources, using their RSSs as weights.

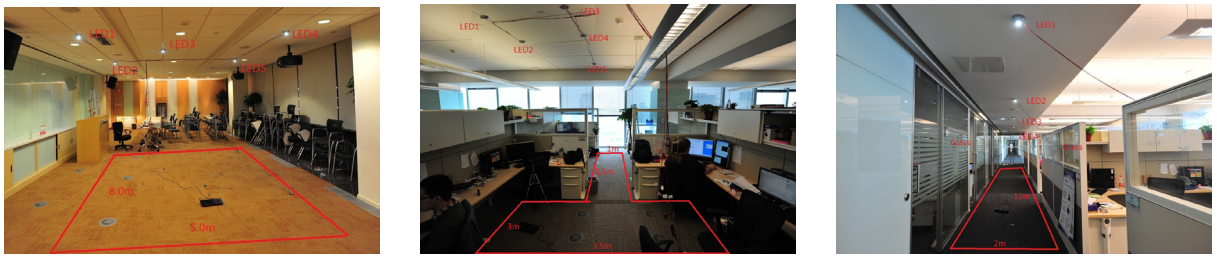
It is difficult to compare Epsilon with existing localization algorithms based on other signals (e.g., WiFi) side by side. We thus empirically elaborate some numeric results from our evaluation as well as those reported by the state-of-art in §8.4.

8.1 Localization with Multiple LEDs

Figure 13 plots the localization errors in three scenarios where the sensor is put at various locations in the interested area. It shows that Epsilon yields high accuracy for all the three environments. The medium error is about 0.3m and the 90th percentile errors are 0.45m, 0.7m, and 0.8m in the conference room, the cubicle area, and the corridor, respectively. Among the three scenarios, the conference room is the simplest as it is mostly empty; the cubicle environment is actually the most complicated. However, thanks to the better layout of the LEDs, the performance in the cubicle area is actually better than that in the corridor. The corridor is also empty, but the LEDs are placed almost in a straight line.

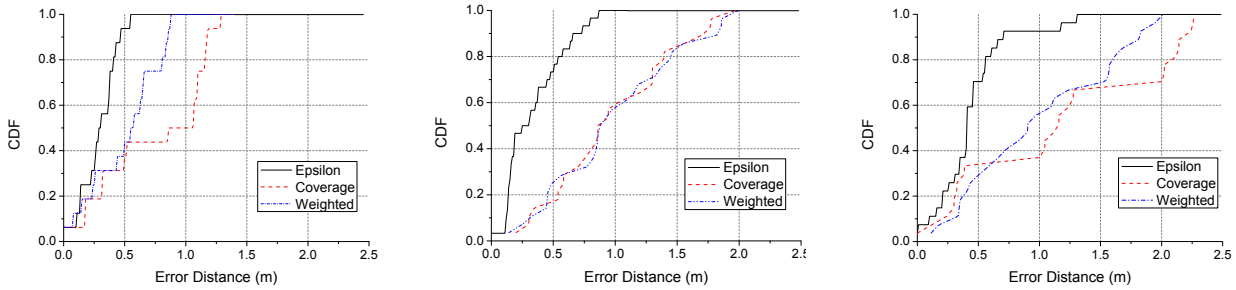
We also examine the localization accuracy for each position in detail. We find that center area (i.e., the area surrounded by lamps) positions tends to have smaller errors than outer positions. The reason is that center area positions have a better chance to observe light sources with small incidence angles which are thus more robust to measurement noise. For the corridor environment, we find those positions with the largest errors are exactly the positions at the two edges of the corridor. The result suggests that we should evenly deploy the LEDs for better accuracy. Fortunately, typical deployment of light sources already follows this natural rule to deliver even illumination conditions. One should note that all the evaluations above are performed with various ambient light sources (e.g., sunlight or fluorescent lamps). We also run experiments at night with all other lights off and there is no visible difference with or without ambient lights.

Epsilon always outperforms the Weighted Average method and pure Coverage method. By exploiting the characteristics of optical channels, Epsilon improves the



(a) Conference Room (b) Cubicle Area (c) Corridor

Figure 12: Deployments of Epsilon in a conference room, a corridor and a cubicle area, each with five LEDs.



(a) Conference Room (b) Cubicle Area (c) Corridor

Figure 13: Localization accuracy with multiple LEDs under the three experimental environments.

90th percentile accuracy by 1× (than Weighted Average) or 2× (than Coverage). Nevertheless, the accuracy of the latter two methods is still high. The 90th percentile accuracy is always smaller than 2 meters. Given their simple designs, these results clearly demonstrate the advantage of using visible light.

8.2 Localization with a Single LED

We evaluate our single LED based localization method using one LED in the corridor case. Each measurement follows the process described in §6.2. Note that the key is to measure the angles using the IMU sensors, which are the main error source. We first examine the sensor errors shown in Figure 14. All the data is measured in our office building at various locations. The left figure shows the distribution of compass sensing errors, while the right one shows the distribution of the errors between the measured incidence angles versus the groundtruth. Figure 14 reveals that both errors distribute normally within a certain range: the compass errors fall in $\pm 26^\circ$ while the gyro errors in $\pm 7^\circ$ in 95% credible interval.

Figure 15 shows the resulting location error of both Epsilon and Coverage. We can see that Epsilon significantly outperforms Coverage: the accuracy is improved by 5×. In most cases ($\sim$ 95%), the errors of Epsilon fall below one meter. It demonstrates that with simple user involvement, we are able to achieve quite high localization accuracy even with only one light source.

The experiment above demonstrates the advantage of using perceptible signals under the user's help for localization. Note that, in practice, the user often walks under a LED lamp, which naturally imports the user involvement into Epsilon.

8.3 Model based Simulation

We perform model-based simulation for two reasons. First, our results on the real testbed results have demonstrated that the optical channel model fits our measurements quite well. Hence, using model based simulation does make sense. Second, our testbed with only 5 LED lamps limits us to explore robustness and performance of Epsilon under more light sources or with imperfect incidence angles. In fact, abundant light sources raise an interesting question: does Epsilon perform better as it uses all the measurements from more sources? If not, how should it smartly use the observed light sources for localization? To answer these questions, we propose a new scheme called *Epsilon-s*, which performs a light source selection procedure before localization. Specifically, we select the top four sources with the highest RSSs among all observed light sources. The heuristics is that light sources with higher RSSs tend to be closer and with smaller incidence/irradiation angles.

Regarding the error of the incidence angle $\Delta\theta$, we want to evaluate its impact on the localization accuracy. In Epsilon, the LED-receiver distance depends on three

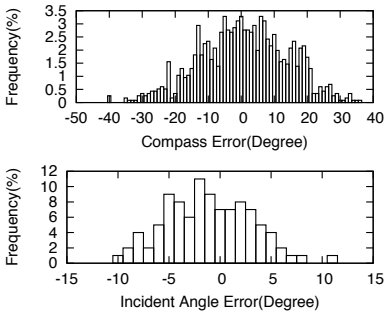


Figure 14: Histograms of the measurement errors caused by compass and gyroscope sensors.

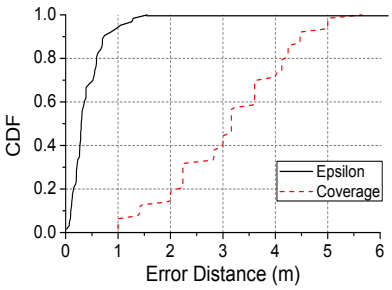


Figure 15: Evaluations of localization with a single LED.

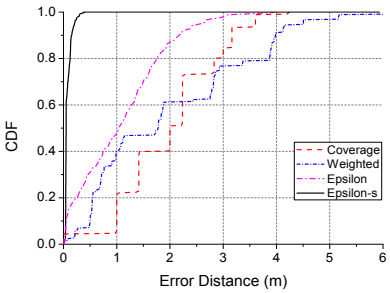


Figure 16: Impact of imperfect incidence angles.

parameters, namely the RSS, irradiation angle ϕ of LED, and the incidence angle θ of the light sensor. The RSS is directly measured by the device, and ϕ is determined by the relative position of the receiver to the LED. The only uncertain variable is θ, as we ask the user to hold the phone horizontally, which is error prone due to various reasons. Therefore, it is necessary to evaluate the impact of imperfect incidence angles to the localization accuracy.

The simulation is done in a $20m \times 20m \times 3m$ space. We place light sources uniformly on the ceiling (height = 3m), each at $\langle 4i, 4j, 3\rangle$ where $i, j \in [1,4]$, and thus we have a total of 16 light sources. We put receiver at $\langle x, y, 0\rangle$ where $x, y \in [0, 20]$. For each receiver location, we set the error of the incidence angle, denoted as $\Delta\theta$, to an angle randomly within $\pm 20°$. We then measure the localization error under different schemes.

Figure 16 plots the simulation results. It shows that Epsilon outperforms both the Coverage and Weighted Average. The Coverage method performs comparably to the Weighted Average. In contrast, Epsilon-s performs significantly better than the other three. Recall that the incidence angle θ relates to the distance via a cosine function, which changes slowly at small angles but very quickly at large angles. Epsilon uses all sensed light sources, which would include faraway ones. Their θs (and ϕs as well) are usually large. Thus small $\Delta\theta$ can make a big impact to the distance estimation, which impairs the localization accuracy. In contrast, Epsilon-s uses only high RSSs lights that have small θs, and is thus more tolerant to $\Delta\theta$. Note that, the tolerance of Epsilon to small $\Delta\theta$ implies less restriction to the actual use, which allows the user to place their phone more casually.

8.4 Comparison with WiFi-based Methods

Current mainstream indoor localization systems are WiFi-based, which basically achieve meter level accuracy. Recently, ArrayTrack [21] achieves sub-meter accuracy using multi-antenna technique. However, it relies on multiple APs to work collaboratively to measure the angle of arrival (AoA), which is non-trivial. In practice, the APs are typically deployed by different parties. Also, the main purpose of APs is for networking, and thus the number of antennas is less than required (16 antennas for each AP) in [21]. We thus only summarize the basic properties of representative WiFi localization systems. Note that it is difficult to fairly compare them with Epsilon due to the impact of infrastructure deployment and database density. We therefore only excerpt their performances from the original paper. We see that Epsilon yields the best accuracy. Even with simple Coverage method, visible light based localization is already as good as the best WiFi localization system.

Name	EZ [6]	Radar [3]	Horus [23]	Coverage	Epsilon
Accuracy	2 - 7m	3 - 5m	$\sim$ 1m	$\sim$1m	$\sim$0.4m
Method	Model	FP	FP	FP	Model
Database	Yes	Yes	Yes	No	No
Overhead	Minimum	WD	WD	DC	DC

Table 1: Comparisons with representative WiFi-based localization systems and coverage-based lighting localization. In the table, FP, WD and DC mean fingerprinting, war-driving and device configuration, respectively.

9 Discussions

Epsilon is still in its infancy. In this section, we briefly discuss potential issues and open questions in real usage.

Applicability: To leverage the visible light, the device needs to be exposed to the light. This may limit its applicability, e.g., it is not possible with the phone in pocket. Thus, Epsilon targets at localization with explicit needs (user awareness), rather than passive tracking scenarios. The light has to stay on, which might be an issue for the sake of energy efficiency. Favourably, for most indoor environments (e.g., offices or shopping malls) where localization is desired, lights (at least a small portion) are mostly, if not always on.

Device Diversity: Different LEDs and light sensors may have different emission power and receiving sensitivity, which would directly affect the distance measurement. Fortunately, as solid-state devices, the intrinsic characteristics of LEDs and light sensors are highly stable over time [14]. Therefore, for each LED and each light sensor, one time calibration is enough. Considering their long lifetime (say 5 years), this cost is still reasonably small. For practical use, we may reduce calibration efforts, for example by automatically calculating the LED parameters as done for WiFi in [6].

Shadow and Reflection: Similar to the multipath issue in WiFi-based localization, using visible light for localization may suffer from shadowing and reflection of the light. For instance, when holding a phone in front of body, body reflection, especially in white shirt, will bring noise to localization. Sometimes, his/her body is a big obstacle, blocking the phone from lighting. For these issues, Epsilon counts on the user's involvement. We admit that involving the user's help is burdensome. However, on the other side, we argue this is also an opportunity. The light is visible which naturally offers a feedback to the user and makes the case to easily obtain the user's help to improve the localization accuracy, unlike other invisible RF signals.

Modeling vs. Fingerprinting: The model-based approach in Epsilon achieves good results only when the LED and the light sensor are within each other's FoV. This limits the application scope and we may have to fall back to coverage-based coarse-grained localization. Fingerprinting method will not have such constraints. However, a fingerprint is highly affected by a variety of factors such as the device attitude, body blocking of light, and etc.. In addition, similar to any fingerprinting-based system, it requires to construct a database, which is a challenging task.

10 Related Work

Most existing localization work leverages signals such as WiFi [3, 6, 20, 21, 23], FM [5], magnetism [7]. WiFi based approaches [3, 23] leveraging existing infrastructure typically achieves meter level accuracy. Recent work [21] exploiting multi-antenna achieves sub-meter accuracy with non-trivial modifications to the hardware. Our work is a radical deviation from these efforts. Here, we only review the closely related work, i.e., those dealing with visible lights.

Visible Light based Indoor localization: A few recent works also explore the idea of using visible light for localization [13, 15, 22, 24], all purely based on simulation. In [15,22], image sensors are used to locate the surrounding light sources based on the ray projection model. In [24] distances to multiple light sources are estimated by varying the transmitting power, which leads to unstable illumination. In [13], the authors infer TDOA from the peak-to-peak value of the interference signals from two LED lights. In contrast, in Epsilon, we build accurate optical channel model applicable to localization with practical considerations like dimming and flickering avoidance, and working with multiple light sources. Compared with our previous work [10], we address more practical challenges, such as enabling reliable communication and robust localization even with insufficient sources or imperfect orientation. To our best knowledge, ByteLight [1] is the only existing commercial LED based solution for indoor localization. However, there is no publicly available information on how their system works.

Visible Light Communication: VLC aims to leverage visible lights as communication carriers. The recent standard IEEE 802.15.7 specifies the hardware, modulation, channel coding, and the MAC protocol for various applications [16]. A number of studies discuss optical channels for VLC such as [9, 11, 12]. While VLC research mainly focus on wideband high-speed communication, we aim at low system complexity and robust broadcast for localization purpose.

11 Conclusion

In this paper, we present the design, implementation and evaluation of Epsilon, a visible light based localization system that exploit LED lamps. The system has no dependency on network access and can be used immediately after proper configuring and calibrating the LED bulbs. We have identified and overcome key technical challenges for accurate distance measurement using light, reliable location beaconing, and robust localization where the number of light sources can be excessive or insufficient. Our evaluation in typical office environment confirmed the effectiveness of the system, which achieve sub-meter accuracy. Our work confirms the potential of visual light for high accuracy indoor localization. In addition, our work also reveals several insights that deserve further exploration.

12 Acknowledgements

We thank all the anonymous reviewers and the shepherd Nikolaos Laoutaris for the valuable constructive comments to help enhancing the quality of this paper.

References

[1] ByteLight. http://www.bytelight.com/.

[2] AVID. M-AUDIO C600. http://www.m-audio.com/.

[3] P. Bahl and V. N. Padmanabhan. RADAR: An In-Building RF-Based User Location and Tracking System. In *IEEE INFOCOM*, 2000.

[4] J. R. Barry. *Wireless infrared communications*, volume 280. Springer, 1994.

[5] Y. Chen, D. Lymberopoulos, J. Liu, and B. Priyantha. FM-based indoor localization. In *ACM MobiSys*, 2012.

[6] K. Chintalapudi, A. P. Iyer, and V. N. Padmanabhan. Indoor localization without the pain. In *ACM MOBICOM*, 2010.

[7] J. Chung, M. Donahoe, C. Schmandt, I.-J. Kim, P. Razavai, and M. Wiseman. Indoor location sensing using geo-magnetism. In *ACM MobiSys*, 2011.

[8] CREE. cree® Xlamp® Xm-l leds. http://www.cree.com/.

[9] D. Giustiniano, N. O. Tippenhauer, and S. Mangold. Low-complexity visible light networking with led-to-led communication. In *Wireless Days (WD), 2012 IFIP*, pages 1–8. IEEE, 2012.

[10] P. Hu, L. Li, C. Peng, G. Shen, and F. Zhao. Pharos: Enable physical analytics through visible light based indoor localization. In *to appear at HotNets-XII*, 2013.

[11] T. Komine and M. Nakagawa. Fundamental analysis for visible-light communication system using led lights. *Consumer Electronics, IEEE Transactions on*, 50(1):100–107, 2004.

[12] D. C. O'Brien, L. Zeng, H. Le-Minh, G. Faulkner, J. W. Walewski, and S. Randel. Visible light communications: Challenges and possibilities. In *PIMRC 2008*, pages 1–5. IEEE, 2008.

[13] K. Panta and J. Armstrong. Indoor localisation using white leds. *Electronics letters*, 48(4):228–230, 2012.

[14] PHILIPS. A Long Lifespan I LED. http://www.lumec.com/newsletter/architect_06-08/led.htm.

[15] M. Rahman, M. Haque, and K.-D. Kim. High precision indoor positioning using lighting led and image sensor. In *ICCIT*, pages 309–314. IEEE, 2011.

[16] S. Rajagopal, R. D. Roberts, and S.-K. Lim. IEEE 802.15.7 visible light communication: modulation schemes and dimming support. *Communications Magazine, IEEE*, 50(3):72–82, 2012.

[17] E. Star. Energy star®. *Program Requirements for Residential*, 2010.

[18] E. Süli and D. F. Mayers. *An introduction to numerical analysis*. Cambridge University Press, 2003.

[19] K. Whitehouse, A. Woo, F. Jiang, J. Polastre, and D. Culler. Exploiting the capture effect for collision detection and recovery. *IEEE EmNetS-II*, pages 45–52, 2005.

[20] J. Xiong and K. Jamieson. Towards fine-grained radio-based indoor location. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, page 13. ACM, 2012.

[21] J. Xiong and K. Jamieson. Arraytrack: a fine-grained indoor location system. In *USENIX NSDI*, page 71. USENIX, 2013.

[22] M. Yoshino, S. Haruyama, and M. Nakagawa. High-accuracy positioning system using visible led lights and image sensor. In *Radio and Wireless Symposium, 2008 IEEE*, pages 439–442. IEEE, 2008.

[23] M. Youssef and A. K. Agrawala. The Horus WLAN location determination system. In *ACM MobiSys*, 2005.

[24] W. Zhang and M. Kavehrad. A 2-d indoor localization system based on visible light led. In *Photonics Society Summer Topical Meeting Series, 2012 IEEE*, pages 80–81. IEEE, 2012.

Enabling Bit-by-Bit Backscatter Communication in Severe Energy Harvesting Environments

Pengyu Zhang, Deepak Ganesan
{*pyzhang, dganesan*}*@cs.umass.edu*
University of Massachusetts Amherst

Abstract

Micro-powered wireless sensors present new challenges due to the severe harvesting conditions under which they need to operate and their tiny energy reservoirs. However, existing low-power network stacks make a slew of design choices that limit the ability to scale down to such environments. We address these issues with QuarkNet, a backscatter-based network stack that is designed to enable continuous communication even if there is only enough harvested energy to transmit a few bits at a time while simultaneously optimizing throughput across a network of micro-powered devices. We design and implement QuarkNet on a software radio based RFID reader and the UMass Moo platform, and show that QuarkNet increases the communication distance by 3.5× over Dewdrop, 9× over Buzz, and is within 96% of the upper bound of achievable range. QuarkNet also improves the communication throughput by 10.5× over EPC Gen 2, 5.8× over Dewdrop, and 3.3× over Flit for tag-to-reader communication and by 1.5× over EPC Gen 2 for reader-to-tag communication.

1 Introduction

The idea of networks of perpetual self-powered sensing, communication and actuation devices that can fly in swarms, swim through the bloodstream, and navigate through pipes and debris has propelled the imagination of science fiction writers for decades, but reality is finally catching up. While practical instantiations of self-powered devices have largely been limited to RFID tags, a new generation of micro-powered devices promises to go beyond simple identification towards computation, sensing, and actuation. Among the key technology trends enabling this vision are advances in micro-harvesters that scavenge energy from light, electro-magnetic waves, vibrations, temperature, and other sources [7]. Such micro-harvesters enable platforms to cut their reliance on stored energy in batteries, thereby enabling true miniaturization and perpetual operation [24, 25].

While micro-powered devices present an exciting opportunity, they present tremendous challenges due to the amount of energy they harvest and the sizes of their energy reservoirs. The amount of harvested power using a micro-energy harvester is of the order of *nano*Watts to μWatts, which is three to six orders of magnitude lower than the average power draw of a Mote. At first glance, this seems to suggest that if we wait long enough, the device can trickle charge to accumulate sufficient energy to operate similar to a battery-powered device. But there are three problems. First, long delays before performing useful work are often unacceptable, particularly for continuous sensing and communication. Second, the voltage from the incoming energy source is often low, therefore accumulating energy into an energy reservoir requires boosting voltage which is wasteful compared to incoming energy (imagine pumping water up a hill to store for future use). Third, micro-powered platforms often have small energy reservoirs to reduce form-factor. For example, the Intel WISP [5] and Michigan Micro Mote (M^3) [15] have energy reservoirs that are 4 – 6 orders of magnitude smaller than a coin cell respectively.

The dual limitations of low harvesting rates and tiny energy reservoirs have profound implications on the design of a network stack for micro-powered devices. Every communication task needs to be small enough to fit within the available energy in the reservoir. Enabling communication despite such minuscule energy budgets is akin to working on a micro-sculpture — optimizations at the granularity of individual instructions, bits, on-off transitions, and analog-to-digital conversions are needed. To compound matters, small short-term variations in harvesting conditions that typically would be smoothed out by a larger energy reservoir begin to impact system operation, and can cause an order of magnitude variation in available energy for a task.

These challenges are not addressed by existing protocols such as EPC Gen 2. RFID tags operate solely on continuous harvested power without buffering energy,

therefore EPC Gen 2 assumes a regime where the tag either has enough power to operate continuously, or not at all. In contrast, micro-powered devices can buffer energy, thereby enabling operation in regimes where there is insufficient power to operate continuously, but enough power to operate intermittently.

Recent systems such as MementOS [18] and Dewdrop [8] tackle this problem in different ways. Both these systems use backscatter similar to RFIDs, but the challenge is fitting the communication stack within the energy budget. MementOS introduces checkpoints within computation tasks such that it can recover from outages and continue execution. Dewdrop continually adapts task execution to harvesting conditions such that the efficiency of execution is optimized. To evaluate the ability of these systems to scale down, we consider two harvesting conditions — strong light (2000 lux) and natural indoor light (200 lux), both of which should, in principle, provide enough energy to operate a micro-powered sensor. But while both Mementos and Dewdrop operate under strong light, they are inoperable under natural light.

The inability of current systems to scale-down illustrates the central challenge in designing a network stack for micro-powered devices. A wireless network stack involves a variety of tasks that are simply too large to fit into the extreme energy constraints of this regime. Even the core primitive of a network stack — packet transfer — can involve hundreds of instructions and bits. In this work we ask the following question — what are the general principles that we, as systems designers, should use to enable these micro-powered platforms to communicate continuously despite trickles of energy, tiny energy reservoirs, and dynamic harvesting conditions?

We present QuarkNet, a network stack that embodies a simple but powerful abstraction — by fragmenting a backscatter network stack into its smallest atomic units, we can enable the system to scale down to resource-impoverished regimes. The fundamental building block of QuarkNet is the ability to dynamically fragment a larger packet transfer into μframes that can be as small as a single bit under severe energy constraints, and as large as the whole packet when sufficient energy is available. On top of this abstraction, we design a variety of innovative techniques to handle dynamic frames that can be abruptly terminated in low energy settings, maximize throughput by tracking harvesting dynamics in a low-overhead manner, interleave μframes across nodes to maximize throughput despite different harvesting rates, and minimize overhead across the entire stack.

Our results on a USRP reader and Moo nodes show that:

- The maximum communication distance achieved by QuarkNet is 21 feet, 3.5× longer than Dewdrop and 4.2× longer than EPC ID transfer. QuarkNet achieves close to the maximum achievable range, beyond which decoding even a single bit fails.
- The minimum illuminance required for QuarkNet to operate is 150 lux, which is 13× lower than the 2000 lux requirement of 12 byte EPC ID transfer. This suggests that μframe can operate when a device is powered by natural indoor illuminance, dramatically increasing utility of micro-powered devices for practical deployments.
- The throughput of QuarkNet for node to reader transfer is 18 kbps, 10.5× higher than EPC Gen 2, 5.8× higher than Dewdrop, and 3.3× higher than Flit. For reader to node transfer, we obtain throughput of 1.5 kbps, 2× higher than a battery-assisted device which uses the EPC Gen 2 write command.
- When ten nodes transmit simultaneously to a reader, we achieve a throughput of 16.5 kbps as a result of variability-aware scheduling and interleaving of μframes, which is 5.4× higher than the throughput when devices are inventoried individually. Flit and EPC Gen 2 obtain zero throughput in this case.

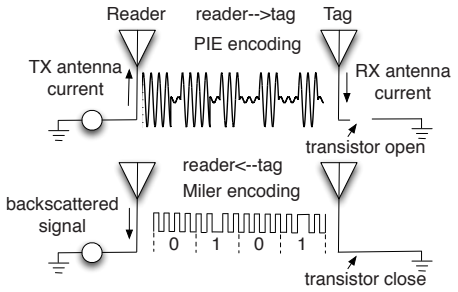


Figure 1: Backscatter signaling at PHY.

2 Case for μframes

A backscatter radio is designed to both provide power to a passive device as well as to enable communication. As shown in Figure 1, the reader provides a carrier wave, which can be reflected by a passive device back to the reader with its own information bits. This makes backscatter a considerably more energy-efficient communication mechanism compared to active radios, and ideally suited to the constraints of micro-powered devices. The Intel WISP [5] and UMass Moo [27] are examples of backscatter-enabled sensor platforms.

Despite the energy benefits of backscatter radios, existing network stacks achieve only short communication range and low throughput. We make an empiric argument these limitations are, in part, due to the design of the network stack. To do this, we compare the range and throughput of existing network stacks versus achievable performance. Our experiment uses a UMass Moo [27]

Table 1: EPC Gen 2 vs Achievable Performance.

	Range(ft)	Throughput(kbps)	SNR(dB)
Gen 2	3.6±0.8	3.6±0.3	9.6±1
Optimal	18.6±3.3	21.7±3.7	6.9±0.9

and a USRP reader [9]. Since combining multiple micro-power sources can enable higher performance, broader operating conditions, and enable wider range of applications, we augment the Moo with a small solar panel [11, 7, 10]. We vary the distance from the reader by small steps, and at each step, we vary RF power from 17dBm to 26dBm, while not changing the light levels (normal indoor light).

To measure the achievable range, we look at the raw backscattered signal at the reader, and find the distance at which the reader is unable to decode even a single bit. This would be the edge of the communication range for our hardware platform.

Measuring the maximum achievable throughput is harder since it is influenced by several system parameters including voltage at the energy reservoir when communication starts, the length of each transmission unit, and control overheads associated with the protocol. We brute-force search across all possible voltages and packet lengths to find the setting that results in the maximum number of transistor flips at the node. We then convert the transistor flips to a maximum number of bits transmitted using the default Miller-4 encoding scheme, and assume zero control overhead for each packet, which gives us an estimate of the maximum throughput.

Table 1 shows the range and throughput while executing the EPC Gen 2 stack (used in Mementos [18], Dewdrop [8], and Blink [30]) versus achievable limits. We see that the achievable range is 18.6 feet, which is over 5× longer than the communication range of EPC Gen 2. Surprisingly, we find that EPC Gen 2 ceases to operate even when its SNR is 9.6dB, 1.4× higher than the optimal case. Similarly, we see that the achievable throughput is 21.7 kbps, whereas EPC Gen 2 achieves barely 1.7 kbps, an order of magnitude difference.

We now investigate the fundamental factors underlying this performance gap, and outline the core challenges that need to be addressed to bridge the gap.

Challenge 1: Variable energy per transmission A key challenge in designing a backscatter network stack is handling variability in the amount of energy accumulated in the energy reservoir. To understand the reasons, let us look at how micro-powered devices work. As shown in Figure 2, micro-powered devices operate in a sequence of charge-discharge cycles since there is too little energy to continually operate the device. The device sleeps for a short period during which it harvests energy and charges a small energy reservoir, and then wakes up and transmits

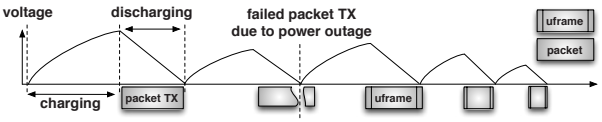


Figure 2: Energy harvesting systems.

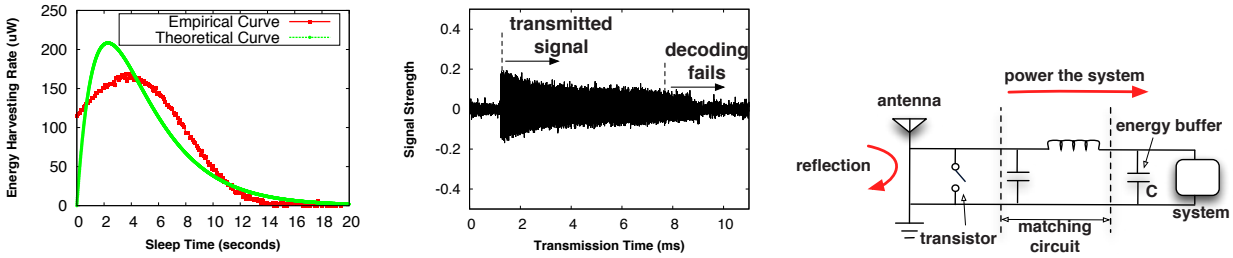
a packet during which the reservoir discharges.

There are several reasons why it is difficult to anticipate how much energy will be available in each discharge cycle. First, if harvesting conditions are too low, it is often too expensive to push more energy into a reservoir due to the inefficiencies of stepping up the voltage. As a result, the maximum amount of energy that can be accumulated depends on current harvesting conditions. Second, RF energy harvested by a node depends on how much energy is output by the reader. When a reader is doing nothing, the RF output power is roughly constant. However when a reader is communicating, this RF carrier wave is being modulated which changes the amount of harvested energy. In a multi-node network, the reader is communicating with different nodes, therefore harvesting rates continually vary at each node. Third, even if the node were to wait until it has a certain amount of energy prior to communication, this requires measurement of energy levels using analog-to-digital conversions (ADC). Each ADC operation consumes 327 uJ on the Moo platform [27], which is equal to the energy budget for transferring 27 bits of data. Such overhead is far too substantial on a micro-powered platform.

While choosing a smaller transmission unit might seem like a straightforward solution to this problem, this over-simplifies the design challenge. As the distance between the node and reader increases to the limit of the achievable range in Table 1, the number of bits that can be successfully transmitted reduces. Thus, we need to use frames that may be as small as one or a few bits in size when the energy levels are low, which requires a network stack that can scale down to unprecedented levels. But such scale down often comes at the expense of throughput, which suffers due to the overheads associated with each transmission, including preambles, headers, and hardware transition overheads. To simultaneously optimize throughput, it is important to transmit as large a transmission as is possible given available energy. Thus, the problem faced by a node is that it needs to scale down its transmission unit to the bare minimum under poor harvesting conditions, while scaling up to improve throughput when the conditions allow.

Challenge 2: Variable harvesting rate

The energy harvesting rate has significant impact on the communication throughput, since higher harvesting rate means that more energy can be used for data transfer.



(a) Energy harvesting rate curve. (b) Decreasing SNR over time. (c) RF matching circuit.

Figure 3: Factors that impact communication throughput.

While energy harvesting rate might seem like a characteristic of the harvesting source, system parameters have a surprisingly high impact. Figure 3(a) shows the empirically measured harvesting rate as we vary the amount of time for which the node replenishes energy between two transmissions. The results are counter-intuitive — while one might expect more energy to be harvested over time, the harvesting rate drops to zero for longer sleep durations.

This observation can be explained analytically by looking at how capacitors buffer energy. The charging process of a capacitor follows its charging equation $V = V_{max}(1 - e^{-t_s/\tau})$, where t_s is the sleep time, τ is the RC circuit time constant, and V_{max} is the maximum voltage to which the capacitor can be charged under the current harvesting conditions. Its energy harvesting rate follows the equation: $H = C \times V_{max}^2 \times \tau^{-1}(1 - e^{-t_s/\tau})e^{-t_s/\tau}$. When the harvesting conditions are constant (i.e. V_{max} and τ are fixed), H is a concave function of t_s, which is shown both analytically and empirically in Figure 3(a). When harvesting conditions change, both V_{max} and τ change, therefore the maximum operating point changes as well. Thus, to optimize throughput, it is important to adapt to current harvesting conditions, and continually track the maximum harvesting point.

One factor that should not be overlooked is keeping the overhead of adaptation low. Most methods to track the charging rate of batteries and capacitors use analog-to-digital conversions to obtain the voltage at the energy reservoir. This overhead is minuscule for most platforms, but a significant part of the harvested energy in our case. Thus, it is important to minimize such overheads while adapting to harvesting conditions.

Challenge 3: Time-decaying SNR A peculiar aspect of backscatter communication is that the signal to noise ratio (SNR) of the received signal at the reader degrades steadily as the size of the transmission unit increases. Figure 3(b) shows that the signal strength of a node response decreases gradually from 0.18 at 1.5ms to 0.05 at 8ms during the transmission process. While decoding the initial part of the transmission is straightforward due to high SNR, it becomes much more challenging after about 8ms since the SNR is too low for reliable decoding, resulting in packet losses.

In order to understand why this happens, let us look at how a backscatter radio works. A backscatter radio provides power to a passive device and enables communication. The reader provides a carrier wave, which can be reflected by a passive device back to the reader with its own information bits. The modulation is achieved by toggling the state of the transistor of a backscatter device shown in Figure 3(c). Since the same RF power source is shared by different system components, some fraction of the incoming power is used to operate the micro-powered device while the rest is reflected back to the reader for communication. The exact fraction depends on the state of the energy reservoir C and the state of the matching circuit, which is designed to charge the energy reservoir C when the voltage is low. Therefore, when the transmission begins, C is fully charged, the antenna resistance is mismatched with the resistance of other hardware components of the system. As a result, most of the incoming power will be reflected back to the reader, which receives a strong signal that can be easily decoded. As the transfer progresses, C slowly discharges, and the antenna resistance matches the resistance of the system load. Therefore, most of the incoming power is harvested to operate the system, and less RF power is reflected. This leads to decreased backscatter signal strength at the reader, and consequently, packet losses. Thus, to ensure that packets are received successfully, the tag needs to adapt the size of each packet such that the SNR at the tail of the packet is higher than the minimum decoding requirement.

Challenge 4: Energy-induced reader to node losses While time-decaying SNR only presents a problem when a node communicates with a reader, reader to node communication presents other challenges. The central issue is that that the energy level on the receiving node might dip below the low watermark at any point during the reception, at which point the node has to shut off its RF circuit and go to sleep to recharge. The reader, however, does not know that the node has gone to sleep, and only

realizes this fact after a timeout.

While such losses can be attributed to small energy harvesting variations at longer ranges, we observed to our surprise that such losses occur even when a tag is placed relatively close to the reader — 40% losses at 2 ft. The reason for this behavior is that data transfer from the reader to tag comes at the expense of RF power being transmitted to the tag. Since the reader is actively transmitting to the tag, the carrier wave from the reader to tag is intermittent, causing substantial variations in RF energy harvesting and consequently variations in energy levels at the tag.

The energy dynamics at the tag makes it difficult to use reader-side estimation to identify the best transmission unit to communicate with a tag. In addition, explicitly providing information to the reader about the current energy level has considerable overhead while not being robust to dynamics. Thus, the challenge we face is that the reader needs to have a way of knowing the instantaneous energy state at the tag, and detecting its shut-off point without using cumbersome protocol-level mechanisms to enable this information exchange.

3 Fragmenting packets into μframes

At the heart of QuarkNet is a simple hypothesis — by breaking down packet transmission into its smallest atomic units, which we refer to as μframes, we can enable the system to scale down to severely limited harvesting regimes. We address the challenges in enabling such extreme fragmentation both for node-to-reader and reader-to-node communication.

3.1 Fragmentation at bit boundaries

The first question we ask is: what are the practical considerations that determine how we can dynamically fragment a logical transmission unit (packet) into μframes? Ideally, we would want to insert fragment boundaries at arbitrary positions within a packet so that we can make μframes as small or large as needed, however, this makes decoding extremely error-prone.

To understand where to place fragmentation boundaries, we need to give some more detail about how backscatter modulation works. Figure 4 shows a sequence of backscatter pulses that compose bits in a packet. Backscatter modulation uses On-Off-Keying (OOK), therefore each bit is composed of a sequence of on and off pulses. As can be seen, the template for a '0' pulse and '1' pulse differ only slightly in the phase information of the pulses within the bit.

The key observation is that placing boundaries at certain points in a packet can be done without disrupting the phase information required for decoding, whereas other

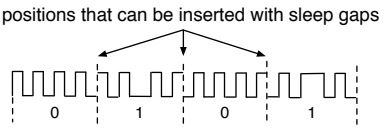


Figure 4: Sleep gaps can be inserted into backscatter pulses at various position (lines with dots).

boundaries would disrupt decoding. For example, suppose that a fragment boundary is inserted between two adjacent bits, the phase information of each bit is maintained, thereby not impacting the ability to match the template to the bit. On the other hand, suppose that a fragment boundary is inserted within a single bit, the phase information within the bit is disrupted, thereby causing a mismatch at the decoder between received bit pulses and its template.

This leads us to a general principle for fragmenting a packet into μframes — μframe boundaries can be inserted between bits but not within a bit. The ability to fragment at any bit boundary gives us the requisite combination of fine-grained fragmentation as well as low decoding error.

3.2 Tuning inter-μframe gap

We now have a method for fine-grained fragmentation of larger packets, but how do we use this to dynamically fragment packets? How do we decide the length of each μframe and the sleep gap between μframes where the node replenishes energy?

We first answer this question for node-to-reader communication. In this case, we need to address two of the challenges discussed in §2: a) how to optimize throughput by operating at the optimal harvesting rate, and b) how to ensure the tail of each μframe transmitted from a node has sufficiently high SNR to be decoded at the reader.

Gradient descent algorithm As can be seen in Figure 3(a), the harvesting rate curve is a concave function of the gap between μframes (under constant harvesting conditions). A fast and effective method for converging to the optimum of a concave function is to use gradient descent [2]. The gradient descent algorithm works as follows: first, we start with an initial guess about the optimal sleep gap. Second, we compute the gradient at this point, and look for the direction of the positive gradient. Third, we take a step along the direction of the positive gradient with step size proportional to the gradient. We repeat this process until convergence (i.e. step is smaller than a threshold). The algorithm takes large steps when the gradient is steep (i.e. point is far from optimal), and small as the gradient reduces (i.e. point is near optimal).

What if the harvesting conditions change and the curve itself shifts to create a new optimal harvesting point? Our gradient descent-based sleep gap adaptation algorithm operates continually — once it converges to the optimal, it periodically probes the gradient at the current optimal, and moves along the positive gradient if the optimal harvesting rate changes. In this manner, the algorithm seamlessly adapts to such dynamics.

Handling time-varying SNR We need to add another constraint to to the gradient descent algorithm — the SNR at the tail of the frame should be higher than the decoding threshold at the reader, otherwise the frame cannot be decoded. This constraint is easy to add since it simply translates to a bound on the maximum length of the inter-μframe gap. Since the length of the gap directly impacts the length of the μframe, capping the inter-μframe gap ensures that the length of each μframe is lower than the decoding threshold. The only change to the gradient descent algorithm is that a step cannot exceed the maximum inter-μframe as determined by the SNR constraint.

Duty-cycling the radio One important aspect of the inter-μframe gap is that we shut off the node's RF circuit for this length of time. In a multi-node environment, the reader is constantly talking to other nodes, so leaving the RF circuit on results in substantial reception overhead since backscatter is a broadcast-based protocol, and wakes up every node that has its radio circuit turned on. To avoid these costs, we turn off the RF circuit during the recharge cycle. Once the node has slept for the intended duration, it switches on its RF circuit. One side-effect of our decision to turn off the RF circuit during gaps is that the reader now has to be more careful to avoid transmitting to a node or scheduling a node for transmission while it is inactive. We return to this question in §4.2.

3.3 Remote interrupts

We now turn to μframe adaptation for communication from a reader to a node. As described in §2, the key challenge is that the reader cannot detect when a node's energy level drops below a low watermark, and it should stop transmitting. Similarly, once a node has gone to sleep, a reader does not know when it will wake up for the next μframe. Given these constraints, how can we enable reader-to-node communication?

Estimating μframe length Our idea is to use a remote interruption mechanism, where a node issues an in-band interrupt during reader transmission, and informs the reader that it has reached a low-energy state. This remote interrupt is generated by toggling its transistor while receiving the current frame. In other words, the remote interrupt is a signal that is overlaid on the same time-slot and frequency signal as the message from the reader to node.

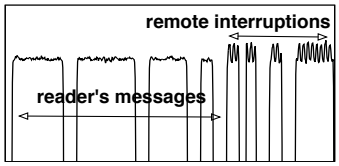


Figure 5: In-band remote interruptions from nodes.

How can the reader decode an in-band interrupt from the node? The key insight is that the reader modulates the carrier by toggling the carrier wave whereas the node communicates back to the reader by changing the amplitude of the backscattered signal. In other words, both can occur simultaneously! Thus, when the reader is sending an ON pulse, the amplitude of the backscattered signal that it receives depends on whether the state of the transistor at the node is ON or OFF — the amplitude is higher when the node's transistor is ON and lower when it is OFF. When the carrier is OFF at the reader, then the state of the node's transistor does not matter since there is no backscattered signal. The reader can detect the remote interrupt by looking for a large signal variance in the carrier wave when the reader has the carrier wave turned on.

Figure 5 shows an example signal where toggling the transistor causes a large variance on the carrier wave, which is monitored by a reader and can be identified by tracking the signal variance within a reader pulse. However, the signal variance is detected only when the carrier wave is on. As shown in the figure, a reader cannot observe the large signal variance when the carrier wave is off. Fortunately, the carrier wave is on for 50% of the time when the reader transmits 0s and 75% for 1s. Thus, as long as a remote interrupt is longer than 50% of the length of a '0' bit from a reader, it can reliably detect the interrupt and pause its transfer.

Finally, an auxiliary benefit of the remote interrupt is that it acts as an inexpensive μframe ACK from the node, which obviates the need for more explicit protocol-level mechanisms and reduces our overhead.

One limitation of our current design is that it is not robust to noise spikes in the frequency band. Such spikes can occur because of multiple readers transmitting to nodes since backscatter is a broadcast medium and reader-to-node communication has to be serialized. Robustness against external interference could be improved by making the remote interrupt longer and encoding the signal, but we do not do this in our current implementation.

Estimating inter-μframe gap We now have a way for the node to interrupt a reader when it needs to replenish

energy, but how long should the reader wait before initiating the next μframe transfer? Clearly, this duration should be at least as long as the inter-μframe gap that the node is using, otherwise the reader might be trying to communicate to a node that has its RF circuit turned off. We address this by using a simple probing-based approach at the reader — for each μframe gap that the reader selects, it knows whether the frame was received or not by checking the presence of a remote interrupt. If no remote interrupt is received, the reader knows the node does not receive the frame properly. The reader continually adjusts the gap to minimize missed frames at the node.

4 QuarkNet for multi-node networks

So far, we have focused on communication between a single node and reader. We now turn to the case where there are several nodes in the vicinity of a reader. The key difference between a single node and multi-node setting is that in the former, the reader stays idle during times when the node is asleep to replenish energy, whereas in the latter, these inter-μframe intervals present an opportunity to schedule another node's μframe transfer, thereby ensuring that throughput is maximized.

4.1 Design Options

Before launching into the details of our design, lets step back and look at the design options. Co-ordination mechanisms for backscatter networks are more restrictive than typical active radio-based networks for two reasons: a) nodes cannot overhear each other's transfer, hence carrier sense-based approaches are infeasible, and b) the stringent resource constraints of nodes render approaches that require complex coding and synchronization infeasible. As a result, existing proposals have focused on two classes of techniques — EPC Gen 2 and variants which use a sequence of random-access slots, and rateless transfer where nodes transfer concurrently, and the reader simultaneously and successively decodes all transmissions.

While the deficiencies of EPC Gen 2 for severely energy constrained regimes have been detailed earlier in this paper, other alternatives and enhancements are surprisingly poor in dealing with this regime as well. In particular, consider two prominent recent techniques — Flit [12] and Buzz [23]. Our earlier work, Flit, re-purposes EPC Gen 2 slots for bulk transfer, thereby amortizing overhead, but it assumes that nodes are able to sustain a long stream of transfer, which we realized was not the case in severe harvesting conditions. Buzz uses rateless codes, but in-order to get these codes to work, it has to use synchronous single-bit slots across nodes. Each single-bit slot incurs substantial overhead due to slot indicators, and turning on and off the radio, which dramatically impacts performance. Given that existing approaches are not well-suited to our nodes, the question is what protocol to use for co-ordinating nodes.

4.2 Variability-aware node scheduling

Our scheduler is designed to interleave μframes from different nodes, thereby fully utilizing the inter-μframe gaps. The reader divides time into variable-sized μslots, during which it explicitly schedules a single node to transmit its μframe. The length of each μslot depends on the size of the μframe — a node-to-reader μframe terminates when the node reaches its low watermark energy level and the reader ACK is received, and a reader-to-node μframe terminates when the node issues a remote interrupt. In both cases, there is a maximum bound on the μframe size to deal with nodes that have plentiful energy.

While the μslot mechanism appears relatively straightforward, the main challenge is handling the fact that nodes turn off their RF circuit when they are asleep. As a result, if a node is scheduled too early by the reader, then it may not be awake to utilize the slot, but if it is scheduled too late, then it is not operating at its maximal harvesting rate.

To handle this, we use a token-based scheduler to deal with the stochastic nature of harvesting conditions, while optimizing throughput. For each node, the scheduler maintains a running estimate of the gap between μslots assigned to a specific node, and whether the μslot resulted in a successful transfer. It uses the estimate to select the inter-μframe gap that ensures a high likelihood of obtaining a node response.

The reader's estimate of the inter-μframe gap is used as input to a token bucket scheduler, which assigns tokens to nodes at a rate inversely proportional to its inter-μframe gap. Once a node has accumulated sufficient tokens, it is likely to have woken up after sleep, therefore the reader places the node into a ready queue since it is ready to be scheduled. The ready nodes can be scheduled based on a suitable metric — for example, the highest throughput node may be selected from the queue to maximize throughput, or the node that has received least slots may be selected for fairness.

5 Implementation

In this section, we describe key implementation details not covered in earlier sections. We use the USRP reader and UMass Moo for our instantiation of QuarkNet. The source code of QuarkNet is available at [3].

5.1 Platforms

USRP Reader QuarkNet is built based on the USRP software radio reader developed by Buettner [9] with a ANT-NA-2CO antenna [4]. We modify the signal processing pipeline to enable variable sized μframe decoding, harvesting-aware tag scheduling, and detection of in-band remote interrupts. The RFX900 USRP RF daughterboard on our platform is only able to transmit 200mW of power, which is 5$\times$ smaller than the 1W of power issued by a commercial reader. Therefore, we attach a 3cm$\times$3cm solar panel to each Moo to increase the amount of harvested energy. The use of hybrid power (RF + ambient) is known to increase range from a reader, which enhances the regimes where backscatter can be used [11].

Backscatter node The UMass Moo is a passive computational RFID that operates in the 902MHz $\sim$ 928MHz band. Perhaps the most challenging aspect of our implementation is debugging under extreme low energy conditions. Traditional methods for debugging embedded systems, such as using JTAG, supply power to the node and change its behavior. Instead, we instrument the Moo to toggle GPIO pins at key points during its execution, and a logic analyzer to record the toggle events. In many cases, however, it is difficult to insert sufficient instrumentation to have visibility while still working with tiny energy harvesting levels. Thus, intuition and experience is particularly important in designing systems for these regimes.

5.2 Trimming Overheads

One important aspect of our system is careful measurement and tuning of all overheads, which impacts our ability to scale-down to severe harvesting conditions.

Radio transition overhead: An important source of overhead is transition times for turning on or off the radio. Fortunately, since hardware timers are responsible for generating the pulses on the backscatter radio, sleep gaps can be inserted by clearing the hardware timers and turning the micro-controller into its low power mode. These operations are inexpensive energy-wise, and consume roughly the same amount of energy as a data frame of size 3 bits. Note that this observation does not hold for more complex active radios — for example, a WiFi radio takes 79.1ms to be on, and 238.1ms to be turned off [13], which is five orders of magnitude higher than the corresponding numbers for a backscatter radio.

Pilot tone: Each backscatter frame can potentially include a pilot tone in addition to the payload. A pilot tone is used when a tag changes its baud rate [19]. We focus on a minimalist protocol that uses a fixed baud rate, therefore we remove the pilot tone. The total overhead per μframe is 6 bits of preamble, in contrast to the 22 bits overhead of EPC Gen 2 (and variants such as Flit [12]).

Probing energy state: As mentioned earlier, analog-to-digital conversions are expensive, and should be avoided while tracking the maximum energy harvesting rate. Our key insight is that rather than measure the voltage on the node, we can leverage the existing low watermark threshold detector that is already present on such nodes. Such a detector is common on harvesting-based sensor platforms for two reasons: a) the platform needs to know when to save state and go to sleep to avoid an outage, and b) the platform needs to know when to wake up after sleep to continue operation. Thus, QuarkNet gets an interrupt both when the voltage crosses above the threshold, as well as when it drops below the threshold, and uses this information as a one-bit proxy for the actual voltage. The voltage threshold is chosen to be 2V which is slightly higher than 1.8V, the minimum voltage required for operating a micro controller. This information is input to a sleep time tracker, which determines how long to wait after crossing the threshold in the upward direction before initiating transfer. Our approach is 100$\times$ less expensive energy-wise than an ADC conversion.

5.3 Protocols and Algorithms

While we do not describe the complete protocol in the interest of space, more details as well as pseudocode for our algorithms can be found in our technical report [28].

6 Evaluation

The evaluation consists of three parts: 1) demonstrating the range and throughput benefit of μframe transmission, 2) benchmarking the performance of our reader-to-node communication, and 3) evaluating the benefit of interleaving μframes from multiple nodes.

6.1 Benefit of μframes

In this section, we validate our claim that the ability to breakdown packets into μframes that can be as small as a single bit can allow us to operate under lower energy conditions and achieve higher operating range. To focus on the effect of the choice of frame size, we strip off overheads (slot indicators, handshakes, etc) for all protocols that we compare.

Minimum operating conditions We look at two harvesters — RF and solar — and ask what is the minimum power requirements for different approaches. We find

that the minimum illuminance required for a 1 bit μframe is 150 lux, which is 13$\times$ lower than the 2000 lux budget of 12 byte packet transmission (the same packet size used by EPC Gen 2, Dewdrop, Flit, etc). We choose 12 byte packet size for EPC Gen 2-based protocols because the 12 byte EPC identifier needs to be transmitted in a singulation phase prior to executing Read or Write commands. Thus, this packet is the bottleneck for operation. To translate from lux to the typical energy available from indoor energy sources, we measure the natural indoor illuminance in 30 positions in an office room. We find that 92% the measured illuminance value is between 150 lux and 1000 lux. This suggests that μframes can operate in most of natural indoor illuminance conditions while a canonical 12 byte transfer scheme can almost never operate under natural indoor light.

The minimum RF power required for a 1 bit μframe is 13dBm, which is 20$\times$ smaller than the 26dBm budget of a 12 byte packet transmission that is the minimum needed for EPC Gen 2 and its variants to operate. Both experiments illustrate the benefits of using tiny μframes.

Increased operational range Our second claim is that we can improve operational range by using μframes. Figure 6 shows the maximum range that is achieved by QuarkNet with 1 bit μframes, EPC Gen 2 with fixed 12 byte packets, Dewdrop with fixed 12 byte packets, Buzz with two slot choices, and a battery-assisted node which represents the best-case scenario. We adjust the RF power of the USRP RFID reader from 17dBm to 25.7dBm, which represents the range of RF power that can be generated by the USRP RFX900 daughterboard.

The results show that the communication range of QuarkNet is longer than other schemes across all RF power levels. At the lowest power level (17.5dBm), μframes do not improve range since the node is not able to decode the reader signal beyond 5ft. But as the RF level increases, the operational range increases dramatically, and is about 4$\times$ longer than EPC Gen 2 at the highest power. In fact, the performance of 1 bit μframe transfer while using harvested energy almost matches the performance of a battery-assisted node, which shows that we are able to reach the ceiling of operational range despite operating on micro-power.

Figure 6 also shows that Buzz [23] performs poorly compared to other schemes. This can be attributed to the fact that each one-bit slot in Buzz has substantial overhead — the reader sends a pulse, followed by one bit from the node, random number generation for deciding whether to transfer in the next slot, and a recharge period. Thus, while Buzz has high range in some settings, the overhead is too high to scale gracefully.

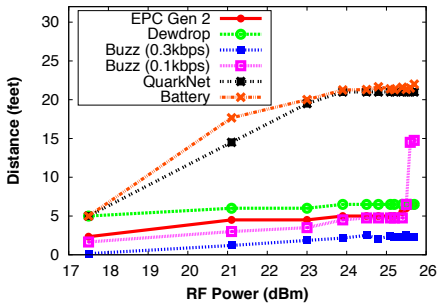


Figure 6: The maximum range achieved by EPC Gen 2, Dewdrop, Buzz, QuarkNet, and a battery assisted node. QuarkNet operates at ranges close to the battery assisted node.

6.2 Benefits of μframe adaptation

We now turn to the benefits of adapting the inter-μframe gap to maximize throughput.

Convergence of gradient descent How well does the gradient-descent algorithm learn the optimal harvesting rate? Figure 7 shows the results for a node placed in three RF+light harvesting combinations that include short and medium range, and low and medium light. In all cases, we see convergence to close to the optimal point — the best inter-μframe gap ranges from 0ms for 350lux at 1 foot since there is enough power to continuously operate the node, 4ms when the node is moved to 6ft, to 12ms when the light conditions dip further. In all cases, our tracking algorithm converges in very few steps (≤ 4).

Throughput benefits We now know that QuarkNet picks close to the optimal harvesting rate, but what are the benefits in terms of throughput? To understand this, we place a node 3 feet from a reader, vary RF power from 17dBm to 26dBm in small steps of 0.3dBm, and inventory the node 2000 times for each scheme. Figure 10(a) shows the throughput achieved by EPC Gen 2, Dewdrop, Flit, and QuarkNet. We find that the throughput achieved by QuarkNet is higher than EPC Gen 2, Dewdrop and Flit across all RF power levels. The average communication throughput of QuarkNet is 18kbps, 10.5$\times$ higher than EPC Gen 2, 5.8$\times$ higher than Dewdrop, and 3.3$\times$ higher than Flit. While the figure does not show Buzz's throughput, note that Figure 6 already showed that this number is low since the per-slot overhead dominates. The lowest slot size we achieved in our implementation of Buzz is 3ms, which means about 0.3kbps throughput.

The previous experiments were done by varying the RF power level. To be sure that these results translate to the case where nodes are placed at different locations in front of a reader, we measure the throughput achieved by EPC Gen 2, Dewdrop, Flit, and QuarkNet at 30 different randomly chosen locations between 2 to 13 ft in front of a reader. Figure 8 shows that the throughput achieved by

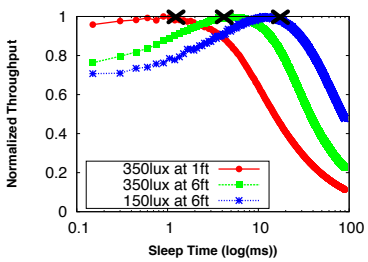


Figure 7: Throughput achieved for different sleep times (inter-μframe gaps). The sleep time chosen by QuarkNet is within 98% of the optimal.

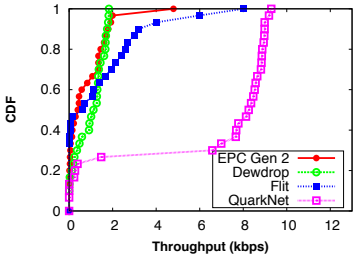


Figure 8: Throughput achieved by EPC Gen 2, Dewdrop, Flit, and QuarkNet across 30 locations. QuarkNet has at least 4.4× higher throughput than other schemes.

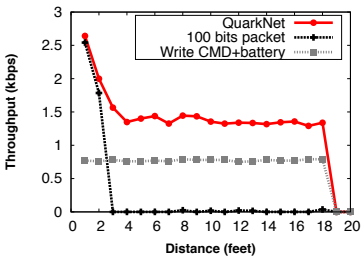
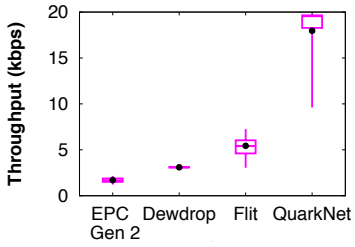
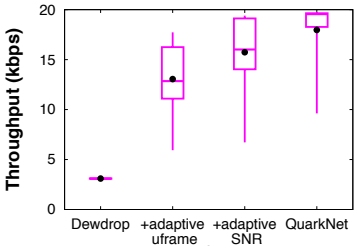


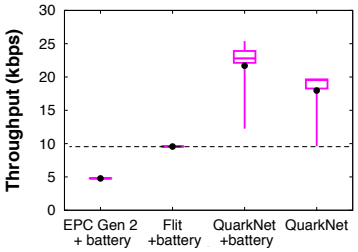
Figure 9: Throughput of reader-to-node communication. QuarkNet has 2× higher throughput than battery-assisted EPC Gen 2 Writes.



(a) QuarkNet vs micro-powered nodes.



(b) QuarkNet vs Dewdrop.



(c) QuarkNet vs battery assisted nodes.

Figure 10: For micro-powered devices, QuarkNet improves throughput by at least 3.3× over all other schemes, and even performs better than battery assisted nodes. The benefit comes from reducing overhead, and adapting μframe sizes to energy and SNR.

QuarkNet is higher than the other three schemes across all locations. The average throughput of QuarkNet is 7.8× higher than EPC Gen 2, 6.4× higher than Dewdrop, and 4.4× higher than Flit. In particular, QuarkNet continues to operate in many locations where other schemes cease to operate.

Breaking down the benefits QuarkNet has a variety of optimizations including reduced overheads, variable-sized μframes, and SNR adaptation. To understand the contributions of these techniques to throughput, we start with the default implementation of Dewdrop, and add one optimization at a time: a) Dewdrop + adaptive frame, which includes variable-length μframes, and b) Dewdrop + SNR adaptation which includes the SNR adaptation. Figure 10(b) shows the throughput achieved by the three variants of Dewdrop vs QuarkNet. Clearly, each of the optimizations plays a major role in the throughput improvements observed by QuarkNet. The average communication throughput of μframe is 18kbps, 5.79× higher than Dewdrop, 1.37× higher than Dewdrop with adaptive μframes, and 1.14× higher than the case when SNR adaptation is included. In the final step, we replace Dewdrop's adaptation algorithm with our version that eliminates ADC conversions to get QuarkNet.

QuarkNet vs battery-assisted alternatives Another interesting question is how QuarkNet performs when compared to battery-assisted versions of the other protocols (excluding Dewdrop + battery, which is identical to EPC Gen 2 + battery). Some protocols, such as Flit [12], improve in performance when there is more energy since there is more opportunity for bulk transfer. Would these outperform QuarkNet in battery-assisted scenarios? Figure 10(c) shows that throughput achieved by QuarkNet is consistently better. The average throughput of QuarkNet is 18kbps, 3.75× higher than EPC Gen 2 + battery, and 1.87× higher than Flit + battery. This result shows the benefit of reducing per-frame overheads in QuarkNet.

6.3 Reader-to-node communication

We now turn to an evaluation of reader-to-node communication. We begin by looking at the effectiveness of remote interrupts. We find that remote interrupts are extremely reliable — the reader detects remote interrupts with 100% accuracy across all distances where the node can communicate with the reader, and detection rate directly drops to 0% at roughly 19 – 20 feet where the node cannot detect the signal sent by the reader. While the accuracy will degrade under external interference, we plan to extend remote interruption to include encoded bits to improve robustness.

Next, we look at the throughput of reader-to-node

communication when a node is placed at different distances from the reader. Figure 9 shows that the throughput achieved by QuarkNet is always higher than fixed 100 bit transfer across all distances. (We chose 100 bits instead of 12 bytes because of the slower baud rate of the reader-to-node link, as a result of which 12 byte transfer ceases to operate even when the node is deployed 1 feet from the reader.) The throughput of QuarkNet is higher than even a battery-assisted EPC Gen 2 node. This shows that the benefit of variable sized μframes is substantial even for reader-to-node communication.

One trend in the graph that requires a bit more explanation is the fact that throughput decreases rapidly when the node is close to the reader (less than 4 feet), and plateaus until about 18 ft after which it quickly drops to zero. This is because RF-harvesting only works until 4ft (because of the limitations of the USRP reader), and beyond this distance, indoor light harvesting plays the dominant role.

6.4 Evaluating the QuarkNet MAC layer

We now turn to the evaluation of our MAC layer that includes all components of the protocol including various co-ordination overheads, frame interleaving, and scheduling. Figure 11 shows the communication throughput when we deploy 10 nodes in front of the reader and adjust the RF power from 17dBm to 26dBm. We use a throughput-maximizing scheduling policy in this experiment. For each RF power level, we plot the averaged throughput across the ten nodes and the confidence interval when they are scheduled in an interleaved manner and when they are inventoried individually. The throughput achieved by other MAC layer designs — EPC Gen 2 and Flit — are close to zero, so we do not plot them.

We find that even at the lowest RF power level, almost all nodes get to transmit data to the reader, and the average throughput steadily increases with higher RF power. In addition, the throughput achieved by interleaving the 10 nodes is 5.4× higher than the throughput when those 10 nodes are inventoried individually. These results show that our algorithm scales well across a wide dynamic range of harvesting conditions, and uses gaps between μframes efficiently.

6.5 Microbenchmarks

Table 2 shows the overhead incurred by different components of QuarkNet. The biggest system overhead is the switch from inactive mode to transmission mode (47.5 us), to configure several registers associated with transmission, such as the hardware timer register and data register. The overhead of the entire μframe size and inter-

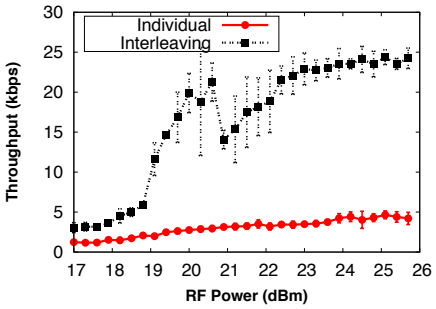


Figure 11: Throughput of 10 nodes is 5.4× higher when interleaved than when individually inventoried.

Table 2: Overhead of μframe transmission.

System overhead (us)		μframe overhead (us)	
TX to inactive	9.9	interrupt config	10.58
inactive to TX	47.5	handle interrupt	9.3
RX to TX	4.08	μframe adaptation	24.3
sleep to wakeup	9.83	voltage detection	3

μframe gap adaptation algorithm (47.2us), is comparable to the total system overhead, and 10× smaller than the cost of an ADC conversion. Overall, the results show that our performance tuning measures have substantial benefits — the sum total of these overheads is smaller than the cost of transmitting 7 bits.

7 Discussion

Interoperability with other PHY mechanisms While our work does not explicitly address co-existence of QuarkNet with other physical layer and upper layer mechanisms, many of these can be easily layered above the methods described in this paper. For example, rate adaptation is widely used to adapt to wireless channel conditions, thereby maximizing communication throughput. This method operates at the bit-level, where each bit is composed of several symbols. Such an approach can be layered above QuarkNet, with gaps introduced between bits. Similarly, error correction codes or other encoding mechanisms that reduce bit error rate can be implemented above QuarkNet.

QuarkNets role with evolving technology As micro-harvesters continue to improve in efficiency, one question is whether QuarkNet will continue to remain relevant. We argue that QuarkNet's relevance will increase for two reasons. First, the maximum harvesting rates are fundamentally limited by the physics of the harvesting source and form-factor. For example, RF energy harvesting is limited by the antenna size and the amount power issued by antennas, solar energy harvesting is limited by the panel size and the intensity of illuminance, and thermal energy harvesting is limited by the surface area and

the temperature differential. Even if micro-harvesters become extremely efficient (say upwards of 80%), there is still a small amount of energy available, and systems optimizations similar to QuarkNet are critical to using the energy in an efficient manner. Second, trends in nano-electronics and low-power embedded systems are resulting in sensing and computing platforms that consume only tens of micro-watts of power [1]. These trends will make it possible to design many more micro-power based applications such as implantables and on-body sensors, enhancing the relevance of QuarkNet.

Fragmenting other tasks While our focus in this paper is on fragmenting the network stack, the abstraction of task fragmentation presented by QuarkNet can be potentially used for breaking down other components of a task such as sensing and computation into smaller atomic units. In our position paper [29], we presented preliminary results that demonstrated the ability to fragment an image sensing task such that the entire sensor can operate with a 3cm$\times$3cm solar panel under natural indoor illuminance. However, many questions remain to fully enable such fragmentation, requiring a combination of architectural modifications to the sensing and computing blocks to facilitate fine-grained fragmentation, systems techniques similar to QuarkNet that can take advantage of the fragmentation capability, as well as data processing techniques to enable useful applications over a layer that dynamically fragments sensing tasks.

8 Related Work

We have already discussed Dewdrop, Flit, Buzz, and EPC Gen 2, so we focus on other approaches.

Computational RFIDs (CRFIDs) There has been increasing emphasis on CRFIDs in recent years given its potential for battery-less perpetual sensing. Ambient Backscatter [16] uses the backscatter of FM signals for short-range communication between tags. This is a severely energy limited platform, and could leverage QuarkNet when harvested energy is low. BLINK [30] is a bit-rate and channel adaptation protocol to maximize communication throughput, which can also leverage QuarkNet for performance. [20] introduces a power-optimized waveform which is a new type of multiple-tone carrier and modulation scheme that is designed to improve the read range and power efficiency of charge pump-based passive RFIDs. [21] presents a system architecture for backscatter communication which reaches 100m communication distance at the cost of slow bit rate (10 bits per second). Such techniques are complementary to QuarkNet — each bit transmitted at slow bit rate can be fragmented into several segments where the information within each bit is still preserved. Also of note is MementOS [18], which uses non-volatile flash storage for checkpoints within a task such that the it can continue execution after an outage. Flash checkpointing is useful for outage tolerance but is more than the cost of transmitting an entire EPC Gen 2 packet, hence it has limited utility in our case.

EPC Gen 2 optimizations Much of the work on backscatter communication is specific to EPC Gen 2 tags, for example, better tag density estimation [22], better search protocols to reduce inventorying time [14], better tag collision avoidance [17], more accurate tag identification [26], better recovery from tag collisions [6], and more efficient bit-rate adaptation [30]. None of these tackle the problem of maximizing range and throughput from RFID-scale sensors, which have the ability to offload sensing data back to a reader.

EPC Gen 2 supports tag user memory operations in addition to simple EPC queries including the Read and Write command, however they are second-class citizens in the protocol since the main goal is to inventory tags. As a result, both are inefficient primitives for data transfer from tag to reader or vice-versa. In our experiments, we found that the Read and Write commands simply do not work at all under low energy conditions.

9 Conclusion

In this paper, we present a powerful network stack, QuarkNet, that can enable systems to seamlessly scale down to severe harvesting conditions as well as substantial harvesting dynamics. At the core, our approach deconstructs every packet into μframes, handles dynamics with variable-sized μframes, and maximizes throughput via low-cost adaptation algorithms and interleaving of μframes. Results show that QuarkNet provides substantial benefits in pushing the limits of micro-powered devices, and allow them to perform useful work under more extreme environments than previously imagined possible. Our network stack tolerates such conditions, thus makes it valuable to a wide range of emerging micro-powered embedded systems and applications.

Acknowledgements

This research was partially funded by NSF grants CNS-1218586, CNS-1217606, and CNS-1239341. We thank our shepherd Shyam Gollakota for providing guidance in preparing our final draft and the anonymous reviewers for their insightful comments. We also thank Michael Buettner for extensive and insightful comments, Jeremy Gummeson for help in Flit implementation, and Jue Wang for assistance with Buzz.

References

[1] Advanced Self-Powered Systems of Integrated Sensors and Technologies. http://assist.ncsu.edu.

[2] Gradient descent. http://en.wikipedia.org/wiki/Gradient_descent.

[3] Source code of QuarkNet. https://github.com/pengyuzhang/QuarkNet.

[4] ThingMagic Bistatic Antenna. http://buyrfid.com/.

[5] WISP: Wireless Identification and Sensing Platform. http://seattle.intel-research.net/wisp/.

[6] C. Angerer, R. Langwieser, and M. Rupp. Rfid reader receivers for physical layer collision recovery. *IEEE Transactions on Communications*, 2010.

[7] S. Bandyopadhyay and A. Chandrakasan. Platform architecture for solar, thermal and vibration energy combining with mppt and single inductor. In *VLSI Circuits (VLSIC), 2011 Symposium on*, pages 238–239. IEEE, 2011.

[8] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: an energy-aware runtime for computational rfid. In *USENIX NSDI*, 2011.

[9] M. Buettner and D. Wetherall. A software radio-based uhf rfid reader for phy/mac experimentation. In *RFID (RFID), 2011 IEEE International Conference on*, pages 134–141. IEEE, 2011.

[10] M. Gorlatova, P. Kinget, I. Kymissis, D. Rubenstein, X. Wang, and G. Zussman. Challenge: ultra-low-power energy-harvesting active networked tags (enhants). In *Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 253–260. ACM, 2009.

[11] J. Gummeson, S. S. Clark, K. Fu, and D. Ganesan. On the limits of effective hybrid micro-energy harvesting on mobile crfid sensors. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 195–208. ACM, 2010.

[12] J. Gummeson, P. Zhang, and D. Ganesan. Flit: A bulk transmission protocol for rfid-scale sensors. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 71–84. ACM, 2012.

[13] 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 the 10th international conference on Mobile systems, applications, and services*, pages 225–238. ACM, 2012.

[14] C. Law, K. Lee, and K. Siu. Efficient memoryless protocol for tag identification. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 75–84. ACM, 2000.

[15] Y. Lee, G. Kim, S. Bang, Y. Kim, I. Lee, P. Dutta, D. Sylvester, and D. Blaauw. A modular 1mm¡ sup¿ 3¡/sup¿ die-stacked sensing platform with optical communication and multi-modal energy harvesting. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 402–404. IEEE, 2012.

[16] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient backscatter: wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 39–50. ACM, 2013.

[17] V. Namboodiri and L. Gao. Energy-aware tag anticollision protocols for rfid systems. *Mobile Computing, IEEE Transactions on*, 9(1):44–59, 2010.

[18] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on rfid-scale devices. *ACM SIGPLAN Notices*, 47(4):159–170, 2012.

[19] K. E. Sundstrom, S. A. Cooper, A. Sarajedini, A. Esterberg, T. E. Humes, and C. J. Diorio. Rfid reader systems detecting pilot tone, Jan. 8 2013. US Patent 8,350,665.

[20] M. S. Trotter, J. D. Griffin, and G. D. Durgin. Power-optimized waveforms for improving the range and reliability of rfid systems. In *RFID, 2009 IEEE International Conference on*, pages 80–87. IEEE, 2009.

[21] G. Vannucci, A. Bletsas, and D. Leigh. A software-defined radio system for backscatter sensor networks. *Wireless Communications, IEEE Transactions on*, 7(6):2170–2179, 2008.

[22] H. Vogt. Efficient object identification with passive rfid tags. *Pervasive Computing*, pages 98–113, 2002.

[23] J. Wang, H. Hassanieh, D. Katabi, and P. Indyk. Efficient and reliable low-power backscatter networks. *ACM SIGCOMM Computer Communication Review*, 42(4):61–72, 2012.

[24] A. Yakovlev, S. Kim, and A. Poon. Implantable biomedical devices: wireless powering and communication. *Communications Magazine, IEEE*, 50(4):152–159, 2012.

[25] D. Yeager, J. Holleman, R. Prasad, J. Smith, and B. Otis. Neuralwisp: A wirelessly powered neural interface with 1-m range. *Biomedical Circuits and Systems, IEEE Transactions on*, 3(6):379–387, 2009.

[26] D. Zanetti et al. Physical-layer identification of uhf rfid tags. In *ACM MobiCom*, 2010.

[27] H. Zhang, J. Gummeson, B. Ransford, and K. Fu. Moo: A batteryless computational rfid and sensing platform. Technical report, Tech. Rep. UM-CS-2011-020, UMass Amherst Department of Computer Science, 2011.

[28] P. Zhang and D. Ganesan. Enabling bit-by-bit backscatter communication in severe energy harvesting environments. Technical report, School of Computer Science, UMass Amherst. URL: http://cs.umass.edu/~pyzhang/papers/QuarkNet-tech-report.pdf, 2014.

[29] P. Zhang, D. Ganesan, and B. Lu. Quarkos: Pushing the operating limits of micro-powered sensors. In *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*, pages 7–7. USENIX Association, 2013.

[30] P. Zhang, J. Gummeson, and D. Ganesan. Blink: A high throughput link layer for backscatter communication. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 99–112. ACM, 2012.

Full Duplex MIMO Radios

Dinesh Bharadia
dineshb@stanford.edu
Stanford University

Sachin Katti
skatti@stanford.edu
Stanford University

Abstract

This paper presents the design and implementation of the first in-band full duplex WiFi-PHY based MIMO radios that practically achieve the theoretical doubling of throughput. Our design solves two fundamental challenges associated with MIMO full duplex: complexity and performance. Our design achieves full duplex with a cancellation design whose complexity scales almost linearly with the number of antennas, this complexity is close to the optimal possible. Further we also design novel digital estimation and cancellation algorithms that eliminate almost all interference and achieves the same performance as a single antenna full duplex SISO system, which is again the best possible performance. We prototype our design by building our own analog circuit boards and integrating them with a WiFi-PHY compatible standard WARP software radio implementation. We show experimentally that our design works robustly in noisy indoor environments, and provides close to the expected theoretical doubling of throughput in practice.

1 Introduction

Full duplex radios have garnered significant attention recently in academia and industry [17, 11, 23, 22, 16, 19, 18, 15, 29, 20, 26, 24]. Several efforts are now underway to include full duplex technology in future cellular 5G standards [3], as well as explore applications of the technology in current wireless infrastructure. However these efforts are hampered by the fact that there aren't viable and efficient full duplex designs that can work in conjunction with MIMO. Specifically, no current practical designs are known which can enable one to build a M antenna full duplex MIMO radio that can transmit and receive from all antennas at the same time and double the throughput. The best known prior MIMO full duplex system, MIDU [11] requires $4M$ antennas for building a full duplex M antenna MIMO radio, and even then fails to provide the needed self-interference cancellation for WiFi systems (20 MHz bandwidth) to achieve the expected doubling of throughput.

Recent work has however demonstrated that a single antenna (SISO) full duplex system is practically possible [14]. Specifically, it demonstrates the design and implementation of a cancellation system for a SISO system that completely cancels self-interference to the noise floor and consequently achieves the theoretical doubling of throughput. A natural question therefore is why not just replicate the same design M times to build a MIMO M full duplex radio? After all, a MIMO radio can be conceptually and physically viewed as a collection of M single antenna SISO full duplex radios.

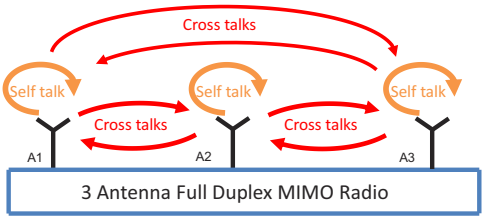


Figure 1: Shows a 3 Antenna MIMO Full Duplex node, with different interference's referred as talk. Every chain sees 2 other cross-talks other than the self-talk.

The challenge is cross-talk interference as seen in Fig. 1. When a full duplex MIMO radio transmits, the transmission from any one of the M antennas (interchangeably referred to as transceiver chains) propagates to the other antenna (chains) and causes a large amount of interference. For the sake of clarity, in this paper we will refer to the self-interference at a receive chain caused by a transmission from the TX-chain with which the receive chain shares an antenna as "self-talk", and the interference from a neighboring TX chain's transmission as "cross-talk". Since MIMO antennas are closely spaced due to size constraints, this cross-talk is extremely strong, almost 75-80dB stronger than the desired signal that is being received on that chain. Consequently, even if we have cancellation circuits and algorithms that cancel every chain's self-talk, there is an extremely strong cross-talk interference that can saturate the receive chain.

A naive solution is to introduce a separate copy of the cancellation circuit and DSP algorithm for each pair of chains that experiences cross-talk. If there are M antennas, then it would imply a total of M^2 circuits and DSP algorithms. In other words complexity grows quadratically with the number of antennas, which is untenable as MIMO systems go towards 4 to 8 antennas. Supporting 16 cancellation circuits and DSP implementations (for 4 antenna MIMO) on even a WiFi AP based form-factor is untenable (our analysis suggested that with the current SISO design we would need 400sq.cm of analog circuit area and a high-end Virtex FPGA that consumes 80W of power to accommodate the DSP computations). Com-

plexity impacts more than space and power consumption, cancellation systems (both analog and digital) need to be tuned continuously to adapt to environmental changes. The time for tuning scales linearly with the complexity, hence it would take M^2 time longer to tune such a design's MIMO self-interference cancellation system. The best known prior algorithm for tuning [14] requires around a millisecond to tune, so we would need 16 ms to tune for a 4 antenna MIMO system which would be untenable even in a slowly changing environment like indoor WiFi (coherence times are on the order of tens of milliseconds), let alone mobile environments such as LTE.

A second problem is performance itself. The key metric is the residual interference left after cancellation at each receive chain, the residual directly translates to decrease in SNR for the desired received signal. As we will show in Sec. 3, even if one could accommodate a quadratic number of circuits and DSP cancellation implementations, the performance degrades linearly with the number of MIMO chains. In other words, the residual interference after cancellation at each receive chain increases linearly with M. This is due to the accumulation of the residual interference from all the cross-talk and self talk cancellation systems. Once again, as MIMO systems scale to support many antennas, this essentially limits the performance gains of full duplex.

This paper presents the design and implementation of a MIMO WiFi full duplex radio. Our M antenna full duplex MIMO radio uses each antenna for simultaneous transmit and receive, i.e., it uses the same number of antennas as a standard half duplex M-antenna MIMO radio unlike prior designs. The design uses slightly more than $M\times$ cancellation circuits and DSP algorithms (w.r.t to SISO full duplex design) to cancel all the self and cross talks. In other words, complexity scales linearly with the number of chains, which is the best performance one could expect. Further, the performance does not degrade linearly with the number of MIMO chains, i.e., the residual interference is the same as the SISO design and does not increase linearly with the number of chains. We prototype our design and integrate it with the off-the-shelf WARP software radios [6] running a stock WiFi baseband and demonstrate experimentally that it achieves close to the theoretical doubling of throughput.

Our design solves the key challenge of efficiently and effectively achieving the MIMO full duplex using two major ideas as follows.

- First, a key insight is that MIMO chains are co-located, i.e., "they share a similar environment". Intuitively, the signals transmitted by two neighboring antennas (separated by a few cm) go through a similar set of reflectors and attenuations in the environment [21]. Cancellation systems are essentially trying to model these distortions, so when we want to model cross-talk, we can reuse the work that has been done for modeling the chain's own self-talk interference. This results in a novel "cascaded" filter structure for cancellation that results in an overall design that has near-linear complexity scaling with the number of MIMO antennas.
- Second, the reason performance degrades linearly with the SISO replication based design is that each of the M independent cancellation algorithms for self-talk and cross-talk at a receive chain produce their own estimation error which add up to the linear degradation. Our key insight here is to leverage the fact that we have M transmitters available that can concurrently send training symbols. Specifically, we design a training preamble for WiFi that allows each receive chain to estimate each of the self-talk and cross-talk channels with an error that is M times lower than the SISO design by combining information from all M training symbols. Consequently, in our design when the estimation errors add up for the self-talk and cross-talk cancellations, *the overall error or residue is the same as a SISO system would have achieved, which is the best one can hope for. Further the algorithms are modular and structured in a way that, if in the future the SISO full duplex design manages to improve its performance even further, the MIMO design in this paper immediately benefits.*

We prototype our design using our own custom designed analog cancellation circuits, and integrate them with novel implementation of our digital cancellation algorithms using off-the-shelf WARP radios [6]. Our experiments demonstrate that in a 3×3 configuration, our system achieves a performance that leaves a negligible 1dB of self-interference after cancellation. We also show that our system achieves a 95% throughput gain over half duplex radios using a standard WiFi compliant OFDM PHY of 20MHz for 802.11n for all different modulations (BPSK, QPSK, 16QAM and 64 QAM) and coding rates of (1/2,2/3,3/4,5/6), supporting three streams for 3×3 MIMO.

2 The Problem

In this section, we describe the nature of interference in a MIMO full duplex radio and then discuss the architectural challenges in designing a cancellation system.

Self-talk or cross talk (or for that matter any transmitted signal) is made up of three major components [7, 12, 4]:

- Linear Signal: This is the signal that the baseband modem wanted to transmit and is then distorted by channel reflections. It's linear because it can be represented as a linear combination of delayed and summed copies of the same signal that arise from environmental multipath reflections.

Power and Interference relative to noise floor of -85 dBm

	Power level in dBm	Cancellation needed in dB
Total TX signal	20	105
Linear component	20	105
Non-linear component	-10	75
Transmitter Noise	-20	65

(a) The different components of the transmitted signal (self-talk) for a typical WiFi radio. The second column tabulates the amount of self-talk cancellation needs to eliminate the corresponding self-talk component to the noise floor.

Power and Interference relative to noise floor of -85 dBm

MIMO FD, Receiver 1	Power in dBm			Cancellation needed (dB)		
	Self-talk	Cross talk 1	Cross talk 2	Self talk	Cross talk 1	Cross talk 2
Overall signal at antenna 1	15	-9	-15	100	76	70
Linear component	15	-9	-15	100	76	70
Non-linear component	-15	-39	-45	70	46	40
Transmitter noise	-25	-49	-55	60	36	30

Cancellation Requirement

MIMO FD, Receiver 1	Self-talk	Cross Talk 1	Cross Talk 2
Analog cancellation	65 dB	41 dB	35 dB
Digital cancellation	35 dB	35 dB	35 dB

(b) Interference components and cancellation requirements for 3 antenna MIMO full duplex. The first table describes the levels of different interference components (linear, non-linear and transmit noise) that make up self-talk and cross-talks at one receiver in a 3 antenna MIMO radio. Cross-talk 1 is from the neighboring antenna and cross-talk 2 is from the farther neighboring antenna. The second table lists the overall cancellation needed, here the values are bumped up by 5dB relative to the first table to ensure that even when the residues left from the self-talk and the two cross-talk cancellations are added up, the overall noise floor does not go up (else it would go up by 5dB if the cancellation requirement for each component did not have a 5dB margin).

Figure 2: Requirement tables

- Non-linear Signal: This is the signal that is generated due to non-linear transformations that the linear signal goes through when it is passed through analog radio components such as mixers, power amplifiers in the transmit chain [28].
- Transmit Noise: This is the noise that is generated by active components in the TX chain such as power amplifiers and local oscillators (we club things such as broadband noise and phase noise into this term for the sake of brevity).

The relative strengths of these components depends on the quality of the radio. Fig. 2a tabulates the strengths of the different components we empirically measured for a commodity 20dBm WiFi SISO radio, and the amount of cancellation needed to eliminate them in a full duplex system. Note that this is a cheap radio widely used in many commercial WiFi devices [4, 6], so we believe this is representative of the WiFi radios in general.

The above analysis is of course true even for a single antenna radio without MIMO, and recent work [14] describes cancellation techniques that eliminate all self-talk. However, what is unique with MIMO is cross-talk. In other words, the interference that results at a receive chain due to a transmission from a neighboring co-located MIMO antenna/chain. In a 3 antenna full duplex MIMO radio, each receiver chain would see two cross-talk signals from the other two antennas as seen in Fig. 1.

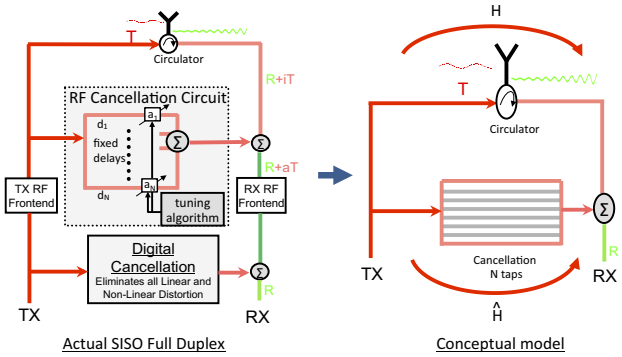


Figure 3: Prior best performing SISO full duplex design. The figure on the right shows an equivalent conceptual filter based view of self-talk cancellation. The filter is parameterized by its complexity, the number of taps. The filter subsumes both analog and digital cancellation.

Cross-talk is slightly weaker than the self-talk generated by the chain's own transmission, but is still quite strong and has all the above three enumerated components. Like the earlier SISO design [14] (as shown in Fig. 3), the transmit noise component of the cross-talk signal has to be canceled in the analog domain, whereas the non-linear and linear components could be canceled in both analog and digital domains. Fig. 2b tabulates the strengths of the various components that make up a cross-talk and self-talk signal in a typical 3-antenna MIMO WiFi radio with 20dBm[1] transmit power (note that the power is divided equally among all three transmitters, so the power out of each antenna is 15dBm).

2.1 Why can't we reuse the SISO full duplex design by replicating it?

At first glance, the MIMO interference cancellation problem looks quite similar to a SISO full duplex problem, only replicated a few times. After all the cross-talk signal that needs to be canceled looks like an attenuated version of a chain's own self-talk signal that the SISO design manages to cancel completely. So why couldn't we replicate the SISO design $M^2 - M$ times for each of the cross-talk signals in a M antenna MIMO radio and be done with it (as shown in Fig. 4)?

To understand the reason this might not work, it will

[1]The FCC specifies that the peak power can be 30 dBm [2]. However OFDM signals have a high PAPR, i.e. the peak power of the output signal is significantly higher than the average power. For WiFi we find that the PAPR is 10dB, so the average power we can use is actually 20dBm.

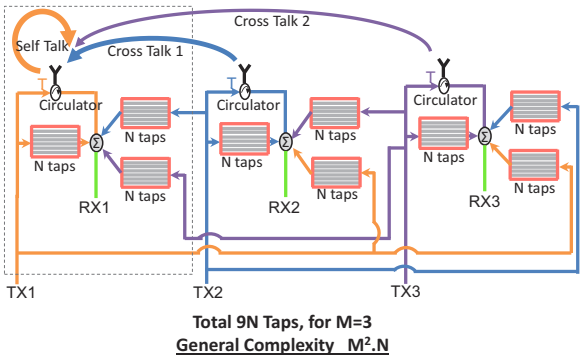


Figure 4: **SISO Replication Based Design**: Shows a 3 antenna full duplex MIMO radio, using nine SISO cancellation circuits (SISO replication design). This design uses in total 9N taps for M=3 assuming each circuit requires N filter taps. In the general case this design would require M^2N for a M antenna full duplex MIMO system.

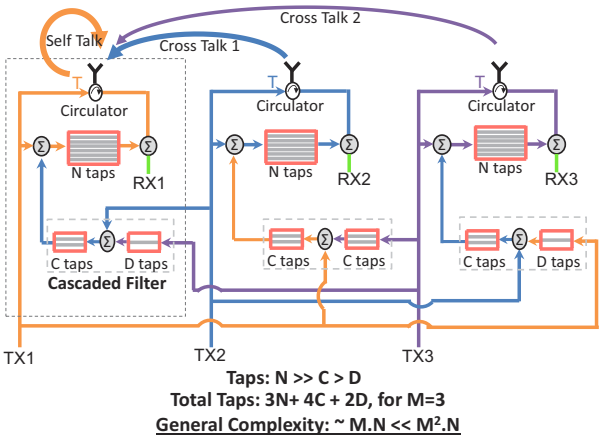


Figure 5: **Cascaded Cancellation Design**: Shows a 3 antenna full duplex MIMO radio design with cascaded filter structure for cancellation. The structure is shown for receiver chain 1 only, but the same structure is repeated for the other chains. For, self-talk cancellation we have N filter taps on every chain. Further we have C and D taps feeding in a cascading fashion at the input of the N tap self-talk cancellation circuit. Notice cross talk 1 is stronger so we need more taps ($C > D$) as compared to cross talk 2. However both C and D are significantly smaller than N.

help to have a conceptual understanding of what a SISO self-talk cancellation system accomplishes. At its core, the self-talk cancellation technique can be thought of as shown in Fig. 3. The input is the baseband signal that is being transmitted, to which transmit noise is added and the combined signal is passed through a linear and non-linear unknown transfer function that captures the distortions introduced by the analog components and the wireless channel and is denoted by H. The cancellation circuits and algorithms are trying to calculate an estimate – $\hat{H}$ – of this unknown transfer function H as accurately as possible (to the tune of 105dB resolution), and then pass a copy of the input baseband transmitted signal and noise through this estimated transfer function $\hat{H}$ to recreate the self-talk and cancel it (shown in Fig. 3). The estimated transfer functions are created using tunable **analog and digital FIR filters**, for example the prior SISO design's analog cancellation circuit requires 12 delay-attenuation taps that each represent a single analog FIR filter tap (refer Fig. 3), and what is being controlled is the weight on each tap (practically this translates to controlling the attenuator on that delay-attenuation analog line). A similar FIR filter structure is used for digital cancellation and the challenge is calculating the weights to use on each of the taps. So the key challenge the SISO self-talk cancellation system is solving is calculating a set of FIR filter weights that can accurately model this unknown and time-varying transfer function.

Consequently, there are two metrics that characterize these estimation circuits and algorithms.

- **Complexity**: can be quantified by the number of filter taps that are used in the implementations that represent the estimated $\hat{H}$. The more taps we need, the more analog circuitry is needed as well as DSP resources in FPGA to implement them. Keeping the number of taps low is important so as to reduce the space and power consumed by analog circuits [10] and DSP logic for FIR implementations (the baseline is the SISO design that requires 12 analog taps and 132 digital FIR taps). To get a sense of the impact, 12 analog taps consume roughly 24sq.cm of board area. A second consequence of complexity is the amount of time it takes us to retune the cancellation when the environment changes (including things such as temperature). The larger the number of taps, the longer it will take to tune since there are more variables to estimated. When cancellation is being tuned, the radio cannot be operated in full duplex mode. Hence tuning time is pure overhead, and needs to be minimized.
- **Estimation error**: A second key metric is estimation error which manifests as residual interference left after cancellation and directly reduces the SNR of the desired received signal. A perfectly accurate cancellation system would leave no residue. The baseline for this metric is the best performing prior SISO self-talk cancellation design that leaves 1dB of residue over the noise floor. In other words, the receiver noise floor is increased by 1dB and therefore the SNR of the received signal is also decreased by 1dB. To put this number in context, this is extremely accurate since at most normal receive link SNRs, a 1dB decrease will have negligible impact. The reason for this residue is estimation and quantization error in the algorithms that calculate the weights for the filter taps used in analog

and digital cancellation. Estimation error is inevitable and cannot be avoided, but its important to keep it as small as possible.

How well would the SISO replication based design for MIMO perform on these two metrics? The optimal scenario is that the complexity of a M antenna full duplex MIMO radio would be $M\times$ the complexity of the SISO design, and it would have the same estimation error as the SISO design. We cannot do better than a linear increase in complexity and no increase in estimation error. However, the SISO replication based design has a complexity of $M^2\times$ the complexity of the SISO design. This is because it requires us to replicate the SISO design for each cross-talk factor, and therefore we need a total of M^2 versions of the SISO design. In terms of taps this implies $12\times M^2$ taps in analog circuits alone, along with the corresponding increase in digital cancellation FIR taps.

Second, this design's estimation error turns out to be worse compared to SISO design. At each receiver chain, we show in Sec. 3.2 that the residual interference scales linearly with the number of MIMO chains M. Intuitively the reason is that each replica of the SISO design is running an independent estimation algorithm for determining the values of the filter taps to use for cancellation. Since at each receiver chain we have M versions of the SISO design running, we will have a $M\times$ increase in estimation error and consequently the interference residue.

3 Design

We present a new cross talk cancellation technique for full duplex MIMO which is scalable and efficient. The key technique behind our MIMO cancellation design is a cascaded filter structure. Specifically, we exploit the fact that in MIMO, cross-talk and self-talk share a similar environment (or similar set of multi-path reflection and attenuation profiles in the channel). Further, cross-talk across chains is naturally reduced compared to the chain's own self-talk because of physical antenna separation. We exploit these insights to design a low complexity and highly accurate cross-talk cancellation system. For canceling the chain's own self-talk we use the design from prior work [14].

3.1 Reducing Complexity: The Cascade

Our design builds on a key insight: co-located MIMO antennas share a similar environment. In other words, the transfer function (i.e., the channel response across the frequency) that transforms the cross-talk signal from a neighboring transmit chain at the receive chain has a close relationship with the transfer function that the chain's own self-talk undergoes. Intuitively, this is because the environment around a radio looks essentially the same to neighboring antennas since they share the same reflectors in the environment, and the distances to these reflectors are almost the same from the closely-spaced antennas. The difference however is that any cross-talk signal experiences an additional delay before it arrives at a receive chain as compared to the chain's own self-talk signal [21, Sec. 2]. Technically this means that the phases of self-talk and cross- talks at a given receive chain might become different due to the delay, but can still be determined by a fixed relationship depending on antenna location and the environment. [2] What's important for MIMO full duplex design however is that cross talk and self talk transfer functions can be expressed as a function of each other, with a modifying factor to account for the antenna separation.

The above insight can be mathematically modeled as a cascade of transfer functions. Let $H_i(f)$ and $H_{ct}(f)$ be the transfer functions of the chain's own self-talk and cross-talk respectively, which are due to environment only, these cannot be directly measured. The overall relationship between these functions can be modeled as follows:

$$H_{ct}(f) = H_c(f).H_i(f) \quad (1)$$

where $H_c(f)$ is the cascade transfer function. The key observation is that $H_c(f)$ which cascaded with $H_i(f)$ results in the cross-talk transfer function, is an extremely simple transfer function. Typically $H_c(f)$ is a simple delay that corresponds to the fact that the two antennas are separated and the cross-talk signal experiences slightly higher delay compared to the self-talk.

How might we exploit this insight? The idea is to mimic the cancellation design in a cascade similar to the equation above as seen in Fig.5. Specifically, we could design simple low-complexity analog cancellation circuits and digital cancellation filters that model the cascade function $H_c(f)$. These circuits and filters would then feed into the cancellation circuits and digital cancellation filters for the chain's own self-talk cancellation and thus reuse all that circuitry to model the cross-talk channel. Remember that the circuits and digital filters for the chain's own self-talk are modeling $H_i(f)$, hence the cascaded structure is essentially recreating the above Eqn. 1. So the only additional complexity compared to the optimal MIMO design would be from the circuits and filters that model the cascade transfer function $H_c(f)$.

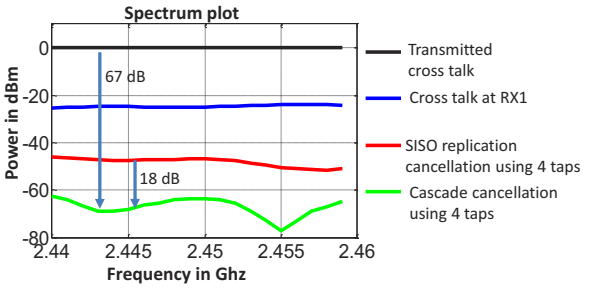
[2] Note that having a deterministic relationship between the self-talk and cross-talk channel responses does not contradict the assumption in MIMO channels that they form spatially independent streams as long as the antennas are separately by half a wavelength. The phase difference typically results in spatially independent streams [25]. Second, note that what we are exploiting is the fact that both the self-talk and cross-talk channels are correlated in their changes across frequency, i.e. the way the self-talk channel and cross-talk channels change across frequency are related and is a function of the environment. This fact has been studied in prior work, for example, a typical point to point LOS indoor MIMO channel can have a specific relationship across frequency across the different MIMO paths and still form spatially independent streams [21, 31, 25].

The natural question is how to design the cascade circuits itself? The intuition behind the design is to consider what the cascade circuits are exactly canceling compared to the self-talk cancellation circuits. The interference in the self-talk comes from two major factors. The first are the reflections from the antenna (impedance mismatch) and other components such as circulators. The second are the reflections from the environment. The reflections from the antenna are only part of the self-talk and are not part of the cross-talk, whereas the reflections from the environment are part of both self and cross-talk. Hence, the cascade cancellation circuit's job is to only cancel the environmental reflections.

The second insight is that the environmental reflections of the cross-talk are related to the environmental reflections the chain's own self-talk cancellation circuit is trying to cancel. To discover this relationship, we conduct the following experiment in a wide variety of locations in indoor scenarios. We first transmit a signal from a single antenna and measure the environmental channel response of the reflections at the same antenna [1]. We then measure the environmental reflection response at the neighboring MIMO antennas. Measuring the responses is possible because we know what we are transmitting, and we can use classic channel estimation techniques to measure the channel impulse response[3]. We then calculate the cascade transfer functions as described in Eqn. 1. We collect these calculated transfer functions and then check what is the complexity of the cascade cancellation circuit that can approximate these responses. This is an optimization problem, where the parameter is the number of taps that we are allowed to use in the cascade circuit, and the calculated responses are what we are trying to fit for. The goal is to minimize the number of taps in the cascade circuit, while fitting the cascade responses to a level of 40dB of cancellation (assuming we get 30dB of interference reduction from antenna separation in the cross-talk). The details of the technique are described in [1].

The number of analog taps required to realize the required performance for MIMO using the cascaded design calculated via the optimization above is tabulated in Fig. 6b. For a typical 3 antenna MIMO WiFi radio with 12cm separation between antennas (typical of APs), the antenna separation itself provides about 24dB of isolation, so we need another 41dB of cross-talk cancellation in analog (see Table. 2b for requirements). As we can see we need only four analog taps with the cascaded structure compared to the 12 taps required by the naive design for canceling cross-talk at an adjacent antenna and only two taps, when canceling to the farther out antenna as shown in Fig. 5. The cascaded design therefore requires

[3]This experiment is done via WARP software radios as discussed in evaluation.



(a) Cancellation performance in the frequency domain for the cascaded design and the replication based design with the same complexity for a 3 antenna MIMO full duplex radio operating a WiFi PHY in a 20MHz band at 0dBm TX power(WARP radios [6]).

Resource Comparison between SISO replication and Our design

	SISO replication design	Our design
Analog Cancellation taps (3X3)	108 (12*9)	56 (reduced by 1.92x)
Digital Cancellation taps (3X3)	1188 (132*9)	485 (reduced by 2.45x)
Tuning time (3X3)	9 ms (1ms*9)	.024 ms (reduced by 375x)
Analog Cancellation taps (mXm)	$O(M^2N)$	O(MN)
Digital Cancellation taps (mXm)	$O(M^2R)$	O(MR)
Tuning time (mXm)	$O(M^2)$	O(M)

(b) Table showing the reduction in complexity and tuning time with the cascaded design compared to the replication based design for both a 3 antenna full duplex MIMO radio as well as the general case of a M antenna full duplex MIMO radio.

Figure 6: **Cascade Design Evaluation.**

1.92× lower number of taps compared to the SISO replication design for a 3 antenna full duplex MIMO radio as seen in Fig. 6b. The reduction factor approaches the optimal 3× number as the number of antennas increases.

To verify the improvement for digital cascading (seen in Fig. 7), we conduct a similar experiment with the same setup (but with 20 dBm of total TX power). However, we provide the SISO replication design the required number of taps to meet the requirement on analog cancellation so we can specifically evaluate the benefits for digital cancellation with cascading. As seen in Fig. 6b, we need a total of 485 taps to cancel self-talk and cross-talk to the noise floor for a 3 antenna MIMO radio. Further, for the SISO replication based design using the same number of taps (485), the residual interference is still an additional 7dB. To achieve the same performance as our cascaded design with the SISO replication based design, we would need 1188 or 2.45× more taps as tabulated in Fig. 6b. Once again the reduction factor approaches the optimal number M and the number of antennas (M) grows. Finally in terms of cancellation performance, a 7dB increase in noise floor or reduction in desired signal's SNR is quite high by itself, and when we take into account the reduction in cancellation for analog of 18dB, we are looking at a 25dB reduction in overall cancellation for the SISO replication based design with the same complexity as our cascaded structure.

There are two main benefits to reducing complexity:

Reduction in size, cost and tuning time: Each addi-

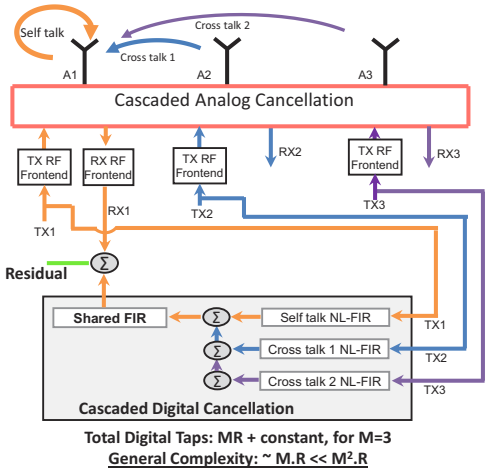


Figure 7: Shows the **cascaded digital cancellation** architecture for receiver chain RX1. Similar cascaded digital cancellation is applied to every receiver i.e., RX2 and RX3, not shown in this figure. The cascaded analog cancellation is implemented as shown in Fig. 5. The shared FIR brings significant saving of taps for overall MIMO cancellation. The NL-FIR's are the non-linear finite impulse response filter, recreating the digital copy of the unique component for the self-talk and cross-talks to be canceled at a receive chain.

tional filter tap increases the size of cancellation boards in analog and FPGA resource consumption in digital cancellation. For analog cancellation, our circuits consumed 110sq.cm of board area compared to nearly 216sq.cm for the SISO replication based design for a 3-antenna MIMO full duplex system. For example, we found experimentally that reducing the number of digital filter taps from 1185 to 485 for a 3 antenna MIMO radio means that a lower class Xilinx Kintex series FPGA has sufficient DSP resources to implement the cancellation, whereas the SISO replication based design would require the higher end Virtex FPGA [8]. This translates to enormous power savings, a Virtex FPGA consumes nearly 80W of power whereas a Kintex consumes only 40W on twice as less [9]. Power reduction translates to less heat and consequently simpler AP designs. Also to ultimately realize the design in compact boards, reducing the number of taps as much as possible is a must. A final consequence is the tuning time to compute the weights for each of these taps also reduces linearly with lesser number of taps (tuning time is pure overhead since during tuning the radio cannot be used for communication).

Reduction in Tx power waste: The amount of power that needs to be coupled off from the transmit paths to powering cancellation circuits depends linearly on the number of taps in the cancellation circuits. This is because each tap is of course only useful if some copy of the transmitted signal is passed through it, and in addition each tap has loss associated with it that adds up. Thus reducing number of taps helps reduce TX power waste.

3.2 Reducing Residue: Joint Training

The goal of digital cancellation is to clean out any remaining residual self-interference. Once again, a natural question is why not reuse the digital cancellation algorithms designed for SISO? In other words, for each receive chain in a M antenna full duplex MIMO radio, run M separate digital cancellation algorithms that estimate the chain's own self-talk and the other $M-1$ cross-talk interference components. These algorithms work by estimating the distortion experienced by each of the interference (both for linear and non-linear components). They then apply the estimated distortion functions to the known baseband copy of the transmitted signal and subtract it from the received signal.

The above approach doesn't work because every additional and independent digital cancellation algorithm we use in the receive chain linearly increases the residual interference after cancellation. In other words, performance worsens linearly with the number of MIMO chains. To see why, we start with describing why even a simplified SISO digital self-interference cancellation algorithm will have some residual interference that cannot be canceled.

Digital cancellation works in two stages, first there is a training phase and then cancellation phase. The training phase uses training symbols (e.g. the WiFi preamble), and the assumption is that there is no desired received signal from the other full duplex node. The training symbols are used to estimate the self-interference. Let's say the training self-interference symbol is s as seen in Fig. 8.a. The self- interference symbol is being received after transmission from the same radio (for simplicity assume there is no distortion from the channel), and the receiver adds its own noise n_1 (variance σ^2) to the received signal (this noise comes from effects such as quantization in the ADC). Hence the received signal y_1 can be written as,

$$y_1 = s + n_1$$

The best estimate of the self-interference s in this case is simply y_1 However this estimate $\hat{s}$ has some estimation error, which in this case is simply the power of the receiver noise as show below:

$$\hat{s} = y_1, \;\; E((s-\hat{s})^2) = E(n_1^2) = \sigma^2$$

How can we use this estimate to cancel subsequent self-interference? For simplifying the description, let's assume the packet that is being transmitted and is acting as self-interference is simply the same training symbol repeated throughout the packet (real world packets are of course not trivial like this, but this assumption does not change the basic insight below). To cancel this self-interference throughout the packet, the algorithm will

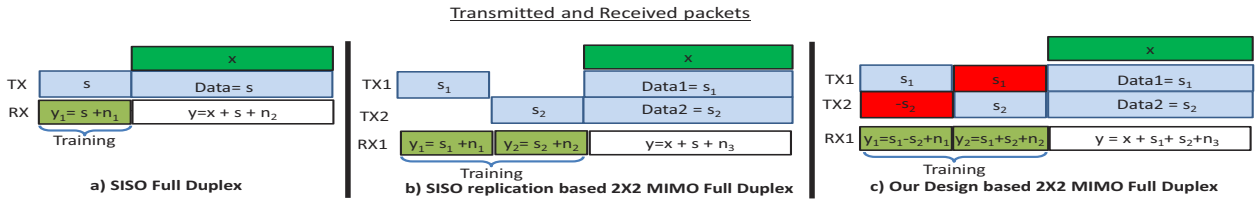


Figure 8: This figure shows the transmitted and received packets for a SISO full duplex, 2 antenna MIMO full duplex with the traditional training technique, and our design with the novel training technique. Notice the training symbol structure in the last figure, this allows us to reduce the estimation error by half for the self-talk and cross-talk components for a 2 antenna MIMO radio.

simply subtract the above estimate from the overall received signal. Lets say x is the actual desired received signal, the overall signal received is y, and the signal after cancellation, are given by:

$$\begin{aligned} y &= x+s+n_3 \\ \underbrace{y-\hat{s}}_{\text{cancellation}} &= x+\underbrace{s-\hat{s}}_{\text{estimation error } =\sigma^2}+\underbrace{n_3}_{\text{RX noise}} \end{aligned}$$

As we can see, the estimation error shows up as residual interference with variance of σ^2. As the best known prior design has shown this is on the order of 1dB over the half-duplex noise floor.

SISO Replication based MIMO design: It's now easy to see why a design for MIMO that simply uses M replicas of the digital cancellation algorithm at each receive chain for the self-talk and the $M-1$ cross-talk interference signals increases the estimation error roughly by a factor of M. The training symbol structure for a 2×2 MIMO transmission is shown in the Fig. 8.b. above, essentially there are two training symbols s_1 and s_2 sent over two slots from the two different transmit chains. The algorithms at a particular receive chain use these symbols like in the SISO case to estimate the self-talk and the cross-talk, and each of them will have their own estimation error. When these estimates are used for cancellation, the estimation errors add up, and the overall estimation error (or residual self-interference) at each receive chain is theoretically two times the SISO case. The math below shows the above intuition formally. First, the estimates for the self-talk and cross-talk symbols are given by:

$$\hat{s_1} = y_1,\ \ E((s_1-\hat{s_1})^2) = \sigma^2$$
$$\hat{s_2} = y_2,\ \ E((s_2-\hat{s_2})^2) = \sigma^2$$

When canceling to attempt to recover the desired received signal x, we can calculate the estimation error as follows:

$$\begin{aligned} y &= x+s_1+s_2+n_3 \\ \underbrace{y-\hat{s_1}-\hat{s_2}}_{\text{cancellation}} &= x+\underbrace{s_1-\hat{s_1}}_{\sigma^2}+\underbrace{s_2-\hat{s_2}}_{\sigma^2}+\underbrace{n_3}_{\text{RX noise}} \end{aligned}$$

As we can see, the estimation error shows up as residual interference with variance of $2\sigma^2$, both self-talk and cross-talk estimation introduce σ^2 error. We can recursively show that for a general M antenna full duplex MIMO radio, the estimation error and consequently residual interference on each receive chain goes to $M\sigma^2$.

Our Design: Our key contribution is a novel training symbol structure and estimation algorithm that reduces the estimation error for each interference component at each receiver chain (self-talk or cross-talk) to σ^2/M for a full duplex $M\times M$ MIMO radio. The key insight is to re-design the training symbols to reduce the estimation error. Specifically instead of sending training symbols from each of the transmit chains separately in consecutive time slots, we send a combination of all of them from each transmitter in parallel. The idea is to actually leverage the fact that there are two transmitters that could be leveraged to transmit training information jointly and thereby improve accuracy, there is no need to treat each of them separately. Doing so requires an intelligent joint training symbol design so that each symbol can be estimated as a linear combination of the received transmissions. Fig. 8.c. shows the main idea. We use a similar set of equations as before to show formally why this works. As seen in Fig. 8.c., the training symbols are transmitted by chain 1 and chain 2 simultaneously. In time slot 1, transmitter 1 and 2 transmit s_1 and $-s_2$, respectively. And in time slot 2, transmitter 1 and 2 transmit s_1 and s_2 respectively. Receiver 1, receives the combined symbols in time-slot 1 and time-slot 2, y_1 and y_2. Thus:

$$y_1 = s_1+s_2+n_1,\ \ y_2 = s_1-s_2+n_2$$

Lets assume the rest of the transmissions from the two chains are just repetitions of the same symbols s_1 and s_2 respectively (again this is for description simplicity and suffices to explain the insight). We need to get estimates for the data symbols s_1 and s_2 using the two received training symbols y_1 and y_2. The best estimates are given by:

$$\hat{s_1} = \frac{y_1+y_2}{2},\ \ E(s_1-\hat{s_1})^2 = E((\frac{n_1+n_2}{2})^2) = \frac{\sigma^2}{2}$$
$$\hat{s_2} = \frac{y_1-y_2}{2},\ \ E(s_2-\hat{s_2})^2 = E((\frac{n_1-n_2}{2})^2) = \frac{\sigma^2}{2}$$

As we can see, the error in each of these estimates (self-talk and cross-talk) is $\sigma^2/2$. Now when these estimates are used for cancellation, the following equation results:

$$\begin{aligned} y &= x + s_1 + s_2 + n_3 \\ \underbrace{y - \hat{s_1} - \hat{s_2}}_{\text{cancellation}} &= x + \underbrace{s_1 - \hat{s_1}}_{\frac{\sigma^2}{2}} + \underbrace{s_2 - \hat{s_2}}_{\frac{\sigma^2}{2}} + \underbrace{n_3}_{\text{RX noise}} \end{aligned}$$

As we can see the residual interference is only σ^2, rather than the $2\sigma^2$ that would have resulted from the SISO replication based design. Further, we can show by recursion that this residual is the same as the SISO design, i.e. there is no linear increase with the number of MIMO chains as the number of antennas increases. Implementation of this technique for wide-band OFDM systems is detailed in [1] based on [27].

Training in presence of another signal: While describing our algorithm above, we implicitly assumed that there is no other signal during the training phase, although in practice that might not be the case. This assumption however is not necessary. That is, even if there is a signal x as in the case of data, the algorithm would still work; the only change would be that the effective noise would now be $x + n_j$ instead of n_j at a given RX chain j and we use regularized least-squares estimation [14]. The downside is that the additional signal increases the interference during the training, thereby also increasing the number of samples or time required for convergence. Specifically, if interference to noise ratio after projecting the received signal on to the Tx signal space in least-squares is z, then it would take z times more samples to converge to the optimal point.

4 Robust MIMO Interference Cancellation

Interference cancellation needs to be robust to enable consistent full duplex operation in the face of frequent channel changes. To accomplish this, both analog and digital cancellation need to continuously tune their filter taps to maintain cancellation. The main bottleneck is tuning analog cancellation, since digital cancellation can be tuned on a per-packet basis in software as prior work has shown [30, 23, 14]. Tuning analog circuits requires measuring the residue in digital and then sending control signals to analog components, which is relatively slow. Minimizing the amount of time required to tune here is therefore critical, since during the time spent tuning packets likely cannot be received. We focus on this problem in this paper and re-use the algorithms from prior work for tuning digital cancellation.

The prior SISO full duplex design demonstrated a technique to tune a single analog cancellation in around a millisecond. However, as before if we were to naively replicate the same algorithm for all the self-interference components, we would need M^2ms for a M antenna full duplex MIMO radio (e.g. 9ms for a 3 antenna full duplex). Such a high overhead is untenable for moderately mobile environments where the channel changes on average every 60ms (e.g. WiFi hotspots).

In this paper we propose a novel technique that reduces tuning time by three orders of magnitude, i.e. an algorithm that tunes the circuit in 8μs. Note that this algorithm also applies to the SISO case, and therefore improves on the best known prior SISO design too. Our insight is to model the cancellation circuit as a filter whose response we are tuning to match as closely as possible the frequency response of the self-interference channel. Like prior work, we estimate the frequency response of the cancellation circuit for different combinations of filter tap values. The pre-calculated response is represented in a matrix A, whose each column is the frequency response of the analog cancellation circuit for a particular value of the filter tap at K different frequencies in the band of interest (e.g. K=128 for a 20MHz bandwidth in our current prototype for WiFi). Now assuming $H(f)$ is the frequency response of the self-talk channel in the frequency domain (i.e. the channel introduced by the antenna, circulator and any strong environmental reflections), the analog cancellation tuning problem reduces to:

$$\min_x ||H - Ax||^2$$

Where, H is the column consisting of $H(f)$ at different frequencies, and x, represents a binary indicator vector for selecting the corresponding filter tap values as in [14].

The efficacy of the tuning that results from the above problem depends on the accuracy in the measurement of $H(f)$. We can measure $H(f)$ using the preamble of the received interference signal $y(t)$ (e.g. the first two OFDM symbols of a transmitted WiFi packet which are known preamble symbols). The challenge is measuring the frequency response of the interference channel accurately. The accuracy is limited by the linearity of the transmit-receive chain, which is 30dB, By this we mean that any initial measurement can only have an accuracy of 30dB. The main reason is that the transceiver produces non-linearities which act as noise to the channel estimation algorithm. In other words the received interference signal $y(t)$ has non-linearities that are only 30dB below the main linear signal component. Our key contribution in this paper is a technique to accurately measure this channel quickly in the presence of non-linearities and tune analog cancellation.

Source of error and its magnitude: The transmitter produces non-linearities 30 dB lower than the transmitted signal. To show mathematically, say $x(t)$ is the baseband signal that is being transmitted after up-conversion and amplification, we can write

$$x_{tx}(t) = x(t) + a_3 x(t)^3 + a_5 x(t)^5 + a_7 x(t)^7 + \ldots + w(t)$$

This transmitted signal $x_{tx}(t)$ is somewhat known to us because we know $x(t)$, however its non-linear components and the transmit noise $w(t)$ are unknown . This signal further undergoes the circulator and antenna channel $H(f)$ (which we wish to estimate), so when its received at the receiver the frequency domain representation of the received signal is given by:

$$Y(f) = H(f) * \mathscr{F}(x(t) + a_3 x(t)^3 + \ldots) + \textit{ transmit noise}$$

Here, a_3 is around $10^{(-30/20)}$, i.e., its 30 dB lower. Further transmit noise distortion is 40 dB lower than the signal level of $x(t)$. The challenge is that our channel estimation algorithm is only going to use its knowledge of $x(t)$ to estimate the channel $H(f)$, and the other terms in the received interference signal limit the accuracy of the estimation to 30dB (the estimation noise is 30dB lower).

Accurate, Iterative method: The key idea is to run the estimation algorithm in an iterative fashion. Remember that the WiFi preamble has two OFDM symbols, each of length $4\mu s$. After the first OFDM symbol, we solve the above equation to produce an inaccurate estimate of the interference channel H_a and tune the cancellation circuit to achieve (at best) 30dB of cancellation (we cannot cancel more than our estimation accuracy). Now when we obtain the second preamble symbol, we know that the non-linearities and the transmit noise components that were producing the error are reduced by 30dB. We can exploit this fact by the following trick:

We transmit one OFDM symbol to estimate the inaccurate H_a, which can be written as a function of accurate H as, $H_a = H + e_1$. Note e_1 is 30 dB lower than H. We use the same algorithm as [14] to optimize the following,

$$\min_x ||H_a - Ax||^2$$

which produces the solution as $\hat{x}$, which gives us the values to use in the filter taps. We program the cancellation circuit using these values and achieve a 30 dB cancellation. Next, when we transmit second OFDM symbol and measure the channel response we get:

$$H_b = (H - A\hat{x}) + e_2$$

Notice that e_2 is 30 dB lower then $H - A\hat{x}$ and $H - A\hat{x}$ is 30 dB lower than H. So in essence e_2 is 60 dB lower than H. Define,

$$\tilde{H} = H_b + A\hat{x}$$
$$\tilde{H} = H + e_2$$

Thus, we can this new estimate $\tilde{H}$ with an error that is 60 dB lower. We use this estimate to re-tune the optimization algorithm and find a solution $\tilde{x}$ that tells us what values to use for the analog filter taps. This new solution provides 60 dB cancellation. Further, we only needed two OFDM symbols of $4\mu s$ each to get to this cancellation.

Extension to Cascaded Filter Structure: The above description is for a single cancellation circuit, but our MIMO design has a cascaded structure of multiple circuits. This leads to a combinatorial explosion in the parameter space that makes the problem NP hard to solve if we use the above approach. In this subsection we present a trick to approximate the overall combinatorial problem via two reduced complexity problems which can be solved using the same technique as the SISO one presented above.

We describe the algorithm in the context of tuning the cancellation circuits at receiver 1 for self and cross-talk in a 2 antenna MIMO radio. Lets say H_{11} is the self-talk channel response and H_{12} is the cross-talk channel response. The general tuning problem can be stated as:

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & t && (2)\\ \text{subject to} \quad & norm(H_{11} - A_1 x_1) \leq t && (3)\\ & norm(H_{12} - (A_1 x_1) \odot (A_2 x_2)) \leq t && (4) \end{aligned}$$

Where, $\odot$ represents the element wise multiplication of the column, and t represents the analog cancellation achieved, and A_1 is the response of the self-talk cancellation board with N taps in Fig.5 and A_2 is the response of the cascade cancellation board with C taps. The second constraint Eq. 4 renders the problem irreducible to a convex solvable form, and in fact the columnwise multiplication of the indicator variable vectors explodes the problem space and makes it a NP hard combinatorial problem.

We use a novel trick to approximate and help solve this problem practically. Since the first constraint in Eq. 3 is trying to find $A_1 x_1 = H_{11}$, we can approximate $A_1 x_1$ in the next constraint, Eq.4 with H_{11} which is known (since we measured H_{11}). This is of course an approximation, but it suffices to solve for x_2 using this substitution since we are after all trying to emulate the same cascaded channel response structure using our circuits as described in Sec. 3. Thus instead of a cascade of unknown variables, the new problem to solve is

$$\begin{aligned} \underset{x_1,x_2}{\text{minimize}} \quad & t && (5)\\ \text{subject to} \quad & norm(H_{11} - A_1 x_1) \leq t && (6)\\ & norm(H_{12} - H_{11} \odot (A_2 x_2)) \leq t && (7) \end{aligned}$$

This new problem is no longer a combinatorial problem. This can be reduced to an integer program, which can be solved using randomized rounding in fraction of micro seconds practically [14]. Thus in effect the substitution trick reduces the non-tractable combinatorial problem into a tractable problem, whose solution can be found using the techniques described above. The tuning time for each MIMO chain is still two OFDM symbols, and the overall tuning time for the MIMO radio therefore scales linearly with M, the number of chains.

5 Evaluation

In this section, we experimentally demonstrate that our MIMO full duplex design almost completely cancels all self-talk and cross-talk interference to the noise floor with a low-complexity design. We also show that this translates to a doubling of throughput for the link performance.

We implement our design using four WARP v2 boards for building a 3×3 MIMO full duplex link. We design our own boards for analog cancellation and integrate them with the WARP boards. At each receive chain, we have analog circuits with 12 taps for the self-talk cancellation, 4 taps for the first cross talk and 2 taps for the farthest transceiver. In total we have 56 taps in the analog cancellation circuits for a 3 antenna full duplex MIMO radio, and total of 485 filter taps in digital cancellation. Since the WARP cannot generate 20dBm transmit power, we use an external off-the-shelf power amplifier [5].

We compare against the SISO replication based design primarily. This is the straightforward replication of the recently published SISO full duplex design as discussed at the start of Sec. 3. We compare against two variants of this design. One is a design that fully replicates the analog and digital cancellation implementations for all self-talk and cross-talk cancellations. As discussed before the complexity of this design is a factor of two higher for analog and 2.5× higher for digital compared to our design. We call this design **SISO Replication**. However to make an apples to apples comparison with our design we also implement a SISO replication design with the same complexity as our design. The difference compared to our design is that, it neither use the cascaded structure nor the novel estimation algorithm, but simply replicates the SISO design with lower number of taps. We experiment with the tap distribution between self-talk and cross-talk to obtain the best overall cancellation. We call this compared approach **SISO Low Complexity Replication**.

The best recent work that we could compare for MIMO full duplex is **MIDUs [11]**. However this design only works for small bandwidths (i.e. 500KHz). Further, it relies on obtaining 50dB of cancellation using antenna cancellation (which itself requires more antennas per MIMO chain and is problematic), and then complements it with another 30dB of digital cancellation. However when we go to normal bandwidths of 20MHz found in WiFi signals, then the antenna cancellation reduces to 40dB at best, and hence we are limited to a total of 70dB of cancellation. This is significantly worse than SISO replication, and hence we omit comparisons against MIDU. SISO replication is in fact the best comparable technique that we can compare our design to.

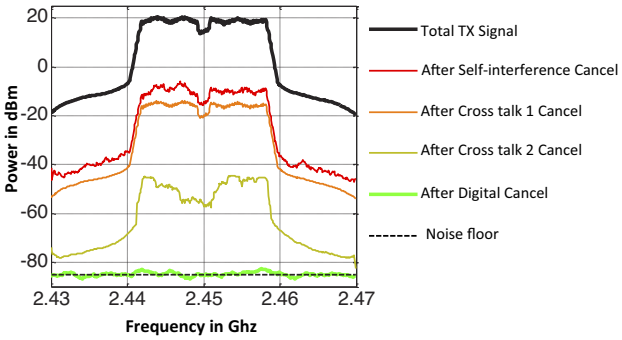


Figure 9: Spectrum plot after cancellation of various self-talk and cross-talk components for RX1 of a 3×3 full duplex system using our design.

Unless stated otherwise, all experiments are conducted by placing the two full duplex nodes at various locations in our department building. At each location, we repeat the experiment ten times and calculate the average performance.

5.1 Can we cancel all the interference for 3 antenna full duplex MIMO ?

The first claim made in this paper is capability of canceling all of the interference for the 3×3 MIMO. To prove this, we experimentally test if we can fully cancel a WiFi 802.11n 20MHz signal upto a max transmit power of 20dBm for a 3×3 MIMO. To demonstrate we first pick one instance of this experiment, and show the spectrum plot of the received self-interference after various stages of cancellation in Fig. 9. Remember, that in analog we first cancel the chain's own self-talk leaking through the circulator, and then the cross-talk from the other two antennas. Finally, we apply our digital cancellation step to clean up the residual. We see that overall in analog we achieve 68-70dB of self-interference cancellation after all three stages. This satisfies the requirements outlined in Sec. 2.

We now place the node at several different locations in the testbed. At each location we vary the overall TX power from 16dBm to 20dBm and plot the average cancellation for each power across all locations. At each location and for each power, we conduct 40 runs. The goal is to show that we can consistently cancel to the noise floor for a variety of transmit powers up to and including the max average TX power of 20dBm. In each instance of the above experiment, we also measure the increase in noise floor due to any residual self-interference that is not canceled. Note that the increase in noise floor represents the SNR loss the received signal will experience when the node is used in full duplex mode. Fig. 10 plots the average cancellation and the increase in noise floor as a function of TX power.

Fig. 10 shows that our 3-antenna MIMO full duplex design cancels the entire self interference almost to the

Figure 10: Increase in noise floor vs TX power on the left side and Cancellation vs TX power on the right side. For different MIMO cancellation designs, we present the performance of a full duplex 3 antenna full duplex MIMO system.

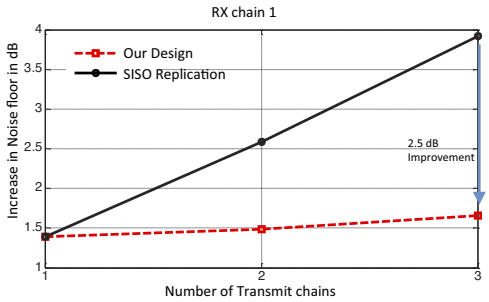


Figure 11: Increase in noise floor at a RX chain as the number of MIMO chains and consequently the number of cross-talk components increase from 1 to 3. With our design we observe a 2.5 dB improvement for 3×3 MIMO per RX chain compared to the SISO replication design.

noise floor. In case of max average transmit power of 20dBm [14], the noise floor is increased by 1.6dB over each receive chain's noise floor. The SISO replication design increases the noise floor by 4dB per receive chain, while the SISO low complexity replication approach increases the noise floor by 25dB. Finally, the performance of our design and the SISO replication design scales with increasing TX power, while the other replication based design is limited due to its inability to cancel the increasing transmit noise and non-linearities due to the reduced number of taps available to it.

5.2 Scaling with the number of MIMO antennas

A question with MIMO is how does full duplex performance scale with increasing number of transmit chains. The ideal case would be to maintain the same level of cancellation at each RX chain as the number of transmit antennas increase, starting from one antenna. In other words, even with increasing number of transmit antennas and cross-talk components that need to be canceled, we retain the same performance as if there was a single transmit antenna and a single self-interference signal to deal with. Fig. 11 plots the increase in the noise floor at one receive chain as we go from one transmit chain to three transmit chains for a MIMO radio for both our design as well as the SISO replication technique. The overall TX power is fixed to be 20dBm (additional 10 dB of PAPR for WiFi [14], i.e., total 30 dBm) to adhere to ISM band EIRP requirements. Hence if we use a single transmit chain, then all the 20dBm is used for a single antenna. If we use two chains, then each antenna produces a 17dBm signal and so on.

As we can see from the figure, our design maintains a near-constant performance even as we go from one to three transmit chains. In other words, the performance is roughly the same regardless of the number of cross-talk components (We do wish to note that we could not go beyond three transmit chains due to hardware limitations, verifying the above claim for higher number of transmit chains is future work). On the other hand, the SISO replication design shows the noise floor increasing linearly with increasing number of transmit chains, a fact we provided theoretical intuition for in Sec. 3.2. Thus this design will look worse as we scale to higher MIMO configurations. We omit the SISO low complexity replication approach because its results are significantly worse.

5.3 Dynamic Adaptation

An important metric for analog cancellation is how quickly can it be tuned, and how often do we need to tune? The best know prior technique [14] required around 1 millisecond to tune a single SISO analog cancellation circuit. So for a 3×3 MIMO, applying the same algorithm will take at least 9ms for the SISO replication based design. In this section we show the efficacy of our new tuning algorithm which cuts the tuning time to $8\mu s$ per receive chain. Fig. 12 shows the tuning time as a function of the amount of analog cancellation. To achieve the 70dB analog cancellation, our algorithm takes $8\mu s$ per chain, for a total of $24\mu s$ for the full radio. The prior work as we can see take a millisecond per chain. The interesting takeaway is that both schemes achieve 40dB of analog cancellation fairly quickly (with one preamble symbol, i.e. $4\mu s$), but our scheme covers the final 30dB in one more step of $4\mu s$, while the prior scheme takes an exponential number of symbols to achieve that. The reason for this improvement is precisely our ability to get a precise measurement of the self-interference channel using the trick described in Sec. 4.

A second question is how often one needs to tune? This depends on the environment and the amount of analog cancellation that needs to be maintained. In this paper, we tune for challenging indoor environments which have strong multi-path (this is the main source of analog cancellation degradation). We define a near-field coherence time which depends on the amount of analog can-

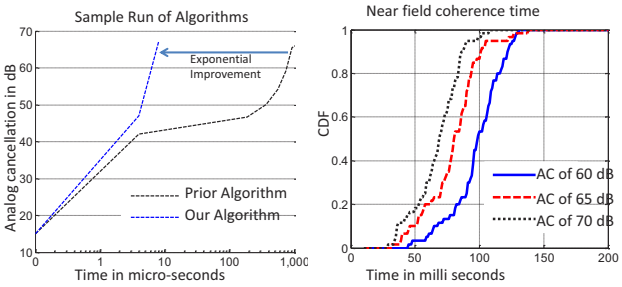


Figure 12: Tuning time for analog cancellation. The first figure shows the three orders of magnitude improvement in tuning time with our algorithm compared to the best known prior approach. The second figure shows how often this tuning algorithm needs to be run for an indoor environment.

cellation and is essentially the time for which that analog cancellation can be maintained on average before the circuits need to be retuned. Fig. 12 plots the near-field coherence time for three different analog cancellation targets. As we can see, to maintain an analog cancellation of 70dB, we need to retune roughly every 60ms. Our tuning overhead is $24\mu s$, which is negligible.

5.4 Does Full Duplex Double Throughput?

A final question is whether all this cancellation performance translates to a the desired doubling of overall throughput. We show experimentally the throughput gains of our 3×3 MIMO full duplex design compared to the SISO replication based design. Two full duplex 3-antenna MIMO nodes are placed at different locations and we send 1000 packets in full duplex mode between them, and then send 1000 packets for each direction of the half duplex mode. We repeat this experiment for each bitrate that is available in WiFi. We pick the bitrate which maximizes the overall throughput for all of the compared full duplex designs and half duplex respectively. We repeat this experiment for 50 different locations. We found the received power of the links varied uniformly between $-45\ to\ -80$dBm, across locations as found in typical indoor deployments. To put these numbers in perspective, this implies that the SNR of the links in half duplex mode ranges from 5dB to 40dB. We plot the throughput for half duplex and full duplex designs in Fig. 13. Note that all of these throughput numbers account for the overhead introduced by the periodic analog cancellation tuning. As we can see, our full duplex system achieves a median throughput gain of $1.95\times$ over the half duplex mode, but the SISO replication based design with full complexity only achieves a $1.36\times$ gain. The reason is the higher increase in noise floor from the SISO replication based design. For example, if the link SNR in half duplex mode is 10dB, a 4dB increase in noise floor will result in worse overall throughput for full duplex compared to running

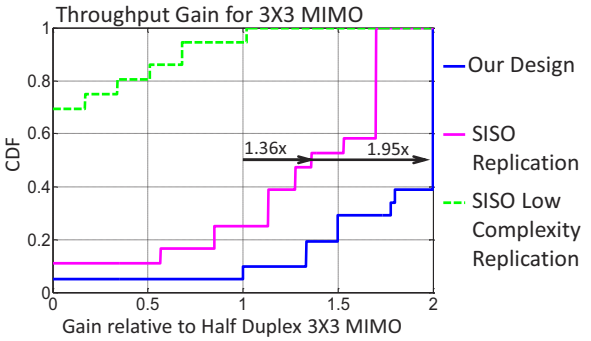


Figure 13: CDF of throughput gain relative to half duplex 3×3 WiFi MIMO. Our 3×3 MIMO system provides a median gain of 95% relative to half duplex, whereas the SISO replication design only provides a $1.36\times$ relative gain.

the link in half duplex mode. Our ability to keep the noise floor constant results in a performance close to the theoretical optimum.

The SISO replication based design with lower complexity is quite poor, in fact in 70% of the scenarios, the throughput was zero. This is because it increases the noise floor by at least 25dB which acts as noise and if the SNR is below 30dB no signal is decoded (WiFi requires a minimum of $4-5$dB SNR to decode the lowest rate packet). As the half-duplex link SNR increases, the performance improves but is still not sufficient to beat the system throughput achieved by half duplex. The reason is that even if the link half-duplex SNR is 35dB, it implies that we only have two 10dB links for full duplex. The throughput achieved with a single 35dB half duplex link is still higher than two 10dB links. Consequently the only region where we could find improvements for full duplex over half duplex with this design was when the link SNR was greater than 38dB.

6 Conclusion

This paper brings towards completion a line of work on PHY layer of full duplex radios, and shows that practical full duplex is achievable for the most common wireless protocols and for MIMO while using commodity radios. The cancellation techniques developed in this paper are fundamental and apply to a wide variety of problems [20, 13, 18] where self-interference cancellation is needed. While this work wraps up work on board level realizations of full duplex, much work remains in realizing these designs in a chip. Tackling these problems is future work.

Acknowledgments: We would like to thank Aruna Balasubramanian, Rakesh Misra, Kiran Joshi, the anonymous reviewer's and the Stanford Networked Systems Group members for their insightful comments.

References

[1] *Decoupling of Static Channel*. http://snsg.stanford.edu/fullduplexmimo.pdf.

[2] *FCC Part 15.247 Rules Systems Using Digital Modulation*. http://www.semtech.com/images/datasheet/fcc_digital_modulation_systems_semtech.pdf.

[3] *Huawei Sets Out Its 5G Stall*. http://www.lightreading.com/document.asp?doc_id=703466&init_gateway=true.

[4] *Power Amplifier Data-sheet*. http://datasheets.maximintegrated.com/en/ds/MAX2828-MAX2829.pdf.

[5] *Power Amplifier Data-sheet*. http://www.minicircuits.com/pages/npa/PGA-105+_NPA.pdf.

[6] *WARP Project*. http://warpproject.org.

[7] *White paper by NI on Understanding Dynamic Hardware Specifications*. http://www.ni.com/white-paper/5529/en.

[8] *Xilinx 7 Series FPGA Overview*. http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[9] *Xilinx Power Estimator Tool*. http://www.xilinx.com/products/design_tools/logic_design/xpe.htm.

[10] E. B. And. An analog cmos high-speed continuous-time fir filter, 2000.

[11] E. Aryafar, M. A. Khojastepour, K. Sundaresan, S. Rangarajan, and M. Chiang. Midu: enabling mimo full duplex. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, Mobicom '12, pages 257–268, New York, NY, USA, 2012. ACM.

[12] J. Bardwell. *Tech Report*. http://www.connect802.com/download/techpubs/2005/commercial_radios_E0523-15.pdf.

[13] D. Bharadia, K. R. Joshi, and S. Katti. Full duplex backscatter. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 4. ACM, 2013.

[14] D. Bharadia, E. McMilin, and S. Katti. Full duplex radios. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, SIGCOMM '13, pages 375–386, New York, NY, USA, 2013. ACM.

[15] J. I. Choi, M. Jain, K. Srinivasan, P. Levis, and S. Katti. Achieving single channel, full duplex wireless communication. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, MobiCom '10, pages 1–12, New York, NY, USA, 2010. ACM.

[16] M. Duarte, C. Dick, and A. Sabharwal. Experiment-driven characterization of full-duplex wireless systems. *CoRR*, abs/1107.1276, 2011.

[17] E. Everett, A. Sahai, and A. Sabharwal. Passive self-interference suppression for full-duplex infrastructure nodes. *CoRR*, abs/1302.2185, 2013.

[18] E. Fear, S. Hagness, P. Meaney, M. Okoniewski, and M. Stuchly. Enhancing breast tumor detection with near-field imaging. *Microwave Magazine, IEEE*, 3(1):48–56, 2002.

[19] S. Gollakota, H. Hassanieh, B. Ransford, D. Katabi, and K. Fu. They can hear your heartbeats: non-invasive security for implantable medical devices. *SIGCOMM Comput. Commun. Rev.*, 41(4), Aug. 2011.

[20] S. S. Hong, J. Mehlman, and S. Katti. Picasso: flexible rf and spectrum slicing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 37–48, New York, NY, USA, 2012. ACM.

[21] X. Hong, C.-X. Wang, J. Thompson, B. Allen, W. Malik, and X. Ge. On space-frequency correlation of uwb mimo channels. *Vehicular Technology, IEEE Transactions on*, 59(9):4201–4213, Nov 2010.

[22] Y. Hua, P. Liang, Y. Ma, A. Cirik, and Q. Gao. A method for broadband full-duplex mimo radio. *Signal Processing Letters, IEEE*, 19(12):793 –796, dec. 2012.

[23] M. Jain, J. I. Choi, T. Kim, D. Bharadia, S. Seth, K. Srinivasan, P. Levis, S. Katti, and P. Sinha. Practical, real-time, full duplex wireless. MobiCom '11, pages 301–312, New York, NY, USA, 2011. ACM.

[24] S.-C. Jung, M.-S. Kim, and Y. Yang. A reconfigurable carrier leakage canceler for uhf rfid reader front-ends. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 58(1):70 –76, jan. 2011.

[25] J. Kermoal, L. Schumacher, K. Pedersen, P. Mogensen, and F. Frederiksen. A stochastic mimo radio channel model with experimental validation. *Selected Areas in Communications, IEEE Journal on*, 20(6):1211–1226, Aug 2002.

[26] M. Knox. Single antenna full duplex communications using a common carrier. In *Wireless and Microwave Technology Conference (WAMICON), 2012 IEEE 13th Annual*, pages 1–6, 2012.

[27] X. D. Lin and K. H. Chang. Optimal pn sequence design for quasisynchronous cdma communication systems. *Communications, IEEE Transactions on*, 45(2):221–226, 1997.

[28] D. Morgan, Z. Ma, J. Kim, M. Zierdt, and J. Pastalan. A generalized memory polynomial model for digital predistortion of rf power amplifiers. *Signal Processing, IEEE Transactions on*, 54(10):3852–3860, Oct 2006.

[29] B. Radunovic, D. Gunawardena, P. Key, A. Proutiere, N. Singh, V. Balan, and G. Dejean. Rethinking indoor wireless mesh design: Low power, low frequency, full-duplex. In *Wireless Mesh Networks (WIMESH 2010), 2010 Fifth IEEE Workshop on*, pages 1 –6, 2010.

[30] B. Radunovic, D. Gunawardena, A. Proutiere, N. Singh, V. Balan, and P. Key. Efficiency and fairness in distributed wireless networks through self-interference cancellation and scheduling. Technical Report MSR-TR-2009-27, Microsoft Research, 2009.

[31] K. Yu, M. Bengtsson, B. Ottersten, D. McNamara, P. Karlsson, and M. Beach. Modeling of wide-band mimo radio channels based on nlos indoor measurements. *Vehicular Technology, IEEE Transactions on*, 53(3):655–665, May 2004.

Recursively Cautious Congestion Control

Radhika Mittal[*] *Justine Sherry*[*] *Sylvia Ratnasamy*[*] *Scott Shenker*[*†]

[*]*UC Berkeley* [†]*ICSI*

Abstract

TCP's congestion control is deliberately cautious, avoiding network overloads by starting with a small initial window and then iteratively ramping up. As a result, it often takes flows several round-trip times to fully utilize the available bandwidth. In this paper we propose RC3, a technique to quickly take advantage of available capacity from the very first RTT. RC3 uses several levels of lower priority service and a modified TCP behavior to achieve near-optimal throughputs while preserving TCP-friendliness and fairness. We implement RC3 in the Linux kernel and in NS-3. In common wide-area scenarios, RC3 results in over 40% reduction in average flow completion times, with strongest improvements – more than 70% reduction in flow completion time – seen in medium to large sized (100KB - 3MB) flows.

1 Introduction

We begin this paper by noting two facts about networks. First, modern ISPs run their networks at a relatively low utilization [18, 21, 26]. This is not because ISPs are incapable of achieving higher utilization, but because their networks must be prepared for link failures which could, at any time, reduce their available capacity by a significant fraction. Thus, most ISP networks are engineered with substantial headroom, so that ISPs can continue to deliver high-quality service even after failures.

Second, TCP congestion control is designed to be *cautious*, starting from a small window size and then increasing every round-trip time until the flow starts experiencing packet drops. The need for fairness requires that all flows follow the same congestion-control behavior, rather than letting some be cautious and others aggressive. Caution, rather than aggression, is the better choice for a uniform behavior because it can more easily cope with a heavily overloaded network; if every flow started out aggressively, the network could easily reach a congestion-collapsed state with a persistently high packet-drop rate.

These decisions – underloaded networks and cautious congestion control – were arrived at independently, but interact counter-productively. When the network is underloaded, flows will rarely hit congestion at lower speeds. However, the caution of today's congestion control algorithms requires that flows spend significant time ramping up rather than aggressively assuming that more bandwidth is available. In recent years there have been calls to increase TCP's initial window size to alleviate this problem but, as we shall see later in the paper, this approach brings only limited benefits.

In this paper we propose a new approach called *recursively cautious congestion control* (RC3) that retains the advantages of caution while enabling it to efficiently utilize the available bandwidth. The idea builds on a perverse notion of quality-of-service, called WQoS, in which we assume ISPs are willing to offer *worse* service if certain ToS bits are set in the packet header (and the mechanisms for doing so – priority queues, are present in almost all currently deployed routers). While traditional calls for QoS – in which better service is available at a higher price – have foundered on worries about equity (should good Internet service only be available to those who can pay the price?), pricing mechanisms (how do you extract payments for the better service?), and peering (how do peering arrangements cope with these higher-priced classes of service?), in our proposal we are only asking ISPs to make several worse classes of service available that would be treated as regular traffic for the purposes of charging and peering. Thus, we see fewer institutional barriers to deploying WQoS. Upgrading an operational network is a significant undertaking, and we do not make this proposal lightly, but our point is that many of the fundamental sources of resistance to traditional QoS do not apply to WQoS.

The RC3 approach is quite simple. RC3 runs, at the highest priority, the same basic congestion control algorithm as normal TCP. However, it also runs congestion control algorithms at each of the k worse levels of service; each of these levels sends only a fixed number of packets, with exponentially larger numbers at lower priority levels. As a result, all RC3 flows compete fairly at every priority level, and the fact that the highest priority level uses the traditional TCP algorithms ensures that RC3 does not increase the chances of congestion collapse. Moreover, RC3 can immediately "fill the pipe" with packets (assuming there are enough priority levels), so it can leverage the bandwidth available in underutilized networks.

We implemented RC3 in the Linux kernel and in the NS-3 network simulator. We find through experiments on both real and simulated networks that RC3 provides strong gains over traditional TCP, averaging 40% reduction in flow completion times over all flows, with strongest gains – of over 70% – seen in medium to large sized flows.

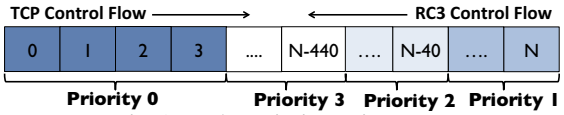


Fig. 1: Packet priority assignments.

2 Design

2.1 RC3 Overview

RC3 runs two parallel control loops: one transmitting at normal priority and obeying the cautious transmission rate of traditional TCP, and a second "recursive low priority" (RLP) control loop keeping the link saturated with low priority packets.

In the primary control loop, TCP proceeds as normal, sending packets in order from index 0 in the byte stream, starting with slow-start and then progressing to normal congestion-avoidance behavior after the first packet loss. The packets sent by this default TCP are transmitted at 'normal' priority – priority 0 (with lower priorities denoted by higher numbers).

In the RLP control loop, the sender transmits additional traffic from the same buffer as TCP to the NIC.[1] To minimize the overlap between the data sent by the two control loops, the RLP sender starts from the very *last* byte in the buffer rather than the first, and works its way towards the beginning of the buffer, as illustrated in Figure 1. RLP packets are sent at low priorities (priority 1 or greater): the first 40 packets (from right) are sent at priority 1; the next 400 are sent at priority 2; the next 4000 at priority 3, and so on.[2] The RLP traffic can only be transmitted when the TCP loop is not transmitting, so its transmission rate is the NIC capacity minus the normal TCP transmission rate.

RC3 enables TCP selective ACK (SACK) to keep track of which of low priority (and normal priority) packets have been accepted at the receiver. When ACKs are received for low priority packets, no new traffic is sent and no windows are adjusted. The RLP control loop transmits each low priority packet once and once only; there are no retransmissions. The RLP loop starts sending packets to the NIC as soon as the TCP send buffer is populated with new packets, terminating when its 'last byte sent' crosses with the TCP loop's 'last byte sent'. Performance gains from RC3 are seen only during the slow-start phase; for long flows where TCP enters congestion avoidance, TCP will keep the network maximally utilized with priority 0 traffic, assuming appropriately sized buffers [8]. If the bottleneck link is the edge link,

[1] As end-hosts support priority queueing discipline, this traffic will never pre-empt the primary TCP traffic.

[2] RC3 requires the packets to be exponentially divided across the priority levels to accommodate large flows within feasible number of priority bits. The exact number of packets in each priority level has little significance, as we shall see in § 5.1.2.

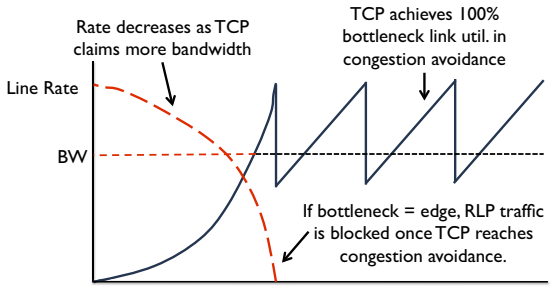


Fig. 2: Congestion window and throughput with RC3.

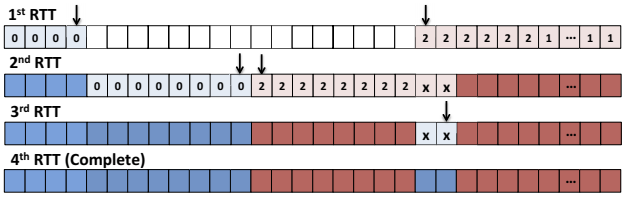


Fig. 3: Example RC3 transmission from §2.2.

high priority packets will pre-empt any packets sourced by the RLP directly at the end host NIC; otherwise the low priority packets will be dropped elsewhere in the network.

Figure 2 illustrates how the two loops interact: as the TCP sender ramps up, the RLP traffic has less and less 'free' bandwidth to take advantage of, until it eventually is fully blocked by the TCP traffic. Since the RLP loop does not perform retransmissions, it can leave behind 'holes' of packets which have been transmitted (at low priority) but never ACKed. Because RC3 enables SACK, the sender knows exactly which segments are missing and the primary control loop retransmits only those segments.[3] Once the TCP 'last byte sent' crosses into traffic that has already been transmitted by the RLP loop, it uses this information to retransmit the missing segments and ensure that all packets have been received. We walk through transmission of a flow with such a 'hole' in the following subsection.

2.2 Example

We now walk through a toy example of a flow with 66 packets transmitted over a link with an edge-limited delay-bandwidth product of 50 packets. Figure 3 illustrates our example.

In the first RTT, TCP sends the first 4 packets at priority 0 (from left); after these high priority packets are transmitted, the RLP loop sends the remaining 62 packets to the NIC – 40 packets at priority 1 and 22 packets at priority 2 (from right), of which 46 packets are transmitted by the NIC (filling up the entire delay-bandwidth product of 50 packets per RTT).

The 21st and 22nd packets from the left (marked as

[3] Enabling SACK allows selective retransmission for dropped low priority packets. However, RC3 still provides significant performance gains when SACK is disabled, despite some redundant retransmissions.

Xs), sent out at priority 2, are dropped. Thus, in the second RTT, ACKs are received for all packets transmitted at priority 0 and for all but packets 21 and 22 sent at lower priorities. The TCP control loop doubles its window and transmits an additional 8 packets; the RLP sender ignores the lost packets and the remaining packets are transmitted by the NIC at priority 2.

In the third RTT, the sender receives ACKs for all packets transmitted in the second RTT and TCP continues to expand its window to 16 under slow start. At this point, the TCP loop sees that all packets except 21st and 22nd have been ACKed. It, therefore, transmits only these two packets.

Finally, in the fourth RTT the sender receives ACKs for the 21st and 22nd packets as well. As all data acknowledgements have now been received by the sender, the connection completes.

3 Performance Model

Having described RC3 in §2, we now model our expected reduction in Flow Completion Time (FCT) for a TCP flow using RC3 as compared to a basic TCP implementation. We quantify gains as ((FCT with TCP) - (FCT with RC3)) / (FCT with TCP) – *i.e.* the percentage reduction in FCT [11]. Our model is very loose and ignores issues of queuing, packet drops, or the interaction between flows. Nonetheless, this model helps us understand some of the basic trends in performance gains. We extensively validate these expected gains in §5 and see the effects of interaction with other flows.

Basic Model: Let BW be the capacity of the bottleneck link a flow traverses, and u be the utilization level of that link. We define A, the available capacity remaining in the bottleneck link as $A = (1-u) \times BW$. Since RC3 utilizes *all* of the available capacity, a simplified expectation for FCTs under RC3 is $RTT + \frac{N}{A}$, where RTT is the round trip time and N is the flow size.

TCP does not utilize all available capacity during its slow start phase; it is only once the congestion window grows to $A \times RTT$, that the link is fully utilized. The slow start phase, during which TCP leaves the link partially idle, lasts $log(min(N, A \times RTT)/i)$ RTTs, with i being the initial congestion window of TCP. This is the interval during which RC3 can benefit TCP.

In Figure 4, the solid line shows our expected gains according to our model. Recall that i denotes the initial congestion window under TCP. For flow sizes $N < i$, RC3 provides no gains over a baseline TCP implementation, as in both scenarios the flow would complete in $RTT + \frac{N}{A}$. For flow sizes $i < N < A \times RTT$, the flow completes in 1 RTT with RC3, and $log(N/i)$ RTTs with basic TCP in slow start. Consequently, the reduction in FCT increases with N over this interval.

Once flow sizes reach $N > A \times RTT$, basic TCP reaches a state where it can ensure 100% link utilization after $log(A \times RTT/i)$ RTTs. Therefore, the improvements from RC3 become a smaller fraction of overall FCT with increasingly large flows; this reduction roughly follows $\frac{log(A \times RTT/i) \times RTT \times A}{N}$ (ignoring a few constants in the denominator).

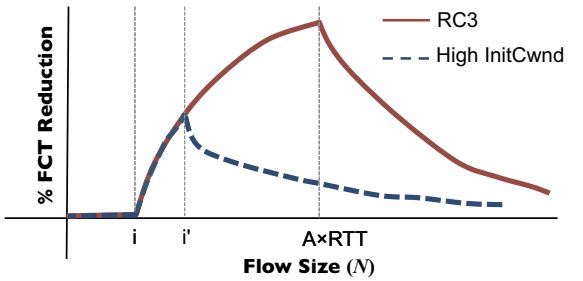


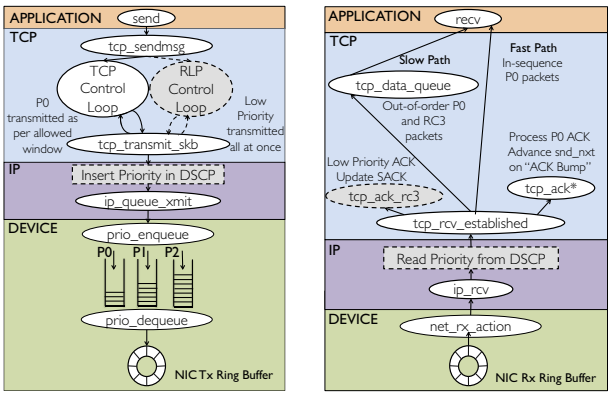
Fig. 4: Performance gains as predicted by a simple model for RC3 and an increased initial congestion window.

Parameter Sensitivity: The above model illustrates that improvements in FCTs due to RC3 are dependent primarily on three parameters: the flow size (N), the effective bandwidth-delay product ($A \times RTT$), and the choice of the initial congestion window (i). Peak improvements are observed when N is close to $A \times RTT$, because under these conditions the flow completes in 1 RTT with RC3 and spends its entire life time in slow start without RC3. When the delay-bandwidth product increases, both the optimal flow size (for performance improvement) increases, and the maximum improvement increases.

Adjusting *i*: There are several proposals [7, 12] to adjust the default initial congestion window in TCP to 10 or even more packets. Assume we adjusted a basic TCP implementation to use a new value, some i' as its initial congestion window. The dotted line in Figure 4 illustrates the gains from such an i'. When i' increases, the amount of time spent in slow start decreases to $log(min(N, A \times RTT)/i') \times RTT$. Flows of up to i' packets complete in a single RTT, but unless $i' = A \times RTT$ (hundreds of packets for today's WAN connections), adjusting the initial congestion window will always underperform when compared to RC3. However, there is good reason not to adjust i' to $A \times RTT$: without the use of low priorities, as in RC3, sending a large amount of traffic without cautious probing can lead to an increase in congestion and overall *worse* performance. Our model does not capture the impact of queueing and drops, however, in §5.1.4 we show via simulation how increasing the initial congestion window to 10 and 50 packets penalizes small flows in the network.

4 Linux Implementation

We implemented RC3 as an extension to the Linux 3.2 kernel on a server with Intel 82599EB 10Gbps NICs.



(a) Sending Data (b) Receiving Data and Acks

Fig. 5: Modifications to Linux kernel TCP stack.

Our implementation cleanly sits within the TCP and IP layers and requires minimal modifications to the kernel source code. Because our implementation does not touch the core TCP congestion control code, different congestion control implementations can easily be 'swapped out' while still retaining compatibility with RC3. After describing our implementation in §4.1, we discuss how RC3 interacts with other components of the networking stack in §4.2, including application buffering, QoS support, hardware performance optimizations, and SACK extensions.

4.1 TCP/IP in the Linux Kernel

We briefly provide high-level context for our implementation, describing the TCP/IP stack under the Linux 3.2 kernel. We expect that RC3 can be easily ported to other implementations and operating systems as well, but leave this task to future work.

Figure 5 illustrates the kernel TCP/IP architecture at a very high level, along with our RC3 extensions shaded in gray. The Linux kernel is as follows. When an application calls *send()*, the *tcp_sendmsg* function is invoked; this function segments the send buffer in to 'packets' represented by the socket buffer (*skb*) datastructure. By default, each *skb* represents one packet to be sent on the wire. After the data to be transmitted has been segmented in to *skb*s, it is passed to the core TCP logic, and then forwarded for transmission through the network device queue to the NIC.

On the receiver side, packets arrive at the NIC and are forwarded up to a receive buffer at the application layer. As packets are read in, they are represented once again as *skb* datatypes. Once packets are copied to *skb*s, they are passed up the stack through the TCP layer. Data arriving in-order is sent along the 'fast-path', directly to the application layer receive buffer. Data arriving out of order is sent along a 'slow path' to an out of order receive queue, where it waits for the missing packets to arrive, before being forwarded up in-order to the application layer.

We now describe how we extend these functions to support RC3.

4.1.1 Sending Data Packets

RC3 extends the send-side code in the networking stack with two simple changes, inserting only 72LOC in the TCP stack and 2LOC in the IP stack. The first change, in the TCP stack, is to invoke the RLP control loop once the data has been segmented in the *tcp_sendmsg* function. We leave all of the segmentation and core TCP logic untouched – we merely add a function call in *tcp_sendmsg* to invoke the RLP loop, as shown in Fig. 5.

The RLP loop then reads the TCP write queue iteratively from the tail end, reading in the packets one by one, marking the appropriate priority in the packet buffer, and then sending out the packet. The field *skb→priority* is assigned according to the sequence number of the packet: the RLP loop subtracts the packet's sequence number from the tail sequence number and then divides this value by the MSS. If this value is ≤ 40, the packet is assigned priority 1, if the value is ≤ 400 it is assigned priority 2, and so on. After the priority assignment, the *skb* packets are forwarded out via the *tcp_transmit_skb* function.

Our second change comes in the IP layer as packets are attached to an IP header; where we ensure that *skb→priority* is not overwritten by the fixed priority assigned to the socket, as in the default case. Instead, the value of *skb→priority* is copied to the DSCP priority bits in the IP header.

Overall, our changes are lightweight and do not interfere with core congestion control logic. Indeed, because the TCP logic is isolated from the RC3 code, we can easily enable TCP CUBIC, TCP New Reno, or any other TCP congestion control algorithms to run alongside RC3.

4.1.2 Receiving Data Packets and ACKs

Extending the receive-side code with RC3 is similarly lightweight and avoids modifications to the core TCP control flow. Our changes here comprise only of 46 LOC in the TCP stack and 1 LOC in the IP stack.

Starting from bottom to top in Figure 5, our first change comes as packets are read in off the wire and converted to *skb*s – here we ensure that the DSCP priority field in the IP header is copied to the *skb* priority field; this change is a simple 1 LOC edit.

Our second set of changes which lie up the stack within TCP. These changes separate out low priority packets from high priority in order to ensure that the high priority ACKing mechanism (and therefore the sender's congestion window) and other TCP variables remain unaffected by the low priority packets. We identify the low priority packets and pass them to the

out of order 'slow path' queue, using the unmodified function *tcp_data_queue*. We then call a new function, *tcp_send_ack_rc3*, which sends an ACK packet for the new data at the same priority the data arrived on, with the cumulative ACK as per the high priority traffic, but SACK tags indicating the additional low priority packets received. The priority is assigned in the field *skb→priority*, and the packets are sent out by calling *tcp_transmit_skb*, as explained in § 4.1.1.

The other modifications within the TCP receive code interpose on the handling of ACKs. We invoke *tcp_ack_rc3* on receiving a low priority ACK packet, which simply calls the function to update the SACK scoreboard (which records the packets that have been SACKed), as per the SACK tags carried by the ACK packet. We also relax the SACK validation criteria to update the SACK "scoreboard" to accept SACKed packets beyond *snd_nxt*, the sequence number up to which data has been sent out by the TCP control loop.

Typically when a new ACK is received, the stack double-checks that the received ACK is at a value less than *snd_nxt*, discarding the ACKs that do not satisfy this constraint. We instead tweak the ACK processing to update the *snd_nxt* value when a high-priority ACK is received for a sequence number that is greater than *snd_nxt*: such an ACK signals that the TCP sender has "crossed paths" with traffic transmitted by the RLP and is entering the cleanup phase. We advance the send queue's head and update *snd_nxt* to the new ACKed value and then allow TCP to continue as usual; we call this jump an "ACK bump."

While these changes dig shallowly in to the core TCP code, they do not impact our compatibility with various congestion control schemes.

4.2 Specific Implementation Features

We now discuss how RC3 interacts with some key features at all layers of the networking stack, from software to NIC hardware to routers and switches.

4.2.1 Socket Buffer Sizes

The default send and receive buffer sizes in Linux are very small - 16KB and 85KB respectively. Performance gains from RC3 are maximized when the entire flow is sent out in the first RTT itself. This requires us to make the send and receive buffers as big as the maximum flow size (up to a few MBs in most cases). Window scaling is turned on by default in Linux, and hence we are not limited by the 64KB receive window carried by the TCP header.

RC3 is nonetheless compatible with smaller send buffer sizes: every call to *tcp_sendmsg* passes a chunk of data to the RLP control loop, which treats that chunk, logically, as a new flow as far as priority assignment is concerned. We include a check to break the RLP control loop to ensure that the same packet is not transmitted twice by subsequent calls to *tcp_sendmsg*. Indeed, this behavior can help flows which resume from an application-layer imposed idle period.

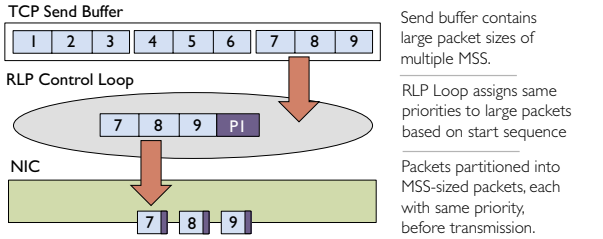


Fig. 6: RC3 combined with TSO.

4.2.2 Using QoS Support

RC3 is only effective if priority queueing is supported at both endhosts and the routers in the network.

Endhosts: We increase the Tx queue length at the software interface, to ensure that it can store all the packets forwarded by the TCP stack. The in-built traffic control functionality of Linux is used to set the queuing discipline (*qdisc*) as *prio* and map the packet priorities to queue 'bands'. The *prio* qdisc maintains priority queues in software, writing to a single NIC ring buffer as shown in Figure 5. Thus, when the NIC is free to transmit, a packet from band N is dequeued only if all bands from 0 to $N-1$ are empty. Up to 16 such bands are supported by the Linux kernel, which are more than enough for RC3.[4]

Routers: All modern routers today support QoS, where flow classes can be created and assigned to a particular priority level. Incoming packets can then be mapped to one of these classes based on the DSCP field. The exact mechanism of doing so may vary across different vendors. Although the ISPs may use the DSCP field for some internal prioritization, all we require them to do is to read the DSCP field of an incoming packet, assign a priority tag to the packet which can be recognized by their routers, and then rewrite the priority in the DSCP field when the packet leaves their network.

4.2.3 Compatibility with TSO/LRO

TCP Segmentation Offload (TSO) and *Large Receiver Offload* (LRO) are two performance extensions within the Linux kernel that improve throughput through batching. TSO allows the TCP/IP stack to send packets comprising of multiple MSSes to the NIC, which then divides them into MSS sized segments before transmitting them on the link. LRO is the receiver counterpart which amasses the incoming segments into larger packets at the driver, before sending them higher up in the stack. TSO and LRO both improve performance by amortizing

[4] 16 priority levels is sufficient to support RC3 flow sizes on the order of a petabyte!

the cost of packet processing across packets in batches. Batched packets reduce the average per-packet processing CPU overhead, consequently improving throughput.

Figure 6 illustrates how RC3 behaves when TSO is enabled at the sender and a larger packet, comprising of multiple MSSes is seen by the RLP control loop. At first glance, TSO stands in the way of RC3: RC3 requires fine-grained control over individual packets to assign priorities, and the over sized packets passing through under TSO hinder RC3's ability to assign priorities correctly when it includes data from packets that should be assigned different priority classes. Rather than partitioning data within these extra large packets, we simply allow RC3 to label them according to the lowest priority of any data in the segment. This means that we might not strictly follow the RC3 design in §2 while assigning priorities for some large packets. However, such a situation can arise only when the MSSes in a large packet overlap with the border at which priority levels switch. Since the traffic is partitioned across priority level exponentially, such cases are infrequent. Further, the largest TSO packet is comprised of at most 64KB. Therefore, no more than 43 packets would be improperly labeled at the border between priority levels.

TSO batching leads to a second deviation from the RC3 specification, in that segments within a large packet are sent in sequence, rather than in reverse order. For example, in Figure 6, the segments in the packet are sent in order (7,8,9) instead of (9,8,7). Hence, although RC3 still processes *skb* packets from tail to front, the physical packets sent on the wire will be sent in short in-order bursts, each burst with a decreasing starting sequence number. Allowing the forward sequencing of packets within a TSO batch turns out to be useful when LRO is enabled at the receiver, where batching at the driver happens *only* if the packets arrive in order. As we'll show in §5.2, combining RC3 with TSO/LRO reduces the OS overhead of processing RC3 packets by almost 50%, and consequently leads to net gains in FCTs.

4.2.4 SACK Enhancements

Although RC3 benefits when SACK is enabled, it is incompatible with some SACK enhancements. Forward Acknowledgment (FACK) [24], is turned on by default in Linux. It estimates the number of outstanding packets by looking at the SACKed bytes. RC3 SACKed packets may lead the FACK control loop to falsely believe that all packets between the highest cumulative ACK received and the lowest SACK received are in flight. We therefore disable FACK to avoid the RC3 SACKed bytes from affecting the default congestion control behavior. Doing so does not penalize the performance in most cases, as the Fast Recovery mechanism continues transmitting unsent data after a loss event has occurred and partial ACKs are received, allowing lost packets to be efficiently detected by duplicate ACKs. DSACK [17] is also disabled, to avoid the TCP control loop from inferring incorrect information about the ordering of packets arriving at the receiver based on RC3 SACKs.

5 Experimental Evaluation

We now evaluate RC3 across several dimensions. In §5.1, we evaluate RC3 extensively using NS-3 simulations - §5.1.1, compares RC3's FCT reductions with the model we described in §3; §5.1.2 and §5.1.3 evaluate RC3's *robustness* and *fairness*, and §5.1.4 compares RC3's FCT reductions relative to other designs. We evaluate our Linux RC3 implementation in §5.2.

5.1 Simulation Based Evaluation

We evaluate RC3 using a wide range of simulation settings. Our primary simulation topology models the Internet-2 network consisting of ten routers, each attached to ten end hosts, with 1Gbps bottleneck bandwidth and 40ms RTT. It runs at 30% average link utilization [18, 21, 26]. The queue buffer size is equal to the delay-bandwidth product (RTT$\times$BW) in all cases, which is 5MB for our baseline. The queues do priority dropping and priority scheduling. All senders transmit using RC3 unless otherwise noted. Flow sizes are drawn from an empirical traffic distribution [6]; with Poisson interarrivals.

For most experiments we present RC3's performance relative to a baseline TCP implementation. Our baseline TCP implementation is TCP New Reno [16] with SACK enabled [9, 25] and an initial congestion window of 4 [7]; maximum segment size is set to 1460 bytes while slow start threshold and advertised received window are set to infinity.

5.1.1 Baseline Simulation

We first investigate the baseline improvements using RC3 and compare them to our modeled results from §3.

Validating the Model: Figure 7 compares the gains predicted by our model (§3) with the gains observed in our simulation. The data displayed is for 1Gbps bottleneck capacity, 40ms average RTT, and 30% load. Error bars plotting the standard deviation across 10 runs are shown; they sit very close to the average. For large flows, the simulated gains are slightly lower than predicted; this is the result of queueing delay which is not included in our model. For small flows – four packets or fewer – we actually see *better* results than predicted by the model. This is due to large flows completing sooner than with regular TCP, leaving the network queues more frequently vacant and thus decreasing average queueing delay for short flows. Despite these variations, the simulated and modeled results track each other quite closely: for all but the smallest flows, we see gains of 40–75%.

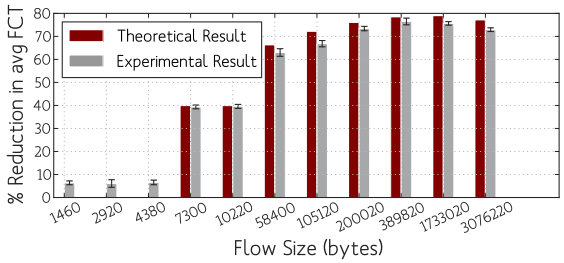
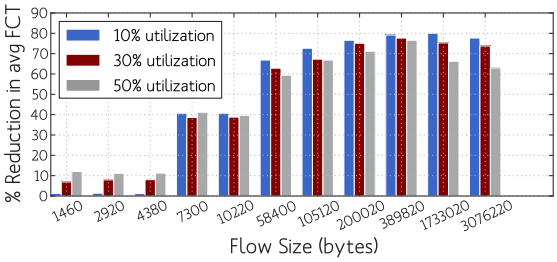
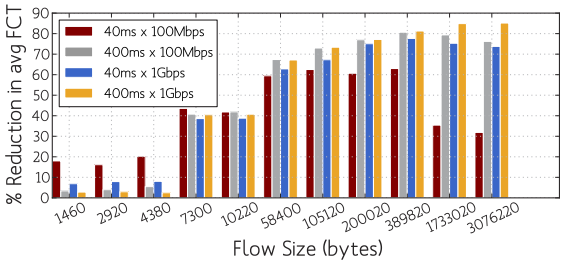


Fig. 7: Reduction in FCT as predicted by model vs simulations. (RTT×BW = 5MB, 30% average link utilization)



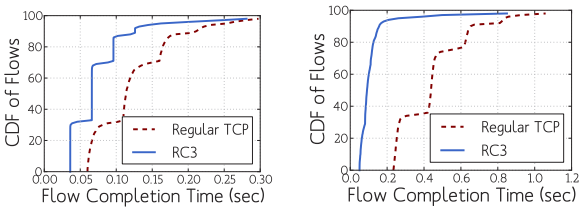
		Average Over Flows	Average Over Bytes
10% Load	Regular FCT (s)	0.125	0.423
	RC3 FCT (s)	0.068	0.091
	% Reduction	**45.56**	**78.36**
30% Load	Regular FCT (s)	0.135	0.443
	RC3 FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**
50% Load	Regular FCT (s)	0.15	0.498
	RC3 FCT (s)	0.088	0.176
	% Reduction	**41.44**	**64.88**

Fig. 8: Reduction in FCT with load variation, with RTT×BW fixed at 5MB



		Average Over Flows	Average Over Bytes
100Mbps Bottleneck 40ms avg. RTT	Regular FCT (s)	0.167	0.691
	RC3 FCT (s)	0.11	0.442
	% Reduction	**33.98**	**36.05**
100Mbps Bottleneck 400ms avg. RTT	Regular FCT (s)	0.948	3.501
	RC3 FCT (s)	0.567	0.783
	% Reduction	**40.29**	**77.62**
1 Gbps Bottleneck 40ms avg. RTT	Regular FCT (s)	0.135	0.443
	RC3 FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**
1 Gbps Bottleneck 400ms avg. RTT	Regular FCT (s)	0.971	3.59
	RC3 FCT (s)	0.558	0.569
	% Reduction	**42.45**	**84.17**

Fig. 9: Reduction in average FCT with variation in RTT×BW with 30% average link utilization



(a) Flow size 7.3KB (b) Flow size 1.7MB

Fig. 10: Cumulative Distribution of FCTs. (RTT×BW = 5MB, 30% average link utilization)

Link Load: Figure 8 shows FCT performance gains comparing RC3 to the base TCP under uniform link load of 10%, 30%, or 50%. RTT×BW is fixed at 5MB across all experiments. As expected, performance improvements decrease for higher average link utilization. For large flows, this follows from the fact that the available capacity ($A = (1-u) \times BW$) reduces with increase in utilization u. Thus, there is less spare capacity to be taken advantage of in scenarios with higher link load. However, for smaller flows, we actually see the opposite trend. This is once again due to reduced average queueing delays, as large flows complete sooner with most packets having lower priorities than the packets from the smaller flows.

RTT×BW: Figure 9 shows the FCT reduction due to RC3 at varying RTT×BW. In this experiment we adjusted RTTs and bandwidth capacities to achieve RTT×BW of 500KB (40ms×100Mbps), 5MB (40ms×1Gbps and 400ms×100Mbps) and 50MB(400ms×1Gbps). As discussed in §3, the performance improvement increases with increasing RTT×BW, as the peak of the curve in Figure 4 shifts towards the right. The opposite trend for very short flows is repeated here as well.

Summary: Overall, RC3 provides strong gains, reducing flow completion times by as much as 80% depending on simulation parameters. These results closely track the results predicted by the model presented in §3.

5.1.2 Robustness

In the prior section, we evaluated RC3 within a single context. We now demonstrate that these results are robust, inspecting RC3 in numerous contexts and under different metrics. Many of the results in this section are summarized in Table 1.

Performance at the Tails: Our previous figures plot the average and standard deviation of flow completion times; in Figures 10(a) and (b) we plot the full cumulative distribution of FCTs from our Internet-2 experiments for two representative flow sizes, 7.3KB and 1.7MB.[5] We

[5]The 'jumps' or 'banding' in the CDF are due to the uniform link latencies in the simulation topologies. Paths of two hops had an RTT of 40, paths of three hops had an RTT of 60, and so on. A flow which completes in some k RTTs while still under slow start thus completes

		Average Over Flows	Average Over Bytes
Default: Internet-2	Regular FCT (s)	0.135	0.443
	RC3 FCT (s)	0.076	0.114
	% Reduction	**43.54**	**74.35**
Telstra Topology	Regular FCT (s)	0.159	0.510
	RC3 FCT (s)	0.084	0.111
	% Reduction	**47.07**	**78.13**
RedClara Topology	Regular FCT (s)	0.17	0.429
	RC3 FCT (s)	0.097	0.098
	% Reduction	**42.78**	**77.16**
ESNet Topology	Regular FCT (s)	0.207	0.478
	RC3 FCT (s)	0.137	0.0976
	% Reduction	**33.91**	**79.58**
2000 Workload	Regular FCT (s)	0.0871	0.238
	RC3 FCT (s)	0.0704	0.079
	% Reduction	**13.83**	**66.73**
Link Heterogeneity	Regular FCT (s)	0.159	0.541
	RC3 FCT (s)	0.087	0.141
	% Reduction	**45.35**	**73.89**

Table 1: RC3 performance in robustness experiments.

see in these results that performance improvements are provided across the board at all percentiles; even the 1^{st} and 99^{th} percentiles improve by using RC3.

Topology: We performed most of our experiments on a simulation topology based off a simplified model of the Internet-2 network. To verify that our results were not somehow biased by this topology, we repeated the experiment using simulation topologies derived from the Telstra network, the Red Clara academic network, and the complete ESNet [1, 4], keeping the average delay as 40ms and the bottleneck bandwidth as 1Gbps. All three topologies provided results similar to our initial Internet-2 experiments: average FCTs for Telstra improved by 47.07%, for Red Clara, by 42.78%, and for ESNet by 33.91%.

Varying Workload Distribution: Our baseline experiments use an empirical flow size distribution [6]. A noticeable property of the flow size distribution in our experiments is the presence of of very large flows (up to a few MBs) in the tail of the distribution. We repeated the Internet-2 experiment with an empirical distribution from a 2000 [2] study, an era when average flow sizes were smaller than today. Here we saw that the average FCT improved by only 13.83% when averaged over all flows. When averaging FCT gains weighted by bytes, however, we still observe strong gains for large flows resulting in a reduction of 66.73%.

Link Heterogeneity: We now break the assumption of uniform link utilization and capacity: in this experiment we assigned core link bandwidths in the Internet-2 topology to a random value between 500Mbps and 2Gbps. We observed that FCTs in the heterogenous experiment were higher than in the uniform experiment, for both TCP and RC3. Nevertheless, the penalty to TCP was worse, resulting in a stronger reduction in flow completion times, when averaged across flows.

in approximately $k * RTT$ time. This created fixed steps in the CDF, as per the RTTs

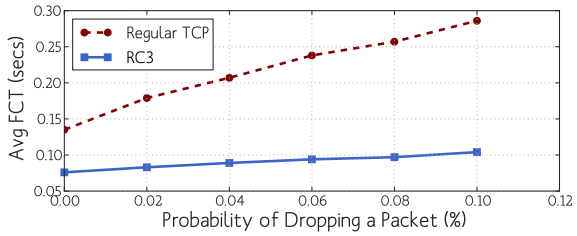
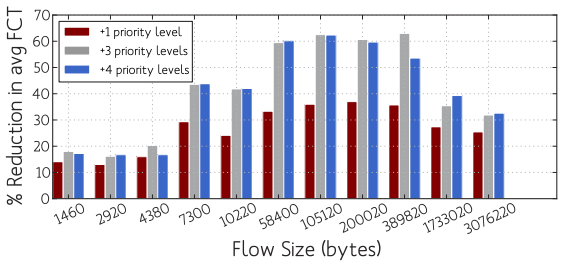


Fig. 11: Average FCTs with increasing arbitrary loss rate. (RTT×BW = 5MB, 30% average link utilization)



		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.167	0.691
1 RC3 Priority Level	FCT	0.126	0.496
	%	**24.55**	**28.22**
3 RC3 Priority Levels (40, 400, 4000)	FCT	0.11	0.442
	%	**33.98**	**36.05**
4 RC3 Priority Levels (10, 100, 1000, 10000)	FCT	0.112	0.434
	%	**32.94**	**37.19**

Fig. 12: Reduction in FCT with varying priority levels. (RTT×BW = 500KB, 30% average link utilization)

Loss Rate: Until now all loss has been the result of queue overflows; we now introduce random arbitrary loss and investigate the impact on RC3. Figure 11 shows flow completion times for TCP and RC3 when arbitrary loss is introduced for 0.02-0.1% of packets. We see that loss strongly penalizes TCP flows, but that RC3 flows do not suffer nearly so much as TCP. RC3 provides even stronger gains in such high loss scenarios because each packet essentially has two chances at transmission. Further, since the RLP loop ignores ACKs and losses, low priority losses do not slow the sending rate.

Priority Assignment: Our design assigns packets across multiple priorities, bucketed exponentially with 40 packets at priority 1, 400 at priority 2, and so on. We performed an experiment to investigate the impact of these design choices by experimenting with an RC3 deployment when 1, 3, or 4 additional priority levels were enabled; the results of these experiments are plotted in Fig. 12. We see that dividing traffic over multiple priority levels provides stronger gains than with only one level of low priority traffic. The flows which benefit the most from extra priorities are the medium-sized flows which, without RC3, require more than one RTT to complete during slow start. A very slight difference is seen in

performance gains when bucketing packets as (10, 100, 1000, 10000) instead of (40, 400, 4000).

Application Pacing: Until now, our model application has been a file transfer where the entire contents of the transfer are available for transmission from the beginning of the connection. However, many modern applications 'pace' or 'chunk' their transfers. For example, after an initial buffering phase YouTube paces video transfers at the application layer, transmitting only a few KB of data at a time proportional to the rate that the video data is consumed. To see the effect of RC3 on these type of flows, we emulated a YouTube transfer [30] with a 1.5MB buffer followed by 64KB chunks sent every 100ms. Ultimately, RC3 helped these video connections by decreasing the amount of time spent in buffering by slightly over 70% in our experimental topology. This means that the time between when a user loads the page and can begin video playback decreases while using RC3. However, in the long run, large videos did not complete transferring the entire file any faster with RC3 because their transfer rate is dominated by the 64KB pacing.

Summary: In this section, we examined RC3 in numerous contexts, changing our experimental setup, looking at alternative application models, and investigating the tail distribution of FCTs under RC3 and TCP. In all contexts, RC3 provides benefits over TCP, typically in the range of 30-75%. Even in the worst case context we evaluated, when downlink rather than uplink capacities bottlenecked transmission, *RC3 still outperformed baseline TCP by 10%*.

5.1.3 RC3 and Fairness

In this subsection we ask, *is RC3 fair?* We evaluate two forms of fairness: how RC3 flows of different sizes interact with each other, and how RC3 interacts with concurrent TCP flows.

RC3 with RC3: It is well-known that TCP in the long run is biased in that its bandwidth allocations benefit longer flows over short ones. We calculated the effective throughput for flows using TCP or RC3 in our baseline experiments (Figure 13). TCP achieves near-optimal throughput for flow sizes less than 4 packets, but throughput is very low for medium-sized flows and only slightly increases for the largest (multiple-MB) flows. RC3 maintains substantially high throughput for all flow sizes, having a slight relative bias towards medium sized flows.

RC3 with TCP: To evaluate how RC3 behaves with concurrent TCP flows, we performed an experiment with mixed RC3 and TCP flows running concurrently. We allowed 50% of end-hosts attached to each core router in our simulations (say in set *A*) to use RC3, while the remaining 50% (set *B*) used regular TCP. Overall, FCTs

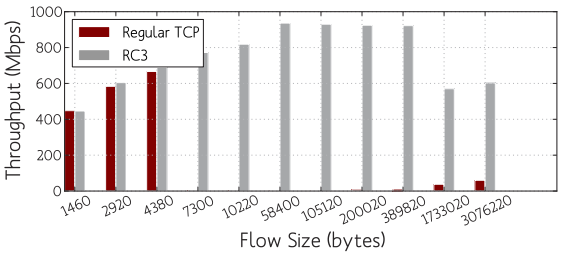


Fig. 13: Median Flow Throughput

for both RC3 and TCP were lower than in the same setup where all flows used regular TCP. Thus, RC3 is not only fair to TCP, but in fact improves TCP FCTs by allowing RC3 flows to complete quickly and 'get out of the way' of the TCP flows.

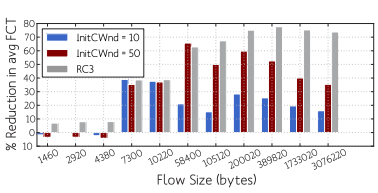
5.1.4 RC3 In Comparison

We now compare the performance gains of RC3 against some other proposals to reduce TCP flow completion times.

Increasing Initial Congestion Window: Figure 14(a) compares the performance gains obtained from RC3 with the performance gains from increasing the baseline TCP's initial congestion window (InitCwnd) to 10 and 50. For most flow sizes, especially larger flows, RC3 provides stronger improvements than simply increasing the initial congestion window. When averaging across all flows, RC3 provides a 44% reduction in FCT whereas increasing the InitCwnd to 10 reduces the FCT by only 13% and 50 reduces it by just 24%. Further, for small flow sizes ($\leq$ 4 packets), increasing the InitCwnd actually introduces a *penalty* due to increased queueing delays. RC3 never makes flows do worse than they would have under traditional TCP. These results confirm our expectations from §3.

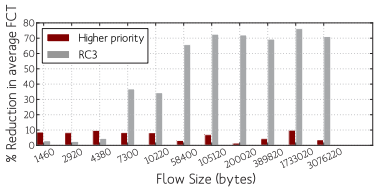
Traditional QoS: An alternate technique to improve FCTs is to designate certain flows as 'critical' and send those flows using unmodified TCP, but at higher priority. We annotated 10% of flows as 'critical'; performance results for the critical flows alone are showed in Fig. 14(b). When the 'critical' 10% of flows simply used higher priority, their average FCT reduces from 0.126 seconds to 0.119 seconds; while the non-critical flows suffered a very slight (<2%) penalty. When we repeated the experiment, but used RC3 for the critical flows (leaving the rest to use TCP), the average FCT reduced from 0.126 seconds to 0.078 seconds, as shown in Figure 14(b) . Furthermore, non-critical flows showed a slight (<1%) *improvement*. This suggests that it is better to be able to send an unrestricted amount of traffic, albeit at low priority, than to send at high priority at a rate limited by TCP.

RCP: Finally, we compare against RCP, an alternative transport protocol to TCP. With RCP, routers calculate average fair rate and signal this to flows; this allows flows



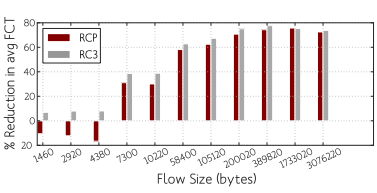
		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.135	0.443
i = 10	FCT	0.117	0.362
	%	**13.21**	**17.87**
i = 50	FCT	0.102	0.272
	%	**24.33**	**38.24**
RC3	FCT	0.076	0.114
	%	**43.54**	**74.35**

(a) Increased InitCwnd



		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.126	0.435
QoS	FCT	0.119	0.411
	%	**5.33**	**5.64**
RC3	FCT	0.078	0.12
	%	**38.31**	**72.43**

(b) Traditional QoS



		Average Over Flows	Average Over Bytes
Regular FCT (s)		0.135	0.443
RCP	FCT	0.088	0.117
	%	**33.86**	**73.58**
RC3	FCT	0.076	0.114
	%	**43.54**	**74.35**

(c) RCP

Fig. 14: RC3 as compared to three alternatives. All FCTs are reported in seconds; % shows percent reduction from baseline. (RTT×BW = 5MB, 30% average link utilization)

to start transmitting at an explicitly allocated rate from the first (post-handshake) RTT, overcoming TCP's slow start penalty. We show the performance improvement for RCP and RC3 in Fig. 14(c). While for large flows, the two schemes are roughly neck-to-neck, RCP actually imposes a penalty for the very smallest (1-4 packet) flows, in part because RCP's explicit rate allocation enforces *pacing* of packets according to the assigned rate, whereas with traditional TCP (and RC3), all packets are transmitted back to back. These results show that RC3 can provide FCTs which are usually comparable or even better than those with RCP. Further, as RC3 can be deployed on legacy hardware and is friendly with existing TCP flows, it is a more deployable path to comparable performance improvements.

Summary: RC3 outperforms traditional QoS and increasing the initial congestion windows; performance with RC3 is on par with RCP without requiring substantial changes to routers.

5.2 Evaluating RC3 Linux Implementation

We now evaluate RC3 using our implementation in the Linux kernel. We extended the Linux 3.2 kernel as described in §4. We did our baseline experiments using both New Reno and CUBIC congestion control mechanisms. We set the send and receive buffer sizes to 2GB, to ensure that an entire flow fits in a single window. We keep the default initial congestion window of 10 [12] in the kernel unchanged.

Our testbed consists of two servers each with two 10Gbps NICs connected to a 10Gbps Arista datacenter switch. As the hosts were physically adjacent, we used *netem* to increase the observed link latency to 10ms, which reflects a short WAN latency.

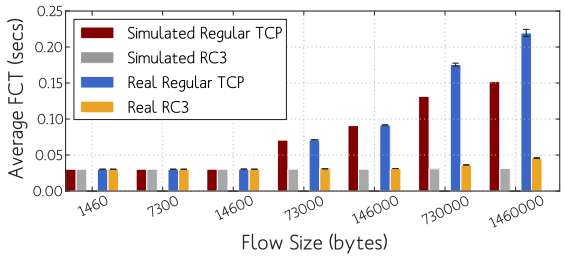
Baseline: Flows with varying sizes were sent from one machine to another. Figure 15(a) shows FCTs with RC3 and baseline TCP implementation in Linux compared to RC3 and baseline TCP NS-3 simulations (with the initial congestion windows set to 10), both running with 10Gbps bandwidth and 20ms RTT. The figure reflects averages over 100 samples.

Overall, RC3 continues to provide strong gains over the baseline TCP design, however, our results in implementation do not tightly match our simulated results from NS. The baseline TCP implementation in Linux performs *worse* than in simulation because of delayed ACK behavior in Linux: when more than two segments are ACKed together, it still only generates an increase in congestion window proportional to two packets being ACKed. This slows down the rate at which the congestion window can increase. The RC3 FCT is slightly higher in Linux than in simulation for large flows because of the extra per-packet overhead in receiving RC3 packets: recall from §4 that RC3 packets are carried over the Linux 'slow path' and thus have slightly higher per-packet overhead.

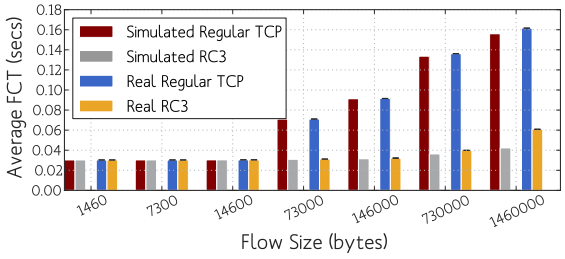
In Figure 15(b), we repeat the same experiment with only 1Gbps bandwidth set by the token bucket filter queueing discipline (retaining 10ms latency through *netem*). In this experiment, results track our simulations more closely. TCP deviates little from the baseline because the arrival rate of the packets ensures that at most two segments are ACKed by the receiver via delayed ACK, and thus the congestion window increases at the correct rate. Overall, we observe that RC3 in implementation continues to provide gains proportional to what we expect from simulation.

While these graphs show the result for New Reno, we repeated these experiments using TCP CUBIC and the FCTs matched very closely to New Reno, since both have the same slow start behavior.

Endhost Correctness: Priority queueing is widely deployed in the OS networking stack, NICs, and routers,

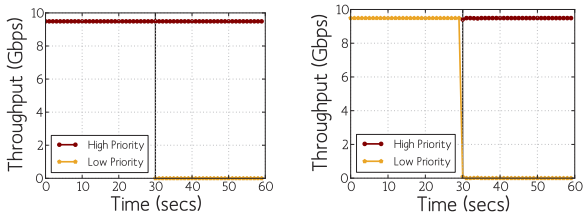


(a) 20ms×10Gbps



(b) 20ms×1Gbps

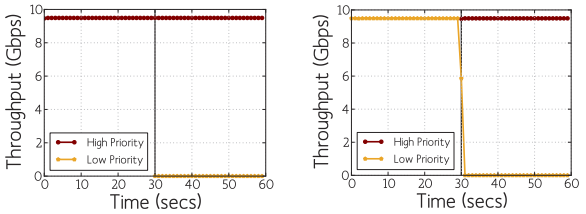
Fig. 15: FCTs for implementation vs. simulation



(a) Low Priority Starts after High Priority

(b) High Priority Starts after Low Priority

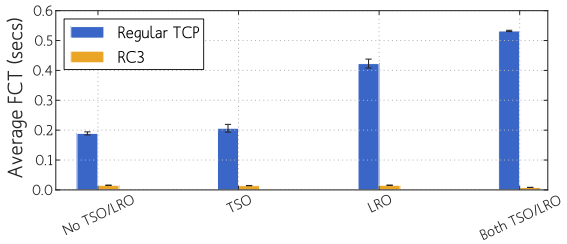
Fig. 16: Correctness of Priority Queuing in Linux



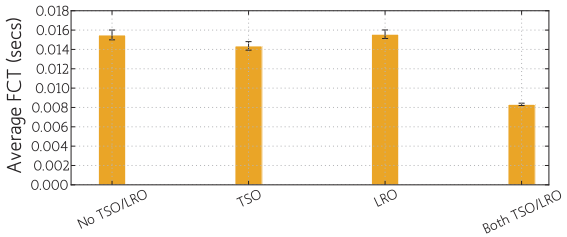
(a) Low Priority Starts after High Priority

(b) High Priority Starts after Low Priority

Fig. 17: Correctness of the Priority Queuing in the Switch



(a) Comparing FCTs for Regular TCP with RC3



(b) Zooming in to observe trends for RC3 FCT

Fig. 18: FCTs for Regular TCP and RC3 with TSO/LRO (20ms×10Gbps)

but is often unused. We now verify the correctness of the prio queueing discipline in Linux. We performed our experiments with iPerf [3] using the default NIC buffer size of 512 packets and with segment offload enabled to achieve a high throughput of 9.5Gbps. All packets in an iPerf flow were assigned the same priority level – this experiment *does not* use RC3 itself. All flows being sent to a particular destination port were marked as priority 1 by changing the DSCP field in the IP header. We connected two endhosts directly, with one acting as the iPerf client sending simultaneous flows to the connected iPerf server (a) with a low priority flow beginning after a high priority flow has begun, and (b) with a high priority flow beginning after a low priority flow has begun. Figure 16 shows that the priority queueing discipline behaves as expected.

Switch Correctness: We extended our topology to connect three endhosts to the switch, two of which acted as iPerf clients, sending simultaneous flows as explained above to the third endhost acting as the iPerf server. Since the two senders were on different machines, prioritization was done by the router. Figure 17 shows that priority queueing at the switch behaves as expected.

Segment and Receiver Offload: In §4.2.3 we discussed how RC3 interacts with segment and receiver offload; we now evaluate the performance of RC3 when combined with these optimizations. For this experiment, we used the same set up, as our baseline and sent a 1000 packet flow without TSO/LRO, with each enabled independently, and with both TSO and LRO enabled. Figure 18 plots the corresponding FCTs excluding the connection set-up time.

For baseline TCP, we see that TSO and LRO each cause a performance penalty in our test scenario. TSO hurts TCP because the increased throughput also increases the number of segments being ACKed with one single delayed ACK, thus slowing the rate at which the congestion window increases. LRO aggravates the same problem by coalescing packets in the receive queue, once again leading them to be ACKed as a batch.

In contrast, RC3's FCTs improve when RC3 is combined with TSO and LRO. TSO and LRO do little to change the performance of RC3, when enabled independently, but when combined they allow chunks of packets to be processed together in batches at the receiver. This reduces the overhead of packet processing by almost 50%, resulting in better overall FCTs.

6 Discussion

Deployment Incentives: For RC3 to be widely used requires ISPs to opt-in by enabling the priority queueing that already exists in their routers. As discussed in the introduction, we believe that giving worse service, rather than better service, for these low priority packets alleviates some of the concerns that has made QoS so hard to offer (in the wide area) today. WQoS is safe and backwards compatible because regular traffic will never be penalized and pricing remains unaffected. Moreover, since RC3 makes more efficient use of bandwidth, it allows providers to run their networks at higher utilization, while still providing good performance, resulting in higher return in investment for their network provisioning.

Partial Support: Our simulations assume that all routers support multiple priorities. If RC3 is to be deployed, it must be usable even when the network is in a state of partial deployment, where some providers but not all support WQoS. When traffic crosses from a network which supports WQoS to a network which does not, a provider has two options: either drop all low priority packets before they cross in to the single-priority domain (obviating the benefits of RC3), or allow the low priority packets to pass through (allowing the packets to subsequently compete with normal TCP traffic at high priority). Simulating this latter scenario, we saw that average FCTs still improved for all flows, from using RC3 when 20% of routers did not support priorities; when 50% of routers did not support priorities small flows experienced a 6-7% FCT penalty, medium-sized flows saw slightly weaker FCT reductions (around 36%), and large flows saw slightly stronger FCT reductions (76-70%).

Middleboxes: Middleboxes which keep tight account of in-flight packets and TCP state are a rare but growing attribute of today's networks. These devices directly challenge the deployment of new protocols; resolving this challenge for proposals like RC3 and others remains an open area of research [13, 20, 27, 29, 31].

Datacenters and Elsewhere: As we've shown via model (§3) and simulation (§5), the benefits of RC3 are strongest in networks with large RTT×BW. Today's datacenter networks typically do not fit this description: with microsecond latencies, RTT×BW is small and thus flows today can very quickly reach 100% link utilization. Nevertheless, given increasing bandwidth, RTT×BW may not remain small forever. In simulations on a fat-tree datacenter topology with (futuristic) 100Gbps links, we observed average FCT improvements of 45% when averaged over flows, and 66% when averaged over bytes. Thus, while RC3 is not a good fit for datacenters today, it may be in the future.

Future: Outside of the datacenter, RTT×BW is already large – and increasing. While increasing TCP's initial congestion window may mitigate the problem in the short term, given the inevitable expansion of available bandwidth, the problem will return again and again with any new choice of new initial congestion window. Our solution, while posing some deployment hurdles, has the advantage of being able to handle future speeds without further modifications.

7 Related Work

Router-assisted Congestion Control: Observing TCP's sometimes poor ability to ensure high link utilization, some have moved away from TCP entirely, designing protocols which use explicit signaling for bandwidth allocation. RCP [11] and XCP [22] are effective protocols in this space. Along similar lines, TCP QuickStart [15] uses an alternate slow-start behavior, which actively requests available capacity from the routers using a new IP Option during the TCP handshake. Using these explicitly supplied rates, a connection can skip slow start entirely and begin sending at its allocated rate immediately following the TCP handshake. Unlike RC3, these algorithms require new router capabilities.

Alternate TCP Designs: There are numerous TCP designs that use alternative congestion avoidance algorithms to TCP New Reno [10, 14, 19, 32, 35]. TCP CUBIC [19] and Compound TCP [32] are deployed in Linux and Windows respectively. Nevertheless, their slow-start behaviors still leave substantial wasted capacity during the first few RTTs – consequently, they could just as easily be used in RC3's primary control loop as TCP New Reno. Indeed, in our implementation we also evaluated TCP CUBIC in combination with RC3.

TCP FastStart [28] targets back-to-back connections, allowing a second connection to re-use cached Cwnd and RTT data from a prior connection. TCP Remy [34] uses machine learning to generate the congestion control algorithm to optimize a given objective function, based on prior knowledge or assumptions about the network. RC3 improves flow completion time even from cold start and without requiring any prior information about the network delay, bandwidth or other parameters.

TCP-Nice [33] and TCP-LP [23] try to utilize the excess bandwidth in the network by using more aggressive back-off algorithms for the low-priority background traffic. RC3 also makes use of the excess bandwidth, but by explicitly using priority queues, with a very different aim of reducing the flow completion time for all flows.

Use of Low Priorities: PFabric [5] is a recent proposal for datacenters that also uses many layers of priorities and ensures high utilization. However, unlike RC3, PFabric's flow scheduling algorithm is targeted exclusively at the datacenter environment, and would not work in the wide-area case.

8 Acknowledgements

We would like to thank all our colleagues in UC Berkeley, for their help and feedback - in particular Sangjin Han, Jon Kuroda, David Zats, Aurojit Panda and Gautam Kumar. We are also very thankful to our anonymous Hotnets 2013 and NSDI 2014 reviewers for their helpful comments and to our shepherd Prof. Srinivasan Seshan for his guidance in shaping the final version of the paper. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1106400.

References

[1] CAIDA Internet Topology Data Kit. `http://goo.gl/QAbecc`.

[2] Internet Traffic Flow Size Analysis. `http://net.doit.wisc.edu/data/flow/size/`.

[3] iPerf. `http://iperf.sourceforge.net/`.

[4] Measuring ISP Topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, 2002.

[5] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing Datacenter Packet Transport. In *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2012.

[6] M. Allman. Comments on bufferbloat. *ACM SIGCOMM Computer Communication Review*, 2013.

[7] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390.

[8] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proc. ACM SIGCOMM*, 2004.

[9] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517.

[10] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. *ACM SIGCOMM Computer Communication Review*, 1994.

[11] N. Dukkipati and N. McKeown. Why Flow-Completion Time is the Right Metric for Congestion Control. *ACM SIGCOMM Computer Communication Review*, 2006.

[12] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An Argument for Increasing TCP's Initial Congestion Window. *ACM SIGCOMM Computer Communication Review*, 2010.

[13] 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 *Proc. SIGCOMM*, 2013.

[14] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649.

[15] S. Floyd, M. Allman, A. Jain, and P. Sarolahti. Quick-Start for TCP and IP. RFC 4782.

[16] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582.

[17] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883.

[18] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-Level Traffic Measurements from the Sprint IP Backbone. *IEEE Network*, 2003.

[19] S. Ha, I. Rhee, and L. Xu. CUBIC: a New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating System Review*, 2008.

[20] M. Honda, Y. Nishida, C. Raiciu, A. Greengalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proc. IMC*, 2011.

[21] S. Iyer, S. Bhattacharyya, N. Taft, and C. Diot. An Approach to Alleviate Link Overload as Observed on an IP Backbone. In *Proc. IEEE INFOCOM*, 2003.

[22] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *Proc. ACM SIGCOMM*, 2002.

[23] A. Kuzmanovic and E. W. Knightly. TCP-LP: Low-priority service via end-Point congestion Control. *In IEEE/ACM ToN*, 2006.

[24] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining tcp congestion control. *ACM SIGCOMM Computer Communication Review*, 1996.

[25] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018.

[26] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *Proc. USENIX NSDI*, 2008.

[27] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Aminy, and B. Fordy. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-compatible with TCP and TLS. In *Proc. USENIX NSDI*, 2012.

[28] V. N. Padmanabhan and R. H. Katz. TCP Fast Start: A Technique for Speeding Up Web Transfers. In *Proc. IEEE Global Internet Conference (GLOBECOM)*, 1998.

[29] 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 *Proc. USENIX NSDI*, 2012.

[30] A. Rao, A. Legout, Y. sup Lim, D. Towlsley, C. Barakat, and W. Dabbous. Network Characteristics of Video Streaming Traffic. In *Proc. ACM CoNeXT*, 2011.

[31] C. Rotsos, H. Howard, D. Sheets, R. Mortier, A. Madhavapeddy, A. Chaudhry, and J. Crowcroft. Lost In the Edge: Finding Your Way With Signposts. In *Proc. USENIX FOCI*, 2013.

[32] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *Proc. IEEE INFOCOM*, 2006.

[33] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp nice: A mechanism for background transfers. In *Proc. USENIX OSDI*, 2002.

[34] K. Winstein and H. Balakrishnan. TCP Ex Machina: Computer-generated Congestion Control. In *Proc. ACM SIGCOMM*, 2013.

[35] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. In *INFOCOM 2004*.

How speedy is SPDY?

Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall
University of Washington

Abstract

SPDY is increasingly being used as an enhancement to HTTP/1.1. To understand its impact on performance, we conduct a systematic study of Web page load time (PLT) under SPDY and compare it to HTTP. To identify the factors that affect PLT, we proceed from simple, synthetic pages to complete page loads based on the top 200 Alexa sites. We find that SPDY provides a significant improvement over HTTP when we ignore dependencies in the page load process and the effects of browser computation. Most SPDY benefits stem from the use of a single TCP connection, but the same feature is also detrimental under high packet loss. Unfortunately, the benefits can be easily overwhelmed by dependencies and computation, reducing the improvements with SPDY to 7% for our lower bandwidth and higher RTT scenarios. We also find that request prioritization is of little help, while server push has good potential; we present a push policy based on dependencies that gives comparable performance to mod_spdy while sending much less data.

1 Introduction

HTTP/1.1 has been used to deliver Web pages using multiple, persistent TCP connections for at least the past decade. Yet as the Web has evolved, it has been criticized for opening too many connections in some settings and too few connections in other settings, not providing sufficient control over the transfer of Web objects, and not supporting various types of compression.

To make the Web faster, Google proposed and deployed a new transport for HTTP messages, called SPDY, starting in 2009. SPDY adds a framing layer for multiplexing concurrent application-level transfers over a single TCP connection, support for prioritization and unsolicited push of Web objects, and a number of other features. SPDY is fast becoming one of the most important protocols for the Web; it is already deployed by many popular websites such as Google, Facebook, and Twitter, and supported by browsers including Chrome, Firefox, and IE 11. Further, IETF is standardizing a HTTP/2.0 proposal that is heavily based on SPDY [10].

Given the central role that SPDY is likely to play in the Web, it is important to understand how SPDY performs relative to HTTP. Unfortunately, the performance of SPDY is not well understood. There have been several studies, predominantly white papers, but the findings often conflict. Some studies show that SPDY improves performance [20, 14], while others show that it provides only a modest improvement [13, 19]. In our own study [25] of page load time (PLT) for the top 200 Web pages from Alexa [1], we found either SPDY or HTTP could provide better performance by a significant margin, with SPDY performing only slightly better than HTTP in the median case.

As we have looked more deeply into the performance of SPDY, we have come to appreciate why it is challenging to understand. Both SPDY and HTTP performance depend on many factors external to the protocols themselves, including network parameters, TCP settings, and Web page characteristics. Any of these factors can have a large impact on performance, and to understand their interplay it is necessary to sweep a large portion of the parameter space. A second challenge is that there is much variability in page load time (PLT). The variability comes not only from random events like network loss, but from browser computation (i.e., JavaScript evaluation and HTML parsing). A third challenge is that dependencies between network activities and browser computation can have a significant impact on PLT [25].

In this work, we present what we believe to be the most in-depth study of page load time under SPDY to date. To make it possible to reproduce experiments, we develop a tool called `Epload` that controls the variability by recording and replaying the process of a page load at fine granularity, complete with browser dependencies and deterministic computational delays; in addition we use a controlled network environment. The other key to our approach is to isolate the different factors that affect PLT with reproducible experiments that progress from simple but unrealistic transfers to full page loads. By looking at results across this progression, we can systematically isolate the impact of the contributing factors and identify when SPDY helps significantly and when it performs poorly compared to HTTP.

Our experiments progress as follows. We first compare SPDY and HTTP simply as a transport protocol (with no browser dependencies or computation) that transfers Web objects from both artificial and real pages (from the top 200 Alexa sites). We use a decision tree analysis to identify the situations in which SPDY outperforms HTTP and vice versa. We find that SPDY improves PLT significantly in a large number of scenarios that track the benefits of using a single TCP connection. Specifically, SPDY helps for small object sizes and under low loss rates by: batching several small objects in a TCP segment; reducing congestion-induced retransmis-

sions; and reducing the time when the TCP pipe is idle. Conversely, SPDY significantly hurts performance under high packet loss for large objects. This is because a set of TCP connections tends to perform better under high packet loss; it is necessary to tune TCP behavior to boost performance.

Next, we examine the complete Web page load process by incorporating dependencies and computational delays. With these factors, the benefits of SPDY are reduced, and can even be negated. This is because: i) there are fewer outstanding objects at a given time; ii) traffic is less bursty; and iii) the impact of the network is degraded by computation. Overall, we find SPDY benefits to be larger when there is less bandwidth and longer RTTs. For these cases SPDY reduces the PLT for 70–80% of Web pages, and for shorter, faster links it has little effect, but it can also increase PLT: the worst 20% of pages see an increase of at least 6% for long RTT networks.

In search of greater benefits, we explore SPDY mechanisms for prioritization and server push. Prioritization helps little because it is limited by load dependencies, but server push has the potential for significant improvements. How to obtain this benefit depends on the server push policy, which is a non-trivial issue because of caching. This leads us to develop a policy based on dependency levels that performs comparably to mod_spdy's policy [11] while pushing 80% less data.

Our contributions are as follows:

- A systematic measurement study using synthetic pages and real pages from 200 popular sites that identifies the combinations of factors for which SPDY improves (and sometimes reduces) PLT compared to HTTP.
- A page load tool, `Epload`, that emulates the detailed page load process of a target page, including its dependencies, while eliminating variability due to browser computation. With a controlled network environment, `Epload` enables reproducible but authentic page load experiments for the first time.
- A SPDY server push policy based on dependency information that provides comparable benefits to mod_spdy while sending much less data over the network.

In the rest of this paper, we first review SPDY background (§2) and then briefly describe our challenge and approach (§3). Next, we extensively study TCP's impact on SPDY (§4) and extend to Web page's impact on SPDY (§5). We discuss in §6, review related work in §7, and conclude in §8.

2 Background

In this section, we review issues with HTTP performance and describe how the new SPDY protocol addresses them.

2.1 Limitations of HTTP/1.1

When HTTP/1.1, or simply HTTP, was designed in the late 1990's, Web applications were fairly simple and rudimentary. Since then, Web pages have become more complex and dynamic, making it difficult for HTTP to meet the increasingly demanding user experience. Below, we identify some of the limitations of HTTP:

i) Browsers open too many TCP connections to load a page. HTTP improves performance by using parallel TCP connections. But if the number of connections is too large, the aggregate flow may cause network congestion, high packet loss, and reduced performance [9]. Further, services often deliver Web objects from multiple domains, which results in even more TCP connections and the possibility of high packet loss.

ii) Web transfers are strictly initiated from the client. Consider the loading of embedded objects. Theoretically, the server can send embedded objects along with the parent object when it receives a request for the parent object. In HTTP, because an object can be sent only in response to a client request, the server has to wait for an explicit request which is sent only after the client has received and processed the parent page.

iii) A TCP segment cannot carry more than one HTTP request or response. HTTP, TCP and other headers could account for a significant portion of a packet when HTTP requests or responses are small. So if there are a large number of small embedded objects in a page, the overhead associated with these headers is substantial.

2.2 SPDY

SPDY addresses several of the issues described above. We now review the key ideas in SPDY's design and implementation and its deployment status.

Design: There are four key SPDY features.

i) Single TCP connection. SPDY opens a single TCP connection to a domain and multiplexes multiple HTTP requests and responses (a.k.a., SPDY streams) over the connection. The multiplexing here is similar to HTTP/1.1 pipelining but is finer-grained. A single connection also helps reduce SSL overhead. Besides client-side benefits, using a single connection helps reduce the number of TCP connections opened at servers.

ii) Request prioritization. Some Web objects, such as JavaScript code modules, are more important than others and thus should be loaded earlier. SPDY allows the client to specify a priority level for each object, which is then used by the server in scheduling the transfer of the object.

iii) Server push. SPDY allows the server to push embedded objects before the client requests for them. This improves latency but could also increase transmitted data if the objects are already cached at the client.

iv) Header compression. SPDY supports HTTP

header compression since experiments suggest that HTTP headers for a single session contain duplicate copies of the same information (e.g., `User-Agent`).

Implementation: SPDY is implemented by adding a framing layer to the network stack between HTTP and the transport layer. Unlike HTTP, SPDY splits HTTP headers and data payloads into two kinds of frames. `SYN_STREAM` frames carry request headers and `SYN_REPLY` frames carry response headers. When a header exceeds the frame size, one or more `HEADERS` frames will follow. HTTP data payloads are sliced into `DATA` frames. There is no standardized value for the frame size, and we find that mod_spdy caps frame size to 4KB [11]. Because frame size is the granularity of multiplexing, too large a frame decreases the ability to multiplex while too small a frame increases overhead. SPDY frames are encapsulated in one or more consecutive TCP segments. A TCP segment can carry multiple SPDY frames, making it possible to batch up small HTTP requests and responses.

Deployment: SPDY is deployed over SSL and TCP. On the client side, SPDY is enabled in Chrome, Firefox, and IE 11. On the server side, popular websites such as Google, Facebook, and Twitter have deployed SPDY. Another popular use of SPDY is between a proxy and a client, such as the Amazon Silk browser [16] and Android Chrome Beta [2]. SPDY version 3 is the most recent specification and is widely deployed [21].

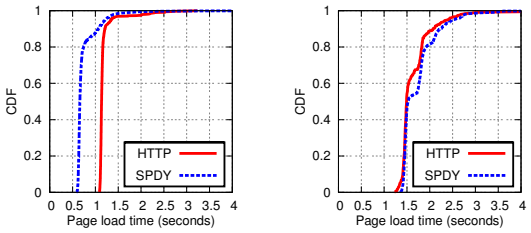
3 Pinning SPDY down

We would like to experimentally evaluate how SPDY performs relative to HTTP because SPDY is likely to play a key role in the Web. But, understanding SPDY performance is hard. Below, we identify three challenges in studying the performance of SPDY and then provide an overview of our approach.

3.1 Challenges

We identify the challenges on the basis of previous studies and our own initial experimentation. As a first step, we extensively load two Web pages for a thousand times using a measurement node at the University of Washington. One page displays fifty world flags [12], which is advertised by mod_spdy [11] to demonstrate the performance benefits of SPDY, and the other is the Twitter home page. The results are depicted in Figure 1.

First, we observe that SPDY helps the flag page but not the Twitter page, and it is not immediately apparent as to why that is the case. Further experimentation in emulated settings also revealed that both the magnitude and the direction of the performance differences vary significantly with network conditions. Taken together, this indicates that SPDY's performance depends on many factors such as Web page characteristics, network parameters, and TCP settings, and that measurement studies will



(a) The flag page (b) Twitter home page

Figure 1: Distributions of PLTs of SPDY and HTTP. Performed a thousand runs for each curve without caching.

likely yield different, even conflicting, results, if they use different experimental settings. Therefore, a comprehensive sweep of the parameter space is necessary to evaluate under what conditions SPDY helps, what kinds of Web pages benefit most from SPDY, and what parameters best support SPDY.

Second, we observed in our experiments that the measured page load times have high variances, and this often overwhelms the differences between SPDY and HTTP. For example, in Figure 1(b), the variance of the PLT for the Twitter page is 0.5 second but the PLT difference between HTTP and SPDY is only 0.02 second. We observe high variance even when we load the two pages in a fully controlled network. This indicates that the variability likely stems from browser computation (i.e., JavaScript evaluation and HTML parsing). Controlling this variability is key to reproducing experiments so as to obtain meaningful comparisons.

Third, prior work has shown that the dependencies between network operations and computation has a significant impact on PLT [25]. Interestingly, page dependencies also influence the scheduling of network traffic and affects how much SPDY helps or hurts performance (§4 and §5). Thus, on one hand, ignoring browser computations can reduce PLT variability, but on the other hand, dependencies need to be preserved in order to obtain accurate measurements under realistic offered loads.

3.2 Approach

Our approach is to separate the various factors that affect SPDY and study them in isolation. This allows us to control and identify the extent to which these factors affect SPDY.

First, we extensively sweep the parameter space of all the factors that affect SPDY including RTT, bandwidth, loss rate, TCP initial window, number of objects on a page, and object sizes. We initially *ignore* page load dependencies and computation in order to simplify our analysis. This systematic study allows us to identify when SPDY helps or hurts and characterize the importance of the contributing factors. Based on further analysis of why SPDY sometimes hurts, we propose some

simple modifications to TCP.

Second, before we perform experiments *with* page load dependencies, we address the variability caused by computation. We develop a tool called `Epload` that emulates the process of a page load. Instead of performing real browser computation, `Epload` records the process of a sample page load, identifies when computations happen, and replays the page load by introducing the appropriate delays associated with the recorded computations. After emulating a computation activity, `Epload` performs real network requests to dependent Web objects. This allows us to control the variability of computation while also modeling page load dependencies. In contrast to the methodology that statistically reduces variability by obtaining a large amount of data (usually from production), our methodology mitigates the root cause of variability and thus largely reduces the amount of required experiments.

Third, we study the effects of dependencies and computation by performing page loads with `Epload`. We are then able to identify how much dependencies and computation affect SPDY, and to identify the relative importance of other contributing factors. To mitigate the negative impact of dependencies and computation, we explore the use of prioritization and server push that enable the client and the server to coordinate the transfers. Here, we are able to evaluate the extent to which these mechanisms can improve performance when used appropriately.

4 TCP and SPDY

In this section, we extensively study the performance of SPDY as a transfer protocol on both synthetic and real pages by ignoring page load dependencies and computation. This allows us to measure SPDY performance without other confounding factors such as browser computation and page load dependencies. Here, SPDY is only different from HTTP in the use of a single TCP connection, header compression, and a framing layer.

4.1 Experimental setup

We conduct the experiments by setting up a client and a server that can communicate over both HTTP and SPDY. Both the server and the client are connected to the campus LAN at the University of Washington. We use Dummynet [6] to vary network parameters. Below details the experimental setup.

Server: Our server is a 64-bit machine with 2.4GHz 16 core CPU and 16GB memory. It runs Ubuntu 12.04 with Linux kernel 3.7.5 using the default TCP variant Cubic. We use a TCP initial window size of ten as the default setting, as suggested by SPDY best practices [18]. HTTP and SPDY are enabled on Apache 2.2.2 with the SPDY module, mod_spdy 0.9.3.3-386, installed. We use SPDY 3 without SSL which allows us to decode the SPDY frames in TCP payloads. To control the exact size of Web objects, we turn off gzip encoding.

Categ	Factor	Range	High
Net	rtt	20ms, 100ms, 200ms	$\geq$100ms
	bw	1Mbps, 10Mbps	$\geq$10Mbps
	pkt loss	0, 0.005, 0.01, 0.02	$\geq$ 0.01
TCP	iw	3, 10, 21, 32	$\geq$ 21
Page	obj size	100B, 1K, 10K, 100K, 1M	$\geq$ 1K
	# of obj	2, 8, 16, 32, 64, 128, 512	$\geq$ 64

Table 1: Contributing factors to SPDY performance. We define a threshold for each factor, so that we can classify a setting as being high or low in our analysis.

Client: Because we issue requests at the granularity of Web objects and not pages, we do not work with browsers, and instead develop our own SPDY client by following the SPDY/3 specification [21]. Unlike other wget-like SPDY clients such as `spdylay` [22] that open a TCP connection per request, our SPDY client allows us to reuse TCP connections. Similarly, we also develop an HTTP client for comparison. We set the maximum number of parallel TCP connections for HTTP to six, as used by all major browsers. As the receive window is autotuned, it is not a bottleneck in our experiments.

Web pages: To experiment with synthetic pages, we create objects with pre-specified sizes and numbers. To experiment with real pages, we download the home pages of the Alexa top 200 websites to our own server. To avoid the negative impact of domain sharding on SPDY [18], we serve all embedded objects from the same server including those that are dynamically generated by JavaScript.

We run the experiments presented in the entire paper from June to September, 2013. We repeat our experiments five times and present the median to exclude the effects of random loss. We collect network traces at both the client and the server. We define page load time (PLT) as the elapsed time between when the first object is requested and when the last object is received. Because we do not experiment within a browser, we do not use the W3C `load` event [24].

4.2 Experimenting with synthetic pages

In experimenting with synthetic pages, we consider a broad range of parameter settings for the various factors that affect performance. Table 1 summarizes the parameter space used in our experiments. The RTT values include 20ms (intra-coast), 100ms (inter-coast), and 200ms (3G link or cross-continent). The bandwidths emulate a broadband link with 10Mbps [4] and a 3G link with 1Mbps [3]. We inject random packet loss rates from zero to 2% since studies suggest that Google servers experience a loss rate between 1% and 2% [5]. At the server,

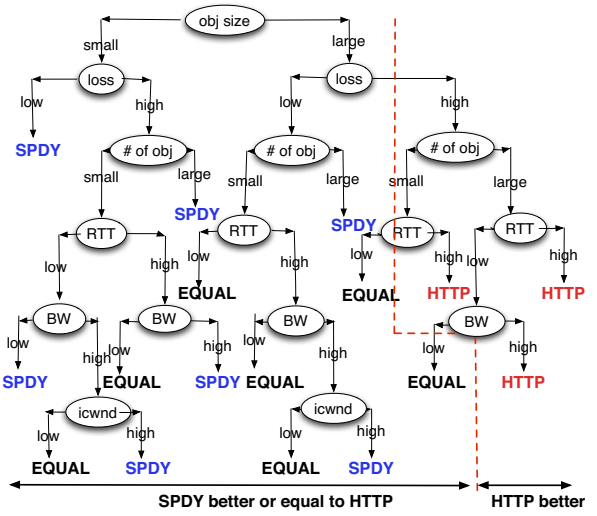


Figure 2: The decision tree that tells when SPDY or HTTP helps. A leaf pointing to SPDY (HTTP) means SPDY (HTTP) helps; a leaf pointing to EQUAL means SPDY and HTTP are comparable. Table 1 shows how we define a factor being high or low.

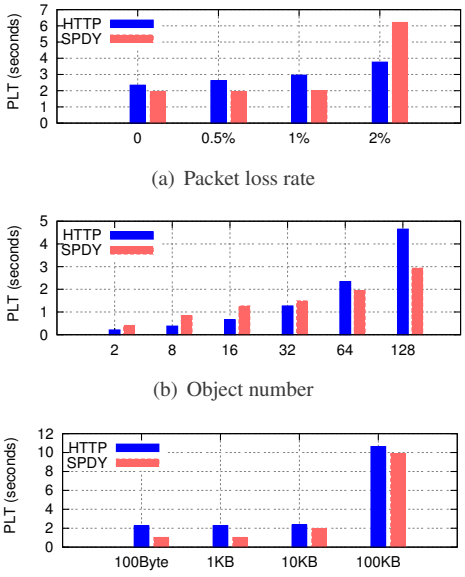
we vary TCP initial window size from 3 (used by earlier Linux kernel versions) to 32 (used by Google servers). We also consider a wide range of Web object sizes (100B to 1M) and object numbers (2 to 512). For simplicity, we choose one value for each factor which means that there is no cross traffic.

When we sweep this large parameter space, we find that SPDY improves performance under certain conditions, but degrades performance under other conditions.

4.2.1 When does SPDY help or hurt

There have been many hypotheses as to whether SPDY helps or hurts based on analytical inference about parallel versus single TCP connections. For example, one hypothesis is that SPDY hurts because a single TCP connection increases congestion window slower than multiple connections; another hypothesis is that SPDY helps stragglers because HTTP has to balance its communications across parallel TCP. However, it is unclear how much hypotheses contribute to SPDY performance. Here, we sort out the most important findings, meaning that hypotheses that are shown here contribute more to SPDY performance than those that are not shown.

Methodology: To understand the conditions under which SPDY helps or hurts, we build a predictive model based on decision tree analysis. In the analysis, each configuration is a combination of values for all factors listed in Table 1. For each configuration, we add an additional variable s, which is the PLT of SPDY divided by that of HTTP. We run the decision tree to predict the configurations under which SPDY outperforms HTTP ($s < 0.9$) and under which HTTP outperforms SPDY ($s > 1.1$). The decision tree analysis generates the likelihood that a configuration works better under SPDY (or HTTP). If this likelihood is over 0.75, we mark the branch as SPDY (or HTTP); otherwise, we say that SPDY and HTTP perform equally.



(a) Packet loss rate

(b) Object number

(c) Object size

Figure 3: Performance trends for three factors with a default setting: rtt=200ms, bw=10Mbps, loss=0, iw=10, obj_size=10K, obj_number=64.

We obtain the decision tree in Figure 2 as follows. First, we produce a decision tree based on all the factors. To populate the branches, we also generate supplemental decision trees based on subsets of factors. Each supplemental decision tree has a prediction accuracy of 84% or higher. Last, we merge the branches from supplemental decision trees into the original decision tree.

Results: The decision tree shows that SPDY hurts when packet loss is high. However, SPDY helps under a number of conditions, for example, when there are:

- Many small objects, or small objects under low loss.
- Many large objects under low loss.
- Few objects under good network conditions and a large TCP initial window.

The decision tree also depicts the relative importance of contributing factors. Intuitively, factors close to the root of the decision tree affect SPDY performance more than those near the leaves. This is because the decision tree places the important factors near the root to reduce the number of branches. We find that object size and loss rate are the most important factors in predicting SPDY performance. However, RTT, bandwidth, and TCP initial window play a less important role.

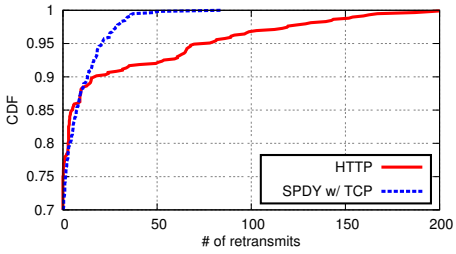


Figure 4: SPDY reduces the number of retransmissions.

How much SPDY helps or hurts: We present three trending graphs in Figure 3. Figure 3(a) shows that HTTP outperforms SPDY by half when loss rate increases to 2%, Figure 3(b) shows the trend that SPDY performs better as the number of objects increases, and Figure 3(c) shows the trend that SPDY performs worse as the object size increases. We publish the results, trends, and network traces at `http://wprof.cs.washington.edu/spdy/`.

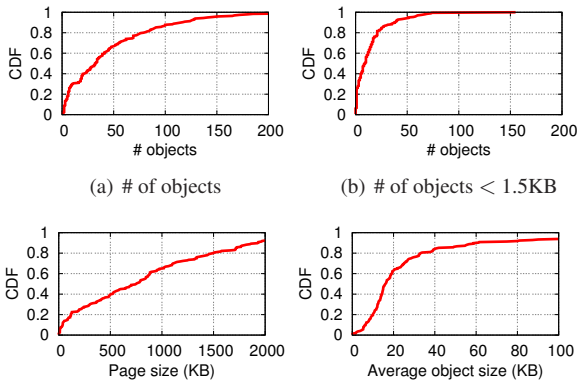
4.2.2 Why does SPDY help or hurt

While the decision tree informs the conditions under which SPDY helps or hurts, it does not explain why. To this end, we analyze the network traces we collected to explain SPDY performance. We discuss below our findings.

SPDY helps on small objects. Our traces suggest that TCP implements congestion control by counting outstanding packets not bytes. Thus, sending a few small objects with HTTP will promptly use up the congestion window, though outstanding bytes are far below the window limit. In contrast, SPDY batches small objects and thus eliminates this problem. This explains why the flag page [12], which mod_spdy advertised, benefits from SPDY.

SPDY benefits from having a single connection. We find several reasons as to why SPDY benefits from a single TCP connection. First, a single connection results in fewer retransmissions. Figure 4 shows the retransmissions in SPDY and HTTP across all configurations except those with zero injected loss. SPDY helps because packet loss occurs more often when concurrent TCP connections are competing with each other. There are additional explanations for why SPDY benefits from using a single connection. In our previous study [25], our experiments showed that SPDY significantly reduced the contribution of the TCP connection setup time to the *critical path* of a page download. Further, our experiments in §5 will show that a single pipe reduces the amount of time the pipe is idle due to delayed client requests.

SPDY degrades under high loss due to the use of a single pipe. We discussed above that a single TCP connection helps under several conditions. However, a single connection hurts under high packet loss because it aggressively reduces the congestion window compared to HTTP which reduces the congestion window on only one of its parallel connections.



(a) # of objects (b) # of objects < 1.5KB

(c) Page size (KB) (d) Mean object size (KB)

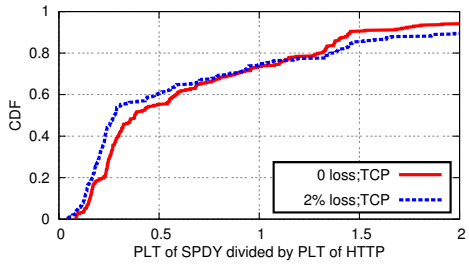
Figure 5: Characteristics of top 200 Alexa Web pages.

4.3 Experimenting with real pages

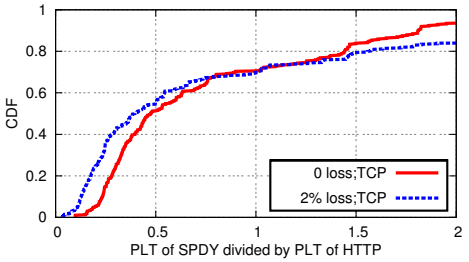
In this section, we study the effects of varying object sizes and number of objects based on the distributions observed in real Web pages. We continue to vary other factors such as network conditions and TCP settings based on the parameter space described in Table 1. Due to space limit, we only show results under a 10Mbps bandwidth.

First, we examine the page characteristics of real pages because they can explain why SPDY helps or hurts when we relate them to the decision tree. Figure 5 shows the characteristics of the top 200 Alexa Web pages [1]. The median number of objects is 30 and the median page size is 750KB. We find high variability in the size of objects within a page. The standard deviation of the object size within a page is 31KB (median), even more than the average object size 17KB (median).

Figure 6 shows PLT of SPDY divided by that of HTTP across the 200 Web pages. It suggests that SPDY helps on 70% of the pages consistently across network conditions. Interestingly, SPDY shows a 2x speedup over half of the pages, likely due to the following reasons. First, SPDY almost eliminates retransmissions (as indicated in Figure 7). Compared to a similar analysis for artificial pages (see Figure 4), SPDY's retransmission rate is even lower. Second, we find in Figure 5(b) that 80% of the pages have small objects, and that half of the pages have more than ten small objects. Since SPDY helps with small objects (based on the decision tree analysis), it is not surprising that SPDY has lower PLT for this set of experiments. In addition, we hypothesize that SPDY could help with stragglers since it multiplexes all objects on to a single connection and thus reduces the dynamics of congestion windows. To check this hypothesis, we ran a set of experiments with overall page size and the



(a) rtt=20ms, bw=10Mbps



(b) rtt=200ms, bw=10Mbps

Figure 6: SPDY performance across 200 pages with object sizes and numbers of objects drawn from real pages. SPDY helps more under a 1Mbps bandwidth.

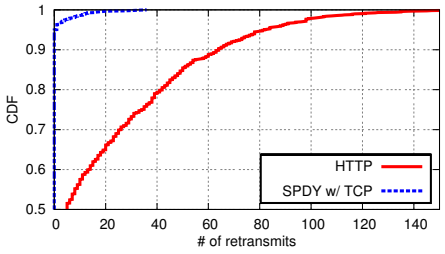


Figure 7: SPDY helps reduce retransmissions.

number of objects drawn from the real pages, but with equal object sizes embedded inside the pages. When we perform this experiment, HTTP's performance improves only marginally indicating that there is very little straggler effect.

4.4 TCP modifications

Previously, we found that SPDY hurts mainly under high packet loss because a single TCP connection reduces the congestion window more aggressively than HTTP's parallel connections. Here, we demonstrate that the negative impact can be mitigated by simple TCP modifications.

Our modification (a.k.a., TCP+) mimics behaviors of concurrent connections with a single connection. Let the number of parallel TCP connections be n. First, we propose to multiply the initial window by n to reduce the effect of slow start. Second, we suggest scaling the receive window by n to ensure that the SPDY connection has the same amount of receive buffer as HTTP's parallel connections. Third, when packet loss occurs, the congestion window (*cwnd*) backs off with a rate $\beta' = 1-(1-\beta)/n$ where β is the original backoff rate. In practice, the number of concurrent connections changes over time. Because we are unable to pass this value to the Linux kernel in real time, we assume that HTTP uses six connections and set $n = 6$. We use six here because it is found optimal and used by major browsers [17].

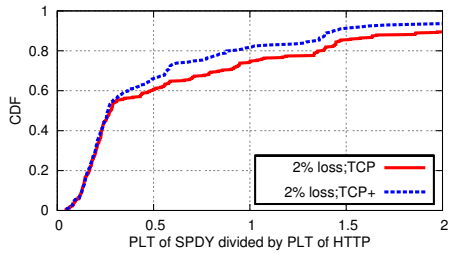


Figure 8: TCP+ helps SPDY across the 200 pages. RTT=20ms, BW=10Mbps. Results on other network settings are similar.

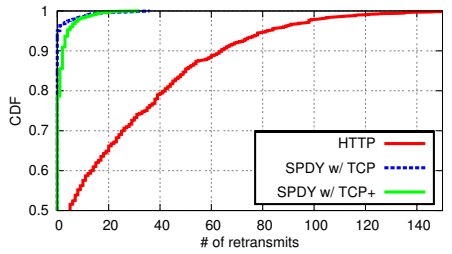


Figure 9: With TCP+, SPDY still produces few retransmissions.

We perform the same set of SPDY experiments with both synthetic and real pages using TCP+. Figure 8 shows that SPDY performs better with TCP+, and the decision tree analysis for TCP+ suggests that loss rate is no longer a key factor that determines SPDY performance.

To evaluate the potential side effects of TCP+, we look at the number of retransmissions produced by TCP+. Figure 9 shows that SPDY still produces much fewer retransmissions with TCP+ than with HTTP, meaning that TCP+ does not abuse the congestion window under the conditions that we experimented with. Here, we aim to demonstrate that SPDY's negative impact under high random loss can be mitigated by tuning the congestion window. Because the loss patterns in real networks are likely more complex, a solution for real networks requires further consideration and extensive evaluations and is out of the scope of this paper.

5 Web pages and SPDY

This section examines how SPDY performs for real Web pages. Real page loads incur dependencies and computation that may affect SPDY's performance. To incorporate dependencies and computation while controlling variability, we develop a page load emulator `Epload`

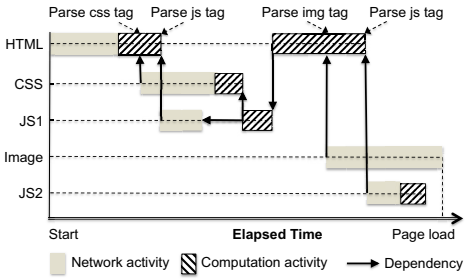


Figure 10: A dependency graph obtained from WProf.

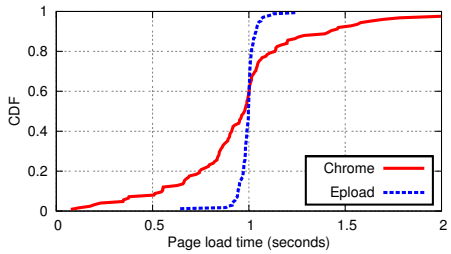


Figure 11: Page loads using Chrome v.s. Epload.

that hides the complexity and variations in browser computation while performing authentic network requests (§5.1). We use `Epload` to identify the effect of page load dependencies and computation on SPDY's performance (§5.2). We further study SPDY's potential by examining prioritization and server push (§5.3).

5.1 Epload: emulating page loads

Web objects in a page are usually not loaded at the same time, because loading an object can depend on loading or evaluating other objects. Therefore, not only network conditions, but also page load dependencies and browser computation, affect page load times. To study how much SPDY helps the overall page load time, we need to evaluate SPDY's performance by preserving dependencies and computation of real page loads.

Dependencies and computation are naturally preserved by loading pages in real browsers. However, this procedure incurs high variances in page load times that stem from both network conditions and browser computation. We have conducted controlled experiments to control the variability of network, and here introduce the `Epload` emulator to control the variability of computation.

Design: The key idea of `Epload` is to decouple network operations and computation in page loads. This allows `Epload` to simplify computation while scheduling network requests at the appropriate points during the page load.

`Epload` records the process of a page load by capturing the dependency graph using our previous work, WProf [25]. WProf captures the dependency and timing information of a page load. Figure 10 shows an example of a dependency graph obtained from WProf where activities depend on each other. This Web page embeds a CSS, a JavaScript, an image, and another JavaScript. A bar represents an activity (i.e., loading objects, evaluating CSS and JavaScript, parsing HTML) while an arrow represents that one activity depends on another. For example, evaluating JS1 depends on both loading JS1 and evaluating CSS. Therefore, evaluating JS1 can only start after the other two activities complete. There are other dependencies such as layout and painting. Because they do not occur deterministically and significantly, we exclude them here.

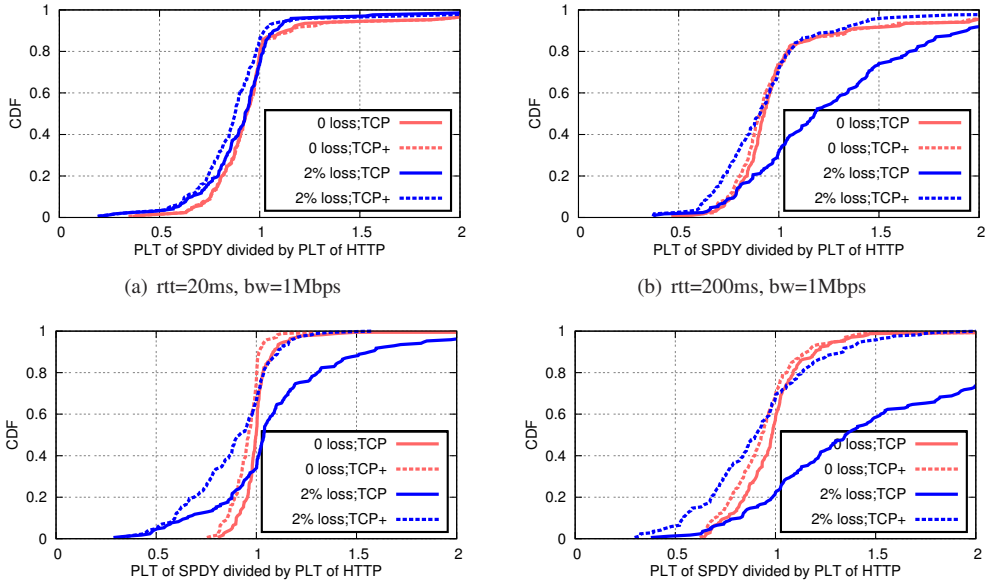
Using the recorded dependency graph, `Epload` replays the page load process as follows. First, `Epload` starts the activity that loads the root HTML. When the activity is finished, `Epload` checks whether it should trigger a dependent activity based on whether all activities that the dependent activity depends on are finished. For example in Figure 10, the dependent activity is parsing the HTML, and it should be triggered. Next, it starts the activity that parses the HTML. Instead of performing HTML parsing, it waits for the same amount of time that parsing takes (based on the recorded information) and checks dependent activities upon completion. This proceeds until all activities are finished. The actual replay process is more complex because a dependent activity can start before an activity is fully completed. For example, parsing an HTML starts after the first chunk of the HTTP response is received; and loading the CSS starts after the first chunk of HTML is fully parsed. `Epload` models all of these aspects of a page load.

Implementation: `Epload` recorder is implemented based on WProf to generate a dependency graph that specifies activities and their dependencies. `Epload` records the computational delays while performing the page load in the browser, whereas the network delays are realized independently for each replay run. We implement `Epload` replayer using node.js. The output from `Epload` replayer is a series of throttled HTTP or SPDY requests to perform a page load. The `Epload` code is available at `http://wprof.cs.washington.edu/spdy/`.

Evaluation: We validate that `Epload` controls the variability of computation. We compare the differences of two runs across 200 pages loaded by `Epload` and by Chrome. The network is tuned to a 20ms RTT, a 10Mbps bandwidth, and zero loss. Figure 11 shows that `Epload` produces at most 5% differences for over 80% of pages which is a 90% reduction compared to Chrome.

5.2 Effects of dependencies and computation

We use `Epload` to measure the impact of dependencies and computation. We set up experiments as follows. The `Epload` recorder uses a WProf-instrumented Chrome to



(a) rtt=20ms, bw=1Mbps

(b) rtt=200ms, bw=1Mbps

(c) rtt=20ms, bw=10Mbps

(d) rtt=200ms, bw=10Mbps

Figure 12: SPDY performance using emulated page loads. Compared to Figure 6, it suggests that dependencies and computation reduce the impact of SPDY and that RTT and bandwidth become more important.

obtain the dependency graphs of the top 200 Alexa Web pages [1]. `Epload` runs on a Mac with 2GHz dual core CPU and 4GB memory. We vary other factors based on the parameter space described in Table 1. Due to space limit, we only show figures under a 10Mbps bandwidth.

Figure 12 shows the performance of SPDY versus HTTP after incorporating dependencies and computation. Compared to Figure 6, dependencies and computation largely reduce the amount that SPDY helps or hurts. We make the following observations along with supporting evidence. First, computation and dependencies increase PLTs of both HTTP and SPDY, reducing the network load. Second, SPDY reduces the amount of time a connection is idle, lowering the possibility of slow start (see Figure 13). Third, dependencies help HTTP by making traffic less bursty, resulting in fewer retransmissions (see Figure 14). Fourth, having fewer outstanding objects diminishes SPDY's gains, because SPDY helps more when there are a large number of outstanding objects (as suggested by the decision tree in Figure 2). Here, we see that dependencies and computation reduce and can easily nullify the benefits of SPDY, implying that speeding up computation or breaking dependencies might be necessary to improve the PLT using SPDY.

Interestingly, we find that RTT and bandwidth now play a more important role in the performance of SPDY. For example, Figure 12 shows that SPDY helps up to 80% of the pages under low bandwidths, but only 55% of the pages under high bandwidths. This is because RTT and bandwidth determine the amount of time page loads spend in network relative to computation, and further the

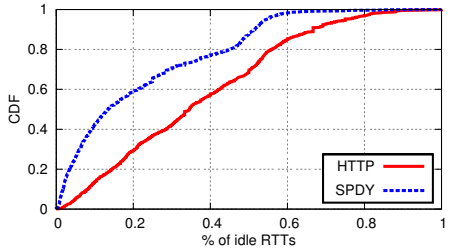


Figure 13: Fractions of RTTs when a TCP connection is idle. Experimented under 2% loss rate.

amount of impact that computation has on SPDY. This explains why SPDY provides minimal improvements under good network conditions (see Figure 12(c)).

To identify the impact of computation, we scale the time spent in each computation activity by factors of 0, 0.5, and 2. Figure 15 shows the performance of SPDY versus HTTP, both with scaled computation and under high bandwidths, suggesting that speeding up computation increases the impact of SPDY. Surprisingly, speeding up computation to the extreme is sometimes no better than a x2 speedup. This is because computation delays the requesting of dependent objects which allows for previously requested objects to be loaded faster, and therefore possibly lowers the PLT.

5.3 Advancing SPDY

SPDY provides two mechanisms, i) prioritization and ii) server push, to mitigate the negative effects of dependencies and computation of real page loads. However, little is known about how to better use the mechanisms. In this section, we explore advanced policies to speed up page

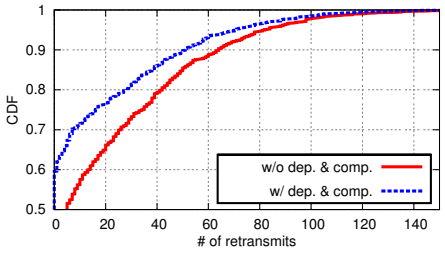


Figure 14: SPDY helps reduce retransmissions.

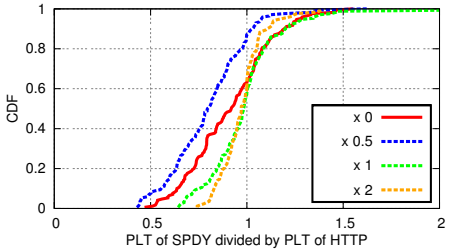


Figure 15: Results by varying computation when bw=10Mbps, rtt=200ms.

loads using these mechanisms.

5.3.1 Basis of advancing

To better schedule objects, both prioritization and server push provide mechanisms to specify the importance for each object. Thus, the key issue is to identify the importance of objects in an automatic manner. To highlight the benefits, we leverage the dependency information obtained from a previous load of the same page. This information gives us ground truth as to which objects are critical for reducing PLT. For example, in Figure 10, all the activities depend on loading the HTML, making HTML the most important object; but no activity depends on loading the image, suggesting that the image is not an important object.

To quantify the importance of an object, we first look at the time required to finish the page load starting from the load of this object. We denote this as time to finish (TTF). In Figure 10, TTF of the image is simply the time to load the image alone, while TTF of JS2 is the time to both load and evaluate it. Because TTF of the image is longer than TTF of JS2, this image is more important than JS2. Unfortunately in practice, it is not clear as to how long it would take to load an object, before we make the decision to prioritize or push it.

Therefore, we simplify the definition of importance. First, we convert the activity-based dependency graph to an object-based graph by eliminating computation while preserving dependencies (Figure 16). Second, we calculate the longest path from each object to the leaf objects; this process is equivalent to calculating node depths of a directed acyclic graph. Figure 16 (right) shows an example of assigned depths. Note that the depth here equals

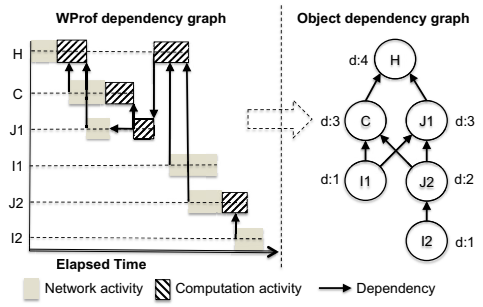
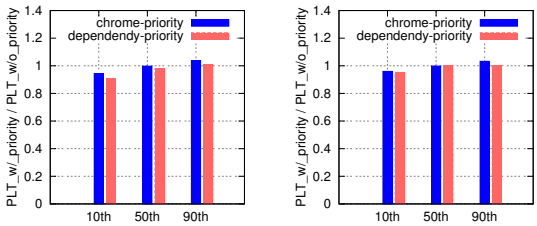


Figure 16: Converting WProf dependency graph to an object-based graph. Calculating a depth to each object in the object-based graph.



(a) rtt=20ms, bw=10Mbps (b) rtt=200ms, bw=10Mbps

Figure 17: Results of priority (zero packet loss) when bw=10Mbps. bw=1Mbps results are similar to (b).

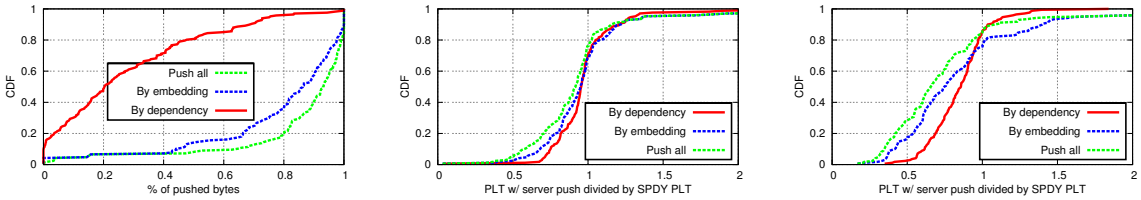
TTF if we ignore computation and suppose that the load of each object takes the same amount of time.

We use this depth information to prioritize and push objects. This implies that the browser or the server should know this beforehand. We provide a tool to let Web developers measure the depth information for objects transported by their pages.

5.3.2 Prioritization

SPDY/3 allows eight priority levels for clients to use when requesting objects. SPDY best practices website [18] recommends prioritizing HTML over CSS/JavaScript and CSS/JS over the rest (`chrome-priority`). Our priority levels are obtained by linearly mapping the depth information computed above (`dependency-priority`).

We compare the two prioritization policies to baseline SPDY in Figure 17. Interestingly, we find that there is almost no benefit by using `chrome-priority` while `dependency-policy` marginally helps under a 20ms RTT. The impact of *explicit* prioritization is minimal because the dependency graph has already *implicitly* prioritized objects. Implicit prioritization results from browser policies, independent of Web pages themselves. For example in Figure 10, all other objects cannot be loaded before HTML; Image and JS2 cannot be loaded before CSS and JS1. As dependencies limit the impact of SPDY, prioritization cannot break dependencies, and thus is unlikely to improve SPDY's PLT.



(a) Pushed bytes (b) rtt=20ms, bw=10Mbps (c) rtt=200ms, bw=10Mbps

Figure 18: Results of server push when bw=10Mbps.

5.3.3 Server push

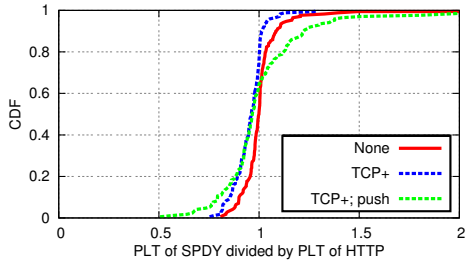
SPDY allows servers to push objects to save round trips. However, server push is non-trivial because there is a tension between making page loads faster and wasting bandwidth. Particularly, one should not overuse server push if pushed objects are already cached. Thus, the key goal is to speed up page loads while keeping the cost low.

We find no standard or best practices guidance from Google on how to do server push. Mod_spdy can be configured to push up to an *embedding level*, which is defined as follows: the root HTML page is at embedding level 0; objects at embedding level i are those whose URLs are embedded in objects at embedding level $i-1$. An alternative policy is to push based on the depth information.

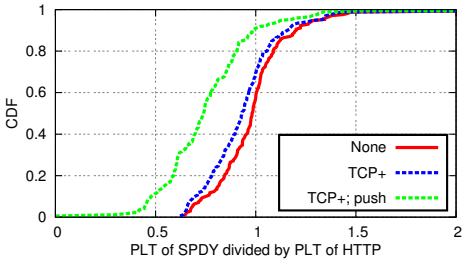
Figure 18 shows server push performance (i.e., push all objects, one embedding level, and one dependency level) compared to baseline SPDY. We find that server push helps, especially under high RTT. We also find that pushing by dependency incurs comparable speedups to pushing by embedding, while benefiting from a 80% reduction in pushed bytes (Figure 18(a)). Note that server push does not always help because pushed objects share bandwidth with more important objects. In contrast to prioritization, server push can help because it breaks dependencies which limits the performance gains of SPDY.

5.4 Putting it all together

We now pool together the various enhancements (i.e., TCP+ and server push by one dependency level). Figure 19 shows that this improves SPDY by 30% under high RTTs. But this improvement largely diminishes under low RTTs where computation dominates page load times.



(a) rtt=20ms, bw=10Mbps



(b) rtt=200ms, bw=10Mbps

Figure 19: Put all together when bw=10Mbps.

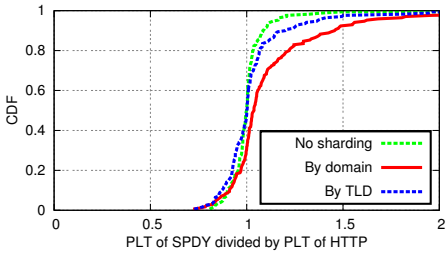


Figure 20: Results of domain shading when bw=10Mbps and rtt=20ms.

6 Discussions

SPDY in the wild: To evaluate SPDY in the wild, we place clients at Virginia (US-East), North California (US-West), and Ireland (Europe) using Amazon EC2 micro-instances. We add explanatory power by periodically probing network parameters between clients and the server, and find that RTTs are consistent: 22ms (US-East), 71ms (US-West), and 168ms (Europe). For all vantage points, bandwidths are high (10Mbps to 143Mbps) and loss rates are extremely low. These network parameters well explain our SPDY evaluations in the wild (not shown due to space limit) that are similar to synthetic ones under high bandwidths and low loss rates. The evaluations here are preliminary and covering a complete set of scenarios would be future work.

Domain sharding: As suggested by SPDY best practices [18], we used a single connection to fetch all the objects of a page to eliminate the negative impact of domain sharing. In practice, migrating objects to one domain suffers from deployment issues given popular uses of third parties (e.g., CDNs, Ads, and Analytics). To this

end, we evaluate situations when objects are distributed to multiple servers that cooperatively use SPDY. We distribute objects by full domain to represent the state-of-the-art of domain sharding. We also distribute objects by top-level domain (TLD). This demonstrates the situation when websites have eliminated domain sharding but still use third-party services. Figure 20 compares SPDY performance under these object distributions. We find that domain sharding hurts as expected but hosting objects by TLD is comparable to using one connection, suggesting that SPDY's performance does not degrade much when some portions of the page are provided by third-party services.

SSL: SSL adds overhead to page loads which can degrade the impact of SPDY, but it keeps the handshake overhead low by using a single connection. We conduct our experiments using SSL and find that the overhead of SSL is too small to affect SPDY's performance.

Mobile: We perform a small set of SPDY measurements under mobile environments. We assume large RTTs, low bandwidths, high losses, and large computational delays, as suggested by related literature [3, 26]. Results with simulated slow networks suggest that SPDY helps more but also hurts more. It also shows that prioritization and server push by dependency help less (not shown due to space limit). However, large computational delays on mobile devices reduce the benefits provided by SPDY. This means that the benefits of SPDY under mobile scenarios depends on the relative changes in performance of the network and computation. Further studies on real mobile devices and networks would advance the understanding in this space.

Limitations: Our work does not consider a number of aspects. First, we did not evaluate the effects of header compression because it is not expected to provide significant benefits. Second, we did not evaluate dynamic pages which take more time in server processing. Similar to browser computation, server processing will likely reduce the impact of SPDY. Last, we are unable to evaluate SPDY under production servers where network is heavily used.

7 Related Work

SPDY studies: Erman et al. [7] studied SPDY in the wild on 20 Web pages by using cellular connections and SPDY proxies. They found that SPDY performed poorly while interacting with radios due to a large body of unnecessary retransmissions. We used more reliable connections, enabled SPDY on servers, and swept a more complete parameter space. Other SPDY studies include the SPDY white paper [20] and measurements by Microsoft [14], Akamai [13], and Cable Labs [19]. The SPDY white paper shows a 27% to 60% speedup for SPDY, but the other studies show that SPDY helps only marginally. While providing invaluable measurements, these studies look at a limited parameter space. Studies by Microsoft [14] and Cable Labs [19] only measured single Web pages and the other studies consider only a limited set of network conditions. Our study extensively swept the parameter space including network parameters, TCP settings, and Web page characteristics. We are the first to isolate the effect of dependencies, which are found to limit the impact of SPDY.

TCP enhancements for the Web: Google have proposed and deployed several TCP enhancements to make the Web faster. TCP fast open eliminates the TCP connection setup time by sending application data in the SYN packet [15]. Proportional rate reduction smoothly backs off congestion window to transmit more data under packet loss [5]. Tail loss probe [23] and other measurement-driven enhancements described in [8] mitigated or eliminated loss recovery by retransmission timeout. Our TCP modifications are specific to SPDY and are orthogonal to Google's proposals.

Advanced SPDY mechanisms: There are no recommended policies on how to use the server push mechanism. We find that mod_spdy [11] implements server push by embedding levels. However, we find that this push policy wastes bandwidths. We provide a server push policy based on dependency levels that performs comparably to mod_spdy's while pushing 80% less data.

8 Conclusion

Our experiments and prior work show that SPDY can either help or sometimes hurt the load times of real Web pages by browsers compared to using HTTP. To learn which factors lead to performance improvements, we start with simple, synthetic page loads and progressively add key features of the real page load process. We find that most of the performance impact of SPDY comes from its use of a single TCP connection: when there is little network loss a single connection tends to perform well, but when there is high loss a set of connections tend to perform better. However, the benefits from a single TCP connection can be easily overwhelmed by dependencies in real Web pages and browser computation. We conclude that further benefits in PLT will require changes to restructure the page load process, such as the server push feature of SPDY, as well as careful configuration at the TCP level to ensure good network performance.

Acknowledgements

We thank Will Chan, Yu-Chung Cheng, and Roberto Peon from Google, our shepherd, Sanjay Rao, and the anonymous reviewers for their feedback. We thank Ruhui Yan for helping analyze packet traces.

References

[1] Alexa - The Web Information Company. http://www.alexa.com/topsites/countries/US.

[2] Data compression in chrome beta for android. http://blog.chromium.org/2013/03/data-compression-in-chrome-beta-for.html.

[3] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting Mobile 3G Using WiFi. In *Proc. of the international conference on Mobile systems, applications, and services (Mobisys), 2010*.

[4] National Broadband Map. http://www.broadbandmap.gov/.

[5] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. Proportional Rate Reduction for TCP. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[6] Dummynet. http://info.iet.unipi.it/~luigi/dummynet/.

[7] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDYier Mobile Web? In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2013*.

[8] 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 *Proc. of the ACM Sigcomm, 2013*.

[9] T. J. Hacker, B. D. Noble, and B. D. Athey. The Effects of Systemic Packet Loss on Aggregate TCP Flows . In *Proc. of IEEE Iternational Parallel and Distributed Processing Symposium (IPDPS), 2002*.

[10] HTTP/2.0 Draft Specifications. https://github.com/http2/http2-spec.

[11] mod_spdy. https://code.google.com/p/mod-spdy/.

[12] World Flags mod_spdy Demo. https://www.modspdy.com/world-flags/.

[13] Not as SPDY as you thought. http://www.guypo.com/technical/not-as-spdy-as-you-thought/.

[14] J. Padhye and H. F. Nielsen. A comparison of SPDY and HTTP performance. In *MSR-TR-2012-102*.

[15] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2011*.

[16] Amazon silk browser. http://amazonsilk.wordpress.com/.

[17] Chapter 11. HTTP 1.X. http://chimera.labs.oreilly.com/books/1230000000545/ch11.html.

[18] SPDY best practices. http://dev.chromium.org/spdy/spdy-best-practices.

[19] Analysis of SPDY and TCP Initcwnd. http://tools.ietf.org/html/draft-white-httpbis-spdy-analysis-00.

[20] SPDY whitepaper. http://www.chromium.org/spdy/spdy-whitepaper.

[21] SPDY protocol–Draft 3. http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3.

[22] Spdylay - SPDY C Library. https://github.com/tatsuhiro-t/spdylay.

[23] Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses. http://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[24] W3C DOM Level 3 Events Specification. http://www.w3.org/TR/DOM-Level-3-Events/.

[25] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2013*.

[26] X. S. Wang, H. Shen, and D. Wetherall. Accelerating the Mobile Web with Selective Offloading. In *Proc. of the ACM Sigcomm Workshop on Mobile Cloud Computing (MCC), 2013*.

FaRM: Fast Remote Memory

Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, Miguel Castro

Microsoft Research

Abstract

We describe the design and implementation of FaRM, a new main memory distributed computing platform that exploits RDMA to improve both latency and throughput by an order of magnitude relative to state of the art main memory systems that use TCP/IP. FaRM exposes the memory of machines in the cluster as a shared address space. Applications can use transactions to allocate, read, write, and free objects in the address space with location transparency. We expect this simple programming model to be sufficient for most application code. FaRM provides two mechanisms to improve performance where required: lock-free reads over RDMA, and support for collocating objects and function shipping to enable the use of efficient single machine transactions. FaRM uses RDMA both to directly access data in the shared address space and for fast messaging and is carefully tuned for the best RDMA performance. We used FaRM to build a key-value store and a graph store similar to Facebook's. They both perform well, for example, a 20-machine cluster can perform 167 million key-value lookups per second with a latency of 31μs.

1 Introduction

Decreasing DRAM prices have made it cost effective to build commodity servers with hundreds of gigabytes of DRAM. A cluster with one hundred machines can hold tens of terabytes of main memory, which is sufficient to store all the data for many applications or at least to cache the applications' working sets [11, 38, 39]. This has the potential to enable applications that perform small random data accesses, because it removes the overhead of disk or flash, but network communication remains a bottleneck. Emerging fast networks are not going to solve this problem while systems continue to use traditional TCP/IP networking. For example, the results in [16] show a state-of-the-art key-value store performing 7x worse in a client-server setup using TCP/IP than in a single-machine setup despite extensive request batching.

RDMA provides reliable user-level reads and writes of remote memory. It achieves low latency and high throughput because it bypasses the kernel, avoids the overheads of complex protocol stacks, and performs remote memory accesses using only the remote NIC without involving the remote CPU. RDMA has long been supported by Infiniband but it has not seen widespread use in data centers because Infiniband has traditionally been expensive and it is not compatible with Ethernet. Today, RoCE [27] supports RDMA over Ethernet with data center bridging [25, 26] at competitive prices.

FaRM uses RDMA writes to implement a fast message passing primitive that achieves an order-of-magnitude improvement in message rate and latency relative to TCP/IP on the same Ethernet network (Figure 2). It also uses one-sided RDMA reads to achieve an additional two-fold improvement for read-only operations that dominate most workloads [9, 11]. We did not get this performance out of the box. We improved performance by up to a factor of eight with careful tuning and changes to the operating system and the NIC driver.

FaRM machines store data in main memory and also execute application threads. This enables locality-aware optimizations which are important because accessing local memory is still up to 23x faster than RDMA.

FaRM exposes the memory of all machines in the cluster as a shared address space. Threads can use ACID transactions with strict serializability to allocate, read, write, and free objects in the address space without worrying about the location of objects. FaRM provides an efficient implementation of this simple programming model that offers sufficient performance for most application code. Transactions use optimistic concurrency control with an optimized two-phase commit protocol that takes advantage of RDMA. FaRM achieves availability and durability using replicated logging [39] to SSDs, but it can also be deployed as a cache [11, 38].

FaRM offers two mechanisms to improve performance where required with only localized changes to the code: lock-free reads that can be performed with a single RDMA and are strictly serializable with transactions, and support for collocating objects and function shipping to allow applications to replace distributed transactions by optimized single machine transactions.

We designed and implemented a new hashtable algorithm on top of FaRM that combines hopscotch hashing [21] with chaining and associativity to achieve high space efficiency while requiring a small number of RDMA reads for lookups: small object reads are performed with only 1.04 RDMA reads at 90% occupancy. We optimize inserts, updates, and removes by taking advantage of FaRM's support for collocating related objects and shipping transactions.

We used YCSB [15] to evaluate the performance of FaRM's hashtable. We compared FaRM with a baseline system that uses TCP/IP for messaging and performs better than MemC3 [16] (which is the best main-memory key-value store in the literature). Our evaluation on a cluster of 20 servers connected by 40 Gbps Ethernet demonstrates good scalability and performance: FaRM provides an order-of-magnitude better throughput and latency than the baseline across a wide range of settings.

We also implemented a version of Facebook's Tao graph store [11] using FaRM. Once more FaRM achieves an order of magnitude better throughput and latency than the numbers reported in [11].

2 Background on RDMA

RDMA requests are sent over reliable connections (also called *queue pairs*) and network failures are exposed as a terminated connection. Requests are sent directly to the NIC without involving the kernel and are serviced by the remote NIC without interrupting the CPU. A *memory region* must be *registered* with the NIC before it can be made available for remote access. During registration the NIC driver pins the pages in physical memory, stores the virtual to physical page mappings in a page table in the NIC, and returns a *region capability* that the clients can use to access the region. When the NIC receives an RDMA request, it obtains the page table for the target region, maps the target offset and size into the corresponding physical pages, and uses DMA to access the memory. Many NICs (including the ones we are using) guarantee that RDMA writes (but not reads) are performed in increasing address order. DMA operations are cache coherent on our hardware platform.

NICs have limited memory for page tables and connection data. Therefore, many NICs (including ours) store this information in system memory and use NIC memory as a cache. Accessing information that is not cached requires issuing a DMA to fetch it from system memory across the PCI bus. This is a common limitation of offload technology and requires careful use of available memory to achieve good performance.

RDMA has long been supported by Infiniband networks which are widely used by the HPC community. There have been deployments with thousands of nodes and full bisection bandwidth (e.g., [45]). Today, Infiniband has become cost competitive with Ethernet [37], but Ethernet remains prevalent in data centers.

RoCE (RDMA over Converged Ethernet) hardware supports RDMA over Ethernet with data center bridging extensions, which are already available in many switches. These extensions add priority based flow control [26] and congestion notification [25]. They eliminate losses due to congestion and allow segregation of RDMA from other traffic. The hardware manages connection state and acknowledgments eliminating the need for a protocol stack like TCP to ensure reliable delivery.

RoCE is price competitive at the rack level: $19/Gbps for 40 Gbps RoCE compared to $60/Gbps for 10 Gbps Ethernet[1], but there are some concerns about the scalability of RoCE. We expect it to scale to hundreds of nodes and there is ongoing work to improve scalability to thousands of nodes. This paper presents results on a 20-machine cluster using 40 Gbps RoCE but we have also run FaRM on a 78-machine Infiniband cluster.

3 FaRM

This section describes the design and implementation of FaRM. It starts by discussing FaRM's communication primitives and how their implementation is optimized for RDMA. Then it describes how FaRM implements a shared address space and how it ensures consistent accesses to the address space with good performance.

3.1 Communication primitives

FaRM uses one-sided RDMA reads to access data directly and it uses RDMA writes to implement a fast message passing primitive. This primitive uses a circular buffer, as in Figure 1, to implement a unidirectional channel. The buffer is stored on the receiver, and there is one buffer for each sender/receiver pair. The unused portions of the buffer (marked as "Processed" and "Free") are kept zeroed to allow the receiver to detect new messages. The receiver periodically polls the word at the "Head" position to detect new messages. Any non-zero value L in the head indicates a new message, of length L. The receiver then polls the message trailer; when it

[1] Prices include NICs and switches as of October 2013. Ethernet prices are for Intel X520-T2 NICs and Juniper EX4550 switches. RoCE prices are for Mellanox ConnectX 3 NICs and SX1036 switches.

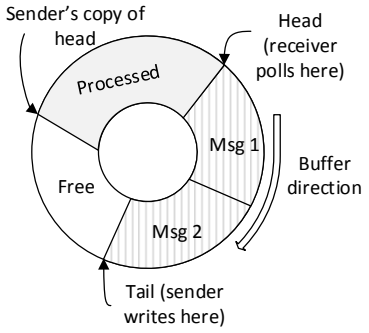


Figure 1: Circular buffer for RDMA messaging

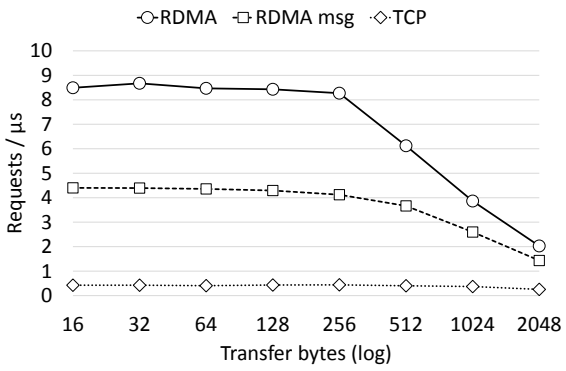


Figure 2: Random reads: request rate per machine

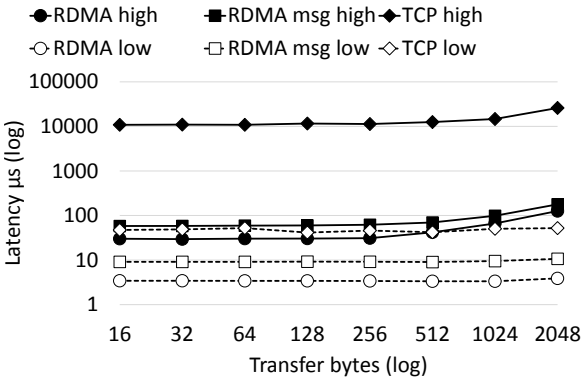


Figure 3: Random reads: latency with high and low load

becomes non-zero, the entire message has been received because RDMA writes are performed in increasing address order. The message is delivered to the application layer, and once it has been processed the receiver zeroes the message buffer and advances the head pointer.

The sender uses RDMA to write messages to the buffer tail and it advances the tail pointer on every send. It maintains a local copy of the receiver's head pointer and never writes messages beyond that point. The receiver makes processed space available to the sender lazily by writing the current value of the head to the sender's copy using RDMA. To reduce overhead, the receiver only updates the sender's copy after processing at least half of the buffer. The sender's copy of the head always lags the receiver's head pointer and thus the sender is guaranteed never to overwrite unprocessed messages.

Polling overhead increases linearly with the number of channels, so we establish a single channel from a thread to a remote machine. We observed negligible polling overhead with 78 machines. We also found that, at this scale, RDMA writes and polling significantly outperform the more complex Infiniband send and receive verbs. In large clusters, it may be better to use RDMA write with immediate and a shared receive queue [35], which would make polling overhead constant.

FaRM messaging is similar to the one described in [35] but our implementation uses a contiguous ring buffer as opposed to a ring of buffers to provide better memory utilization with variable-sized messages. Additionally, the receiver in [35] piggybacks updates to the sender's head pointer in messages.

We ran a micro-benchmark to compare the performance of FaRM's communication primitives with TCP/IP on a cluster with 20 machines connected by a 40 Gbps RoCE network (more details in Section 4). Each machine ran a number of threads that issued requests to read a random block of memory from a random remote machine in an all-to-all communication pattern. Figure 2 shows the average request rate per machine in a configuration optimized for peak throughput. Towards the left of the graph, FaRM's communication primitives are bottlenecked on packet rate and, towards the right, on bit rate. One-sided RDMA reads achieve a bit rate of nearly 33 Gbps with 2 KB request sizes and the bit rate saturates around 35 Gbps for request sizes greater than 8 KB.

FaRM's RDMA-based messaging achieves a request rate between 11x and 9x higher than TCP/IP for request sizes between 16 and 512 bytes, which are typical of data center applications (e.g., [9]). One-sided RDMA reads achieve an additional 2x improvement for sizes up to 256 bytes because they require half the network packets. We expect this performance gap to increase with the next generation of NICs that support 4x the message rate [36]; one-sided RDMA reads do not involve the remote CPU and RDMA-based messaging will be CPU bound.

We also measured UDP throughput and observed it is less than half the throughput of TCP configured for maximum throughput (with Nagle on). So we decided to compare against TCP in the rest of the paper.

Figure 3 shows average request latency both at peak request rate and using only 2 machines configured for minimum latency. The latency of TCP/IP at peak request rate is at least 145x higher than that of RDMA-based messaging across all request sizes. Using one-sided RDMA reads reduces latency by an extra factor of two for sizes up to 256 bytes. In an unloaded system, the latency of RDMA reads is at least 12x lower than TCP/IP

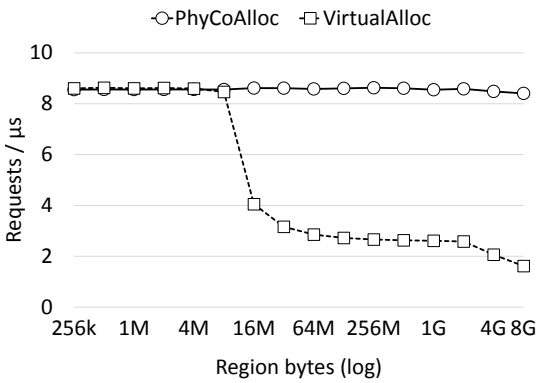


Figure 4: Impact of physically contiguous regions

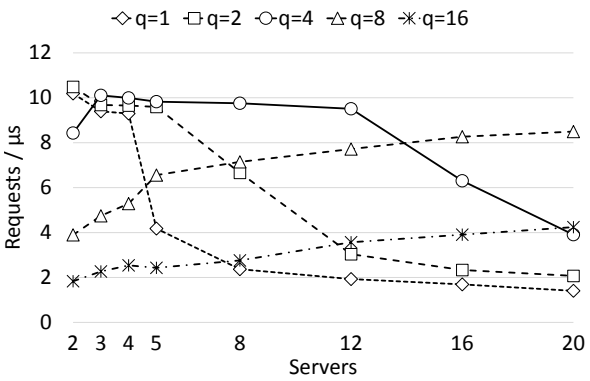


Figure 5: Impact of connection multiplexing

and 3x lower than RDMA-based messaging across all request sizes. The micro-benchmark shows that FaRM's communication primitives can achieve both low latency and high message rates at the same time.

Achieving this level of performance for the two communication primitives was non-trivial and it required solving several problems. The first problem we observed was that the performance of RDMA operations decreased significantly as we increased the amount of memory registered for remote access. The reason was that the NIC was running out of space to cache all the page tables. So it kept fetching page table entries from system memory across the PCI bus.

We fixed this problem by using larger pages to reduce the number of entries in NIC page tables. Unfortunately, existing large page support in Windows and Linux was not sufficient to eliminate all the fetches because of the large amount of memory registered by FaRM. So we implemented PhyCo, a kernel driver that allocates a large number of physically-contiguous and naturally-aligned 2 GB memory regions at boot time (2 GB is the maximum page size supported by our NICs). PhyCo maps the regions into the virtual address space of the FaRM process aligned on a 2 GB boundary. This allowed us to modify the NIC driver to use 2 GB pages, which reduced the number of page table entries per region from more than half a million to one.

We ran the random read benchmark to compare the request rate of 64-byte RDMA reads when regions are allocated with VirtualAlloc and with PhyCo. Figure 4 shows that with VirtualAlloc the request rate drops by a factor of 4 when more than 16 MBs of memory is registered with the NIC. With PhyCo, the request rate remains constant even when registering 100 GB of memory.

We also observed a significant decrease in request rate when the cluster size increased because the NIC ran out of space to cache queue pair data. Using a queue pair between every pair of threads requires $2 \times m \times t^2$ queue pairs per machine (where m is the number of machines and t is the number of threads per machine). We reduced this to $2 \times m \times t$ using a single connection between a thread and each remote machine. In addition, we introduced queue pair sharing among q threads in a NUMA-aware way, resulting in a total of $2 \times m \times t/q$ queue pairs per machine. This trades off parallelism for a reduction in the amount of queue pair data on the NIC.

We ran the random read benchmark with 64-byte transfers while varying the cluster size and the value of q. Figure 5 shows that the optimal value of q depends on the size of the cluster. Small values provide more parallelism and lower sharing overhead, which results in better performance in small clusters, but they also require more queue pair data, which results in degraded performance with larger clusters. In the remainder of the paper, we use these results to select the best value of q for each cluster size. We expect to solve this problem in the future by using Dynamically Connected Transport [36], which improves scalability by setting up connections on demand.

Early experiments showed that using interrupts and blocking could increase RDMA latency by a factor of four. Therefore, we use an event-based programming model. Each FaRM machine runs a user-level process and pins threads to hardware threads. Threads run an event loop which executes application work items and polls for the arrival of RDMA-based messages and the completion of RDMA requests. This polling is done at the user level without involving the OS.

3.2 Architecture and programming model

FaRM's architecture is motivated by the performance results presented in the previous section. FaRM's communication primitives are fast but accesses to main memory still achieve up to 23x higher request rate. Therefore, we designed FaRM to enable applications to improve performance by collocating data and computation on the same machine. FaRM machines store data in main memory and they also execute application threads. The memory of all machines in the cluster is exposed as a shared address space that can be read using one-sided RDMA.

```
Tx*  txCreate();
void txAlloc(Tx *t, int size, Addr a, Cont *c);
void txFree(Tx *t, Addr a, Cont *c);
void txRead(Tx *t, Addr a, int size, Cont *c);
void txWrite(Tx *t, ObjBuf *old, ObjBuf *new);
void txCommit(Tx *t, Cont *c);

Lf* lockFreeStart();
void lockFreeRead(Lf* op,Addr a,int size,Cont *c);
void lockFreeEnd(Lf *op);
Incarnation objGetIncarnation(ObjBuf *o);
void objIncrementIncarnation(ObjBuf *o);

void msgRegisterHandler(MsgId i, Cont *c);
void msgSend(Addr a, MsgId i, Msg *m, Cont *c);
```

Figure 6: FaRM's API

Currently, we support a single FaRM protection domain across the cluster.

Figure 6 shows the main operations in FaRM's interface. FaRM provides an event-based programming model. Operations that require polling to complete take a *continuation* argument, which consists of a *continuation function* and a *context* pointer. The continuation function is invoked when the operation completes and it is passed the result of the operation and the context pointer. The continuation is always invoked on the thread that initiated the operation.

FaRM provides strictly serializable ACID transactions as a general mechanism to ensure consistency. Applications start a transaction by creating a transaction context. They can allocate and free objects using `txAlloc` and `txFree` inside transactions. Allocations return opaque 64-bit pointers that can be used to access objects or stored in object fields to build pointer linked data structures. Applications can request that the new object is allocated close to an existing object by supplying the existing object's address to `txAlloc`. FaRM attempts to store the two objects in the same machine and keep them on the same machine even after recovering from failures or adding new machines. This allows applications to collocate data that is commonly accessed together.

The `txRead` operation can be used to read an object given its address and size. It allocates an object buffer and uses RDMA to read the object's data and meta-data into the buffer. When it completes, it passes the object buffer to the continuation. To update an object, a transaction must first read the object and then call `txWrite` to create a writable copy of the object buffer. Applications commit transactions by calling `txCommit`, which returns the outcome and frees any allocated buffers. Transactions can abort due to conflicts or failures; otherwise, the writes are committed.

General distributed transactions provide a simple programming model but can be too expensive to implement performance critical operations. FaRM's API allows applications to implement efficient lock-free read-only operations that are serializable with transactions. `lockFreeStart` and `lockFreeEnd` are used to bracket lock-free operations. `lockFreeRead` is similar to `txRead` but any object buffers it allocates are freed by `lockFreeEnd`. FaRM also exposes object incarnations, which can be used to combine several lock-free reads into more complex operations. Transactions and lock-free operations are described in Sections 3.4 and 3.5.

The last two API operations are used to send RDMA-based messages to a thread in the machine that stores an object, which allows shipping transactions to the server that stores the object. Together with the ability to collocate related data on the same machine, this enables replacing distributed transactions by single machine transactions, which are significantly less expensive.

FaRM also offers functions to allocate, read, and free arrays of objects. This allows efficient reads of consecutive elements in an array with a single RDMA.

FaRM uses replicated logging to provide ACID transactions with strict serializability and high availability under the following assumptions: crash failures, a bound on the maximum number of failures per replica group, a bound on clock drift in an otherwise asynchronous system for safety, and eventual synchrony for liveness. We do not describe or evaluate recovery from failures in this paper but the common-case (non failure) performance reported in this paper includes all the overheads of replication and logging.

We used this interface to implement a distributed hashtable (Section 3.6) and a graph store similar to Facebook's Tao [11] (Section 4.4).

3.3 Distributed memory management

FaRM's shared address space consists of many 2 GB *shared memory regions* that are the unit of address mapping, the unit of recovery, and the unit of registration for RDMA with the NIC. The address of an object in the shared address space consists of the 32-bit region identifier and the 32-bit offset relative to the start of the region. To access an object, FaRM uses a form of consistent hashing [31] to map the region identifier to the machine that stores the object. If the region is stored locally, FaRM obtains the base address for the region and uses local memory accesses. Otherwise, FaRM contacts the remote machine to obtain a capability for the region, and then uses the capability, the offset in the address and the object size to build an RDMA request. Capabilities for remote regions are cached to improve performance.

Consistent hashing is implemented using a one-hop distributed hashtable [6]. Each machine is mapped into k virtual rings by hashing its IP address with k hash functions. FaRM uses multiple rings to allow multiple regions to be recovered in parallel as in RAMCloud [39] and also to improve load balancing [44]. We currently

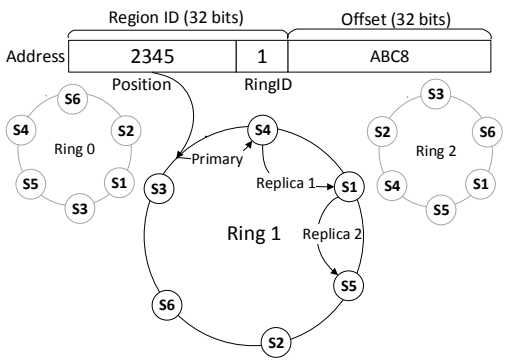


Figure 7: Resolving an address

use $k = 100$. The 32-bit shared region identifier identifies both a ring and a position in the ring. The primary copy and replicas of the region are then stored at the r machines immediately following the region's position in the ring. Figure 7 shows a simple example with $k = 3$ and $r = 3$. The mapping of the region to the machine can be performed locally given the set of machines in the cluster. Cluster membership can be maintained reliably using Zookeeper [24].

Memory allocators are organized into a three-level hierarchy — slabs, blocks, and regions — to reduce synchronization overheads (as in parallel allocators [10]). At the lowest level, threads have private *slab allocators* that allocate small objects from large blocks. Each block is used to allocate objects of the same size. FaRM supports 256 distinct sizes from 64 bytes to 1 MB. The sizes are selected so the average fragmentation is 1.8% and the maximum is 3.6%. An object is allocated in the smallest size class that can fit it. Slab allocators use a single bit in the header of each object to mark it allocated. This state is replicated when a transaction that allocates or frees objects commits and it is scanned during recovery to reconstruct allocator data structures.

The blocks are obtained from a machine-wide *block allocator* that allocates blocks from shared memory regions. It splits the regions into blocks whose size is a multiple of 1 MB. Each region has a table with an 8-byte allocation state per block. The regions are obtained from a cluster-wide *region allocator*. The region allocator uses PhyCo to allocate memory for the region and then it registers the region with the NIC to allow remote access (as described in Section 2). It picks an identifier for the region by selecting a ring at random and a position in the ring that ensures the local node stores the primary copy. Information about region and block allocations is replicated at allocation time.

FaRM allows applications to supply a location hint, which is the address of an existing object, when allocating an object. FaRM attempts to allocate the object in the following order: in the same block as the hint, in the same region, or in a region with a nearby position in the same virtual ring. This ensures that the allocated object and the hint remain collocated both on the primary and on the replicas with high probability even after failures and reconfigurations. If the hint is an address stored at another server, the allocation is performed using an RPC to the remote server.

3.4 Transactions

FaRM supports distributed transactions as a general mechanism to ensure consistency. Our implementation uses optimistic concurrency control [32] and two-phase commit [18] to ensure strict serializability [41]. A transaction context records the version numbers of objects read by the transaction (the *read set*), the version numbers of objects written by the transaction (the *write set*), and it buffers writes. At commit time, the machine running the transaction acts as the *coordinator*. It starts by sending *prepare* messages to all *participants*, which are the primaries and replicas of objects in the write set. The primaries lock the modified objects and both primaries and replicas log the message before sending replies back. After receiving replies from all participants, the coordinator sends *validate* messages to the primaries of objects in the read set to check if the versions read by the transaction are up to date. If read set validation succeeds, the coordinator sends commit messages first to the participant replicas and then to the participant primaries. The primaries update the modified objects and unlock them, and both primaries and replicas log the commit message. The transaction aborts if any modified object is locked, if read set validation fails, or if the coordinator fails to receive replies for all the prepare and validate messages.

FaRM replicas keep the log on SSDs. To improve logging performance, they use a few megabytes of non-volatile RAM [2] to hold the circular message buffers and to buffer log entries [39]. The entries are flushed when the buffers fill up and log cleaning is invoked when the log is half full. These logs are used to implement a parallel recovery mechanism similar to RAMCloud [39].

The two-phase commit protocol is implemented using RDMA-based messaging, which was shown to have very low latency. This reduces conflicts and improves performance by reducing the amount of time locks are held. Despite these optimizations, two-phase commit may be too expensive to implement common case operations.

FaRM provides two mechanisms to achieve good performance in the common case: single machine transactions and lock-free read-only operations. Applications can use single machine transactions by collocating the objects accessed by a transaction on the same primary and on the same replicas, and by shipping the transaction

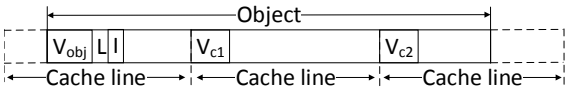


Figure 8: Versioning for lock-free reads. A read is consistent if the lock field L is zero and V_{c1} and V_{c2} match the low-order bits of V_{obj}. The incarnation I is used to detect if the object was freed concurrently with being read.

to the primary. In this case, write set locking and read set validation are local. Therefore, the prepare and validate messages are not needed and the primary only needs to send a commit message with the buffered writes to the replicas before unlocking the modified objects. Additionally, we use two locking modes: objects are first locked in a mode that allows lock-free reads and the primary locks objects in exclusive mode just before updating them (after the commit messages are delivered to the replicas). Single machine transactions improve performance by reducing the number of messages and by further reducing delays due to locks.

3.5 Lock-free operations

FaRM provides lock-free reads that are serializable with transactions and are performed using a single RDMA read without involving the remote CPU. The application is guaranteed to observe a consistent object state even if it is concurrent with writes to the same object. FaRM relies on cache coherent DMA: it stores an object's version number both in the first word of the object header and at the start of each cache line (except the first). These versions are not visible to the application; FaRM automatically converts the object layout on reads and writes.

A `lockFreeRead` reads the object with RDMA and checks if the header version is unlocked and matches all the cache line versions. If the check succeeds, the read is strictly serializable with transactions. Otherwise, the RDMA is retried after a randomized backoff. Figure 8 shows the version fields for an object that spans three cache lines.

An object is written during transaction commit using local memory accesses. The header version is locked with a compare-and-swap during the prepare phase. We use the two least significant bits in the header version to encode the lock mode. During the commit phase, an object is updated by first writing a special lock value to the cache line versions, then updating the data in each cache line, and finally updating the cache line versions and the header version. These steps are separated by memory barriers. On x86 processors, compiler barriers are sufficient to ensure the required ordering. Since DMA is cache-coherent on x86, any RDMA read observes memory writes within each cache line in the order enforced by the memory barriers. Therefore, matching versions across all cache lines spanned by an object ensures strict serializability for lock-free reads.

We use 64-bit header versions to prevent wrapping around but cache line versions only keep the least significant l bits of the version to save space. We can do this because there is a lower bound on the time it takes to perform a write and we abort RDMA reads that take longer than an upper bound to ensure that a read can never overlap two successive writes that produce versions with the same least significant l bits. This relies on a weak bound on clock drift that we already required to maintain leases with ZooKeeper. The results in this paper were obtained with $l = 16$, but our measurements show that $l = 8$ is sufficient in configurations with replication.

To provide consistency, FaRM must ensure that lock-free reads do not access objects that have been freed by concurrent transactions. FaRM uses type stability [19] to ensure that object meta-data remains valid and incarnation checks [46] to detect when objects are freed. Object headers have a 64-bit incarnation that is initially zero and is incremented when the object is freed. FaRM provides 128-bit *fat pointers* that include the object address, size, and expected incarnation. Applications check that the incarnation in the object buffer returned by a lock-free read matches the incarnation in the pointer, which guarantees that the object has not been freed.

FaRM can reuse freed memory to allocate another object of the same size because the incarnation in the object header remains valid. Reusing memory for different object sizes requires more work because the object header may be overwritten with arbitrary data. FaRM implements a distributed version of an epoch-based allocator [17] to do this. It sends an *end of epoch* request to the threads on all machines (we aggregate messages to/from the same machine). When a thread receives this request, it clears any cached pointers, starts a new epoch, and continues processing operations in the new epoch. Once all transactions and read-only operations that started in previous epochs complete, the thread sends a reply to the request. FaRM's API provides primitives that bracket operations to enable detecting when ongoing operations complete. Memory can be reused after receiving responses from all machines in the current configuration. This mechanism does not impact performance significantly because it runs in the background and only when available memory drops below a threshold.

3.6 Hashtable

FaRM also provides a general key-value store interface that is implemented as a hashtable on top of the shared address space. One important use of this interface is as a root to obtain pointers to shared objects given keys.

Designing a hashtable that performs well using RDMA is similar to other forms of memory hierarchy aware data structure design: it is important to balance achieving good space efficiency with minimizing the number and size of RDMAs required to perform common operations. Ideally, we would like to perform lookups, which are the most common operation, using a single RDMA read. We identified hopscotch hashing [21] as a promising approach to achieve this goal because it guarantees that a key-value pair is located in a small contiguous region of memory that may be read with a single RDMA. This contrasts with popular approaches based on cuckoo hashing [40] where a key-value pair is in one of several disjoint regions.

Each bucket in a hopscotch hashtable has a *neighbourhood* that includes the bucket and the $H-1$ buckets that follow. Hopscotch hashing maintains the invariant that a key-value pair is stored in the neighbourhood of the key's bucket (i.e., the bucket the key hashes to). To insert a key-value pair, the algorithm looks for an empty bucket close to the key's bucket by doing a linear probe forward. If the empty bucket is in the neighbourhood of the key's bucket, the key-value pair is stored there. Otherwise, the algorithm attempts to move the empty bucket towards the neighbourhood by repeatedly displacing key-value pairs while preserving the invariant. If the algorithm does not find an empty bucket or is unable to preserve the invariant, the hashtable is resized.

The original algorithm outperforms chaining and cuckoo hashtables at high occupancy using $H = 32$ [21] (where occupancy is the ratio between the number of key-value pairs inserted and the number of slots in the table). Unfortunately, large neighbourhoods perform poorly with RDMA because they result in large reads. For example, using $H = 32$ with 64-byte key-value pairs requires RDMA reads of at least 2 KB, which perform significantly worse than smaller RDMAs (Figure 2). Simply using small neighbourhoods does not work well as it requires frequent resizes and results in poor space efficiency. For example, the original algorithm achieves an average occupancy of only 37% with $H = 8$.

We designed a new algorithm, *chained associative hopscotch hashing*, that achieves a good balance between space efficiency and the size and number of RDMAs used to perform lookups by combining hopscotch hashing with chaining and associativity. For example, on average it requires only 1.04 RDMA reads per lookup with $H = 8$ at 90% occupancy. This is better than techniques based on cuckoo hashing [40] that require 3.2 RDMA reads at 75% occupancy (or 1.6 if key-value pairs were inlined in the table) [37].

The new algorithm uses an overflow chain per bucket. If an insert fails to move an empty bucket into the right neighbourhood, it adds the key-value pair to the overflow chain of the key's bucket instead of resizing the table. This also lets us limit the length of linear probing during inserts. The algorithm uses associativity to amortize the space overhead of chaining and of FaRM's object meta-data across several key-value pairs. Each bucket is a FaRM object with $H/2$ slots to store key-value pairs. The algorithm guarantees that a key-value pair is stored in the key's bucket or the next one. Overflow blocks also store several key-value pairs (currently two) to improve performance and space efficiency.

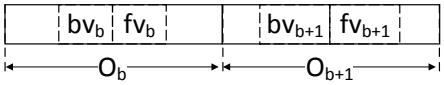


Figure 9: Joint versions for lock-free reads of adjacent buckets. The two objects are consistent with each other if they are individually consistent and $fv_b = bv_{b+1}$.

We implemented the new algorithm using FaRM's API. The hashtable is sharded across the machines in the cluster. Each machine allocates shards, which are FaRM arrays of buckets, and exchanges pointers to the shards with the other machines. We use consistent hashing to partition hash values across shards to enable elasticity.

Lookups are performed using lock-free read-only operations. A lookup for a key k starts by issuing a single RDMA to read both k's bucket b and the next bucket $b+1$. The lookup completes if it finds k in b or $b+1$. Otherwise, it uses lock-free reads to search for k in b's chain of overflow blocks. The chain uses fat pointers to link blocks and lookups check if the incarnation numbers in a fat pointer and the next block match. If they do not, the lookup is restarted. It is inefficient to store large key-value pairs inline in buckets because it results in large RDMAs. FaRM stores large or variable-sized key-value pairs as separate objects and it stores a key (or a hash for large keys) and a fat pointer to the object in the bucket. If the incarnation numbers in the fat pointer and the object do not match, the lookup is restarted.

The version checks in Section 3.5 guarantee that each bucket is individually consistent when read. For hash table lookups, however, we must also ensure that the two buckets in the neighbourhood are consistent with each other. To do this we add *joint versions* for adjacent pair of buckets, meaning that each object bucket stores a forward and a backward joint version (Figure 9). If the corresponding joint versions do not have the same value, the read is restarted. Transactions that update adjacent buckets increment the corresponding joint versions. We reduce the space overhead of joint versions using the same technique we used to reduce the size of cache line versions for lock-free reads. The results in this paper use 16-bit joint versions.

We optimize inserts, updates, and removes by ship-

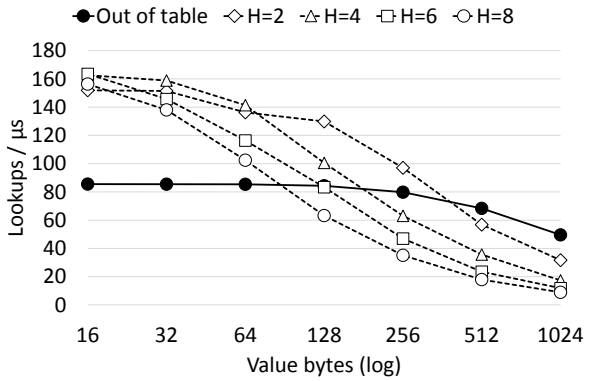


Figure 10: Hashtable: throughput with varying value size

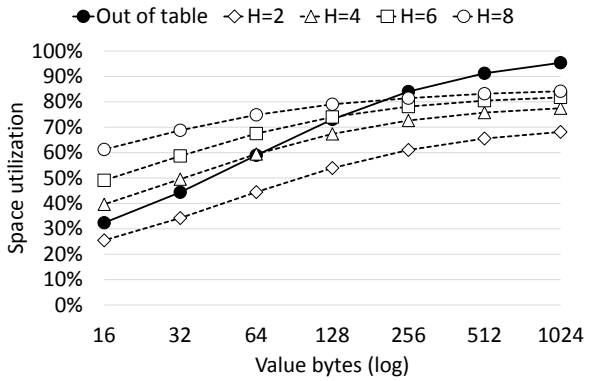


Figure 11: Hashtable: utilization with varying value size

ping transactions to the machine that store the relevant shard. Using transactions simplifies the implementation, which is significantly more complex than for lookups. We use FaRM's API to ensure that shards are collocated with their overflow blocks so we can use more efficient single machine transactions.

We implement inserts as described above. We use a technique inspired by flat combining [20] to combine concurrent inserts and updates to the same key into a single transaction. This improves throughput by more than 4x in our experiments with skewed YCSB workload by reducing the overheads of concurrency control and replication for hot keys. Removes attempt to collapse overflow chains to reduce the number of RDMAs for lookups. They always move the last key-value pair in the chain to the newly freed slot and free the last overflow block if it becomes empty. Otherwise, they increment its incarnation number to ensure lookups observe a consistent view.

FaRM's hashtable guarantees linearizability and it performs well. Figure 10 shows lookup throughput on a 20-machine cluster (see Section 4) at 90% occupancy with 8-byte keys and different value sizes when keys are chosen uniformly at random. It shows results when values are inlined with different neighbourhood sizes and when values are stored outside of the buckets (using $H = 8$). Figure 11 shows the space utilization in the same experiment: this is the ratio between the total number of bytes in key-value pairs and the total amount of memory used by the hashtable. The results show that inlining values with $H = 8$ or $H = 6$ provides a good balance between throughput and space utilization for objects up to 128 bytes. Applications that can tolerate low space utilization to achieve better throughput can inline objects up to 320 bytes with $H = 2$. Objects larger than 320 bytes should be stored outside the table.

4 Evaluation

We evaluate FaRM's performance and its ability to support applications with different data structures and access patterns. We first compare the performance of FaRM's key-value store with a state-of-the art implementation that uses TCP/IP. Then we evaluate FaRM's ability to serve a workload based on Facebook's Tao [8, 11].

4.1 Experimental setup

We ran the experiments on an isolated cluster with 20 machines. Each machine had a 40 Gbps Mellanox ConnectX-3 RoCE NIC connected to a single port of a Mellanox SX-1036 switch. The machines ran Windows Server 2012 R2 on two 2.4 GHz Intel Xeon E5-2665 CPUs with 8 cores and two hyper-threads per core (32 hardware threads per machine). Each machine had a 240 GB Intel 520 SSD for logging and 128 GBs of DRAM (2.5 TB of DRAM across the cluster). We configured machines with 28 GB of private memory and 100 GB of shared memory. We used the results in Figure 5 to select the best connection multiplexing factor q for the number of nodes in each experiment. We report the average of three runs for each experiment. The standard deviation was below 1% of the average except for a very small number of points where it was below 10%.

To evaluate the performance of FaRM's key-value store, we compare it to a distributed hashtable that uses the same chained associative hopscotch hashing algorithm and most of the optimizations in MemC3 [16]. This hashtable uses a neighbourhood size of 12 with six key-value pair slots per bucket. Each key-value pair slot has a 1-byte hash tag [16] and a pointer to the key and value (which are stored outside the table). Remote operations are implemented over TCP/IP running natively in the same RoCE network. We tuned TCP/IP carefully for maximum performance, e.g., we enabled header processing in the NIC and used 16 receiver side queues. We labeled this baseline TCP in the experiments.

The key-value store experiments ran with 16-byte keys and 32-byte values as in [16] to facilitate comparisons and because these are representative of real workloads [16, 38]. Our baseline achieves 40 million lookups

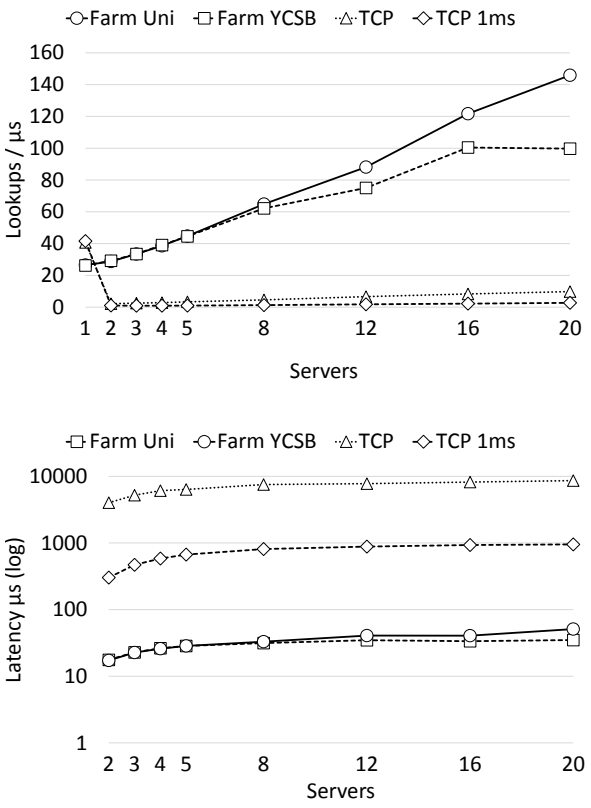


Figure 12: Key-value store: lookup scalability

per second on a single machine which is comparable with the 35 million reported by MemC3 [16] last year.

We loaded FaRM's key-value store and the baseline with 120 million key-value pairs per machine before taking measurements. We configured both stores for 90% occupancy and used a neighbourhood of 6 for FaRM because it provides good throughput and 62% space utilization (see Section 3.6). We used only 120 million key-value pairs to keep experiment running times low but we ran a control experiment with 20 machines and 1.3 billion key-value pairs per machine (58 GB of user data, 94 GB total per machine and 1.8 TB overall) and obtained the same lookup performance.

Except where noted, we measured performance for one minute after a 20 second warm-up. Keys were chosen randomly either with a uniform distribution or with the Zipf distribution prescribed by YCSB [15], which has $\theta = 0.99$. When using the Zipf distribution, the most popular key is accessed by 4% of the operations.

4.2 Key-value store lookups

Figure 12 shows throughput and latency for key-value store lookups as the number of machines increases. We show lines for FaRM with the uniform and YCSB distributions but only show baseline (TCP) performance with

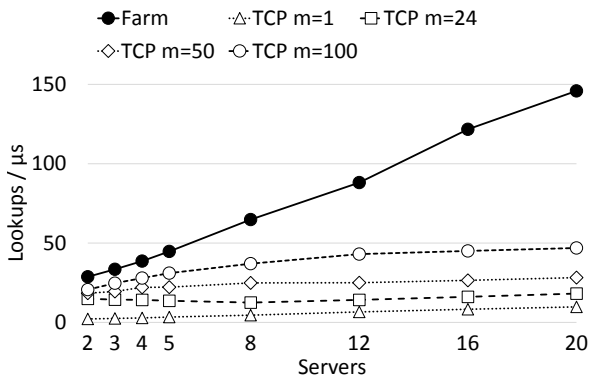


Figure 13: Key-value store: multi-get scalability

the uniform distribution because it is not impacted by the access distribution.

FaRM achieves 146 million lookups per second with a latency of only 35μs with 20 machines and the uniform distribution. This throughput is an order of magnitude higher than the baseline and the latency is two orders of magnitude lower. In fact, the latency of the baseline at peak throughput is more than 8ms, which is unacceptable for several applications (e.g., [38]). We decreased the number of concurrent requests to achieve a latency of 1ms for the baseline which degraded throughput by 3.6x as shown in Figure 12. Trading throughput for latency was not necessary for FaRM as it can achieve both high throughput and low latency at the same time.

The results also show that FaRM's performance scales well with the cluster size. The skew in the YCSB distribution impacts performance with more than 8 servers because it overloads the NICs on the machines that hold the hottest keys, but FaRM is able to achieve more than 100 million lookups per second with latency of 51μs.

Figure 12 also shows single-machine performance for both systems: the baseline can achieve 40 million lookups per second and FaRM achieves 26 million. This is because the baseline uses lock-free reads that are carefully tuned for this particular application whereas FaRM uses general support for lock-free reads, which involves an extra object copy even for local objects. But FaRM achieves 146 million lookups per second while providing 20 times more memory with 20 machines.

In the experiments so far, each lookup retrieves a single item. Main memory key-value stores like Memcached [1] provide a multi-get interface that allows applications to look up several keys. For example, Facebook reports that their applications issue multi-gets for 24 keys on average [38] to amortize communication overheads. We implemented a multi-get interface in our baseline key-value store but not in FaRM because current NICs do not support batching of RDMA. We could implement multi-get batching using FaRM's RDMA-based messaging but we have not yet done this.

Figure 13 compares FaRM's lookup throughput with the baseline key-value store (TCP) using different multi-get batch sizes. Multi-gets can improve the performance of the baseline significantly but FaRM still achieves 8x better throughput relative to multi-gets of 24 keys and 3x better relative to multi-gets of 100 (which are larger than the 95th percentile reported in [38]). However, multi-gets increase the latency of the baseline, which was already high. With multi-gets of 100 and 20 machines, lookup latency increases to 25ms. If we reduce the number of concurrent requests to achieve 10x lower latency, the throughput drops to 10x lower than FaRM's. Another limitation is that for a fixed multi-get size (which is determined by available client parallelism), the benefits of batching decrease with increasing cluster size because the batch must be broken into a message for each machine. For example, multi-gets of 24 improve throughput by 5.7x with 2 servers but only by 2x with 20.

The different mechanisms we discussed in the paper all contribute to the good performance we observed. The low level optimizations discussed in Section 2 improve lookup throughput by 8x. Using lock-free one-sided reads instead of RDMA-based messaging doubles throughput, and our hashtable design reduces the number of RDMAs per lookup by a factor of three when compared to the RDMA-aware design in Pilaf [37].

4.3 Key-value store updates

We also ran experiments with a mix of lookups and updates. We show results without replication (*NoRep*), with logging to SSDs in two replicas (*SSD*), and with logging to memory in two replicas (*Mem*). The first configuration corresponds to using FaRM as a cache and the last allows us to evaluate the overhead of the SSDs. We only show baseline (TCP) results without replication and with the uniform distribution. We set the log size to 32 GB and ran the experiment for 5 minutes with a longer warm-up period of 1 minute to allow the logger to reach a steady state mix of foreground logging and cleaning.

Figure 14 shows scalability with 5% updates (which corresponds to YCSB-B). This update rate is higher than those reported recently for main-memory systems [9, 11, 37, 38]. FaRM scales and performs well: it has an order of magnitude higher throughput than the non-replicated baseline key-value store even when logging to SSDs in two replicas. The throughput when logging to SSDs is 30% lower than without replication mostly due to the additional replication messages. We use a small amount of non-volatile RAM to remove the SSD latency from the critical path and the SSDs have enough bandwidth to cope with logging and cleaning with 5% updates.

The skewed access distribution in YCSB affects scalability as it did with read-only workloads. We omit the

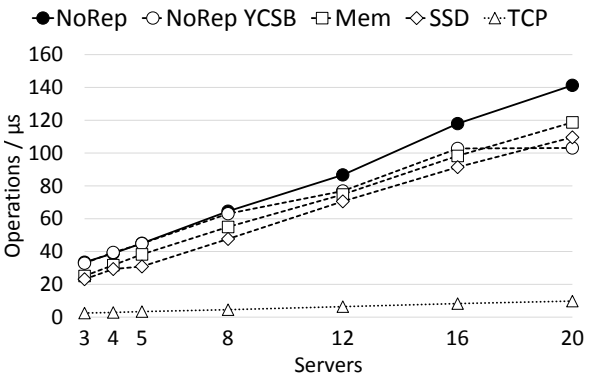
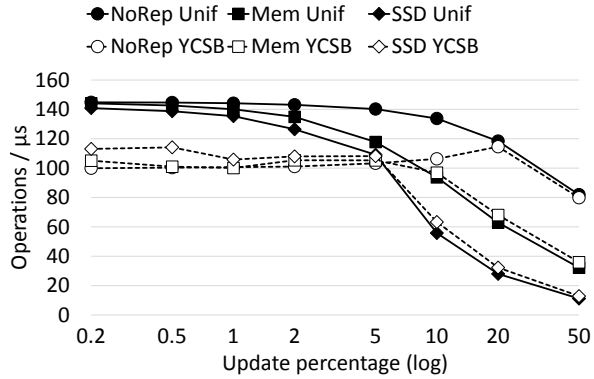
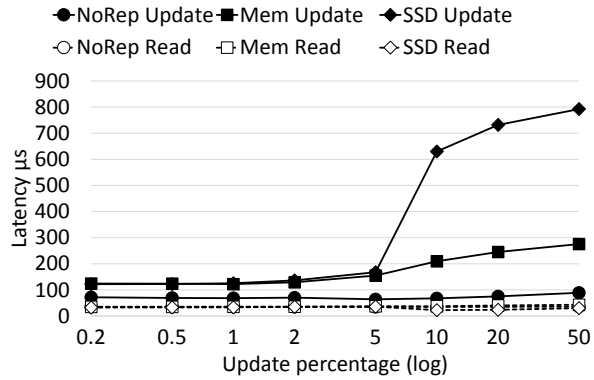


Figure 14: Key-value store: scalability with updates



(a) Throughput (YCSB and uniform)



(b) Latency for lookups and updates (uniform)

Figure 15: Key-value store: varying update rates

lines for the replicated configurations with YCSB because they are similar to the non-replicated configuration. Our measurements show that the update combining optimization in Section 3.6 improves throughput with skew by roughly 4x. Similarly, the dual-mode locking described in Section 3.4 improves the overall throughput by around 25% because it reduces the restart rate of lock-free reads. Both these optimizations have an even larger impact with higher update rates.

We evaluate the performance of FaRM with higher up-

date rates in Figure 15. The overhead of replication is visible with more than 2% updates and SSDs become the bottleneck with more than 5% because we saturate the available I/O bandwidth. With 5% updates, there are 215 MB/s of log writes and 215 MB/s of log reads for cleaning. Roughly half of the writes log new updates and the other half are cleaning writes. The performance with YCSB is slightly better than with the uniform distribution with high update rates because the update combining optimization is able to combine more operations.

Figure 15(b) shows that FaRM's lookup latency is independent of the update rate. Since lookups are implemented using RDMA reads, they are not impacted (as updates are) by increased CPU utilization and queuing delays in FaRM's messaging layer. With low update rates, update operations have a latency of 60μs without replication and 120μs with replication. The latency increases with the update rate because of longer queuing delays.

4.4 Tao

Facebook's Tao [11] is an in-memory graph store. It stores both nodes (e.g., users, comments) and edges (e.g., friend, author of). Both nodes and edges have types and application-specific data. Tao's workload is read dominated (99.8%) with four main operation types. Clients can read a node and its data (`obj_get`), read the most recent outbound edges of a given type from a given node (`assoc_range`), count the number of outbound edges of a given type from a given node (`assoc_count`), or find all outbound edges of a given type from a given node to a set of other nodes (`assoc_get`).

We have implemented a version of Tao. Nodes are FaRM objects with application data inlined and they are uniquely named using fat pointers. Edges are stored as a linked list per edge type in reverse timestamp order and they are collocated with the source node. Each linked list node stores multiple edges. The head pointers and counts of the linked lists are stored in the source node.

`obj_get` and `assoc_count` are implemented with a lock-free read of the node object. `assoc_range` is implemented with a lock-free read of the linked list head using a fat pointer cached by the client. When a new head is inserted, the old head's incarnation is incremented to ensure that clients can detect this and re-fetch the head pointer. These three operations account for 85% of the Tao workload and they require a single RDMA read in the common case. `assoc_get` requires a scan of the edge list. So it is implemented by function-shipping the operation to the machine storing the source node (and the edge list). Update operations use distributed transactions but they account for only 0.2% of the workload.

We evaluated the graph store using Facebook's LinkBench [8] with its default parameters for degree and data size distributions. We used the recommended "full" scale for LinkBench: a graph with 1 billion nodes for which LinkBench generated 4.35 billion edges. The workload was parametrized using the operation mix in [11]. We measured a throughput of 126 million operations per second on our 20-machine cluster. FaRM's per-machine throughput of 6.3 million operations per second is 10x that reported for Tao. FaRM's average latency at peak throughput was 41μs which is 40–50x lower than reported Tao latencies. The three operation types that use lock-free reads required only 1.02 RDMA reads per operation on average. Note that our results were obtained on hardware different from Facebook's and using LinkBench rather than the real workload. Nevertheless, the order-of-magnitude improvements in throughput and latency show that FaRM can implement graph stores efficiently when the graph fits in the cluster's memory.

5 Related work

RDMA has been primarily used to improve message passing performance, e.g., several MPI implementations [34, 35, 42] use RDMA. FaRM's RDMA-based messaging improves on the implementation in [35].

Several libraries (e.g., [22, 49]) and programming languages (e.g., [4, 13, 14, 47, 50]) provide a Partitioned Global Address Space (PGAS) abstraction where processes have both private memory and memory that can be accessed remotely using one-sided operations. Some of them use RDMA to implement one-sided operations but unlike FaRM they do not support efficient lock-free RDMA reads. Instead, they ensure consistency using locks, barriers or messages. Additionally, they were designed for batch computation and are not well suited to building interactive online services, e.g., they lack support for persistence and either provide no fault tolerance or create periodic checkpoints. Distributed shared memory systems (e.g., [5, 12, 43]) are similar to PGAS but lack support for user controlled data placement. FaRM provides a PGAS with ACID transactions.

Several projects have used Infiniband messaging primitives and RDMA to improve the performance of distributed file systems [28, 33, 48], HBase [23], and Memcached [7, 29, 30, 37]. These projects use RDMA to improve performance of a specific service whereas FaRM provides a general distributed computing platform. Additionally, they use RDMA to optimize message passing and do not support one-sided RDMA reads with the exception of [7, 37]. The work in [7] supports one-sided RDMA reads but provides no consistency guarantees.

Pilaf [37] implements a key-value store that uses send/receive verbs to ship update operations to the server and one-sided RDMA reads to implement lookups. It provides linearizability using 64-bit CRCs to detect in-

consistent reads. FaRM's technique to detect inconsistent reads is more general. It provides serializability with respect to general transactions. Additionally, FaRM's RDMA-aware hashtable design performs better because it requires fewer RDMAs to perform lookups with higher space utilization. It is hard to perform a direct performance comparison because the evaluation in [37] uses a single core in a single server and does not address the scalability problems that we discuss in Section 2.

RAMCloud [39] describes techniques for logging and recovery in a main memory key-value store but provides little information about normal case operation. We use similar techniques for logging and recovery but extend them to deal with transactions on general data structures in a shared address space. Unlike [39], we focus on techniques to achieve good performance in the normal case.

Like FaRM, Sinfonia [3] offers a shared address space with transactions. It introduces "mini-transactions" that improve performance by piggybacking execution onto the 2-phase commit protocol. FaRM offers general distributed transactions optimized to take advantage of RDMA together with lock-free reads that require a single RDMA and locality optimizations that enable single machine transactions.

6 Conclusion

We described the design and implementation of FaRM, a new distributed computing platform that stores application data in main memory and exploits RDMA communication to achieve high throughput and low latency at the same time. FaRM provides a shared address space and general distributed transactions to simplify programming. Since distributed transactions can be too expensive for performance critical operations, FaRM also provides two mechanisms to improve performance where needed: lock-free read-only operations and locality optimizations that enable single machine transactions. We demonstrated the effectiveness of these techniques by building RDMA-aware key value and graph stores. Our results show that FaRM performs well: it consistently achieves an order of magnitude better throughput and latency than main memory systems that use TCP/IP on the same physical network.

References

[1] Memcached. `http://memcached.org`.

[2] Viking Technology. `http://www.vikingtechnology.com/`.

[3] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP '07.

[4] ALLEN, E., CHASE, D., HALLETT, J., LUCHANGCO, V., MAESSEN, J.-W., RYU, S., STEELE JR, G. L., TOBIN-HOCHSTADT, S., DIAS, J., EASTLUND, C., ET AL. The Fortress language specification. *Sun Microsystems 139* (2005), 140.

[5] AMZA, C., COX, A. L., DWARKADAS, S., KELEHER, P. J., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer 29*, 2 (1996), 18–28.

[6] ANJALI, G., BARBARA, L., AND RODRIGO, R. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation* (2004), NSDI '04.

[7] APPAVOO, J., WATERLAND, A., DA SILVA, D., UHLIG, V., ROSENBURG, B., VAN HENSBERGEN, E., STOESS, J., WISNIEWSKI, R., AND STEINBERG, U. Providing a cloud network infrastructure on a supercomputer. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (2010), HPDC '10.

[8] ARMSTRONG, T. G., PONNEKANTI, V., BORTHAKUR, D., AND CALLAGHAN, M. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13.

[9] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), SIGMETRICS '12.

[10] BERGER, E. D., MCKINLEY, K. S., BLUMOFE, R. D., AND WILSON, P. R. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (2000), ASPLOS-IX.

[11] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), USENIX ATC'13.

[12] CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. Implementation and performance of Munin. In *ACM SIGOPS Operating Systems Review* (1991), vol. 25, ACM.

[13] CHAMBERLAIN, B. L., CALLAHAN, D., AND ZIMA, H. P. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications 21*, 3 (2007), 291–312.

[14] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (2005), OOPSLA '05.

[15] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing* (2010), SoCC '10.

[16] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (2013), NSDI'13.

[17] FRASER, K. *Practical Lock Freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[18] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. 1992.

[19] GREENWALD, M., AND CHERITON, D. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (1996), OSDI '96.

[20] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures* (2010), SPAA '10.

[21] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing* (2008), DISC '08.

[22] HOEFLER, T., DINAN, J., THAKUR, R., BARRETT, B., BALAJI, P., GROPP, W., AND UNDERWOOD, K. Remote memory access programming in MPI-3. *ACM Trans. Parallel Comput.* (Mar. 2013).

[23] HUANG, J., OUYANG, X., JOSE, J., W.-U. R. M., WANG, H., LUO, M., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High-performance design of HBase with RDMA over Infiniband. In *Parallel and Distributed Processing Symposium* (2012).

[24] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference* (2010), USENIX'10.

[25] IEEE. 802.1Qau - Congestion Notification, 2010.

[26] IEEE. 802.1Qbb - Priority-based Flow Control, 2011.

[27] INFINIBAND TRADE ASSOCIATION. Supplement to InfiniBand Architecture Specification Volume 1 Release 1.2.2 Annex A16: RDMA over Converged Ethernet (RoCE), 2010.

[28] ISLAM, N. S., RAHMAN, M., JOSE, J., RAJACHANDRASEKAR, R., WANG, H., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High performance RDMA-based design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 35.

[29] JOSE, J., SUBRAMONI, H., KANDALLA, K., WASI-UR RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable Memcached design for Infiniband clusters using hybrid transports. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2012).

[30] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing* (2011).

[31] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (1997), STOC '97.

[32] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems* (1981).

[33] LI, B., ZHANG, P., HUO, Z., AND MENG, D. Early experiences with write-write design of NFS over RDMA. In *Networking, Architecture, and Storage, 2009. NAS 2009. IEEE International Conference on* (2009), IEEE, pp. 303–308.

[34] LIU, J., JIANG, W., WYCOFF, P., PANDA, D., ASHTON, D., BUNTINAS, D., GROPP, W., AND TOONEN, B. Design and implementation of MPICH2 over Infiniband with RDMA support. In *Parallel and Distributed Processing Symposium* (2004).

[35] LIU, J., WU, J., AND PANDA, D. K. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming 32*, 3 (June 2004).

[36] MELLANOX TECHNOLOGIES. Connect-IB: Architecture for Scalable High Performance Computing, 2013.

[37] MITCHELL, C., YIFENG, G., AND JINYANG, L. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), USENIX ATC'13.

[38] 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* (2013), NSDI'13.

[39] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), SOSP '11.

[40] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms* (2004).

[41] SETHI, R. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM* (1982).

[42] SHIPMAN, G., WOODALL, T., GRAHAM, R., MACCABE, A., AND BRIDGES, P. Infiniband scalability in Open MPI. In *Parallel and Distributed Processing Symposium* (2006).

[43] STETS, R., DWARKADAS, S., HARDAVELLAS, N., HUNT, G., KONTOTHANASSIS, L., PARTHASARATHY, S., AND SCOTT, M. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *ACM SIGOPS Operating Systems Review* (1997), vol. 31, ACM, pp. 170–183.

[44] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (2001), SIGCOMM '01.

[45] SUBRAMONI, H., POTLURI, S., KANDALLA, K., BARTH, B., VIENNE, J., KEASLER, J., TOMKO, K., SCHULZ, K., MOODY, A., AND PANDA, D. K. Design of a scalable InfiniBand topology service to enable network-topology-aware placement of processes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012).

[46] TREIBER, R. K. Systems programming: Coping with parallelism. Tech. Rep. RJ 5118 (53162), IBM, Thomas J. Watson Research Center, 1986.

[47] UPC CONSORTIUM. UPC Language Specifications, v1.2., Technical Report. Lawrence Berkeley National Laboratory LBNL-59208, 2005.

[48] WU, J., WYCKOFF, P., AND PANDA, D. PVFS over InfiniBand: Design and performance evaluation. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on* (2003), IEEE, pp. 125–132.

[49] WWW.OPENSHMEM.ORG. OpenSHMEM Application Programming Interface, 2012.

[50] YELICK, K., SEMENZATO, L., PIKE, G., MIYAMOTO, C., LIBLIT, B., KRISHNAMURTHY, A., HILFINGER, P., GRAHAM, S., GAY, D., COLELLA, P., ET AL. Titanium: A high-performance Java dialect. *Concurrency Practice and Experience 10*, 11-13 (1998), 825–836.

Easy Freshness with Pequod Cache Joins

Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula*, Yandong Mao*, Robert Morris*
*Harvard University and *MIT CSAIL*

Abstract

Pequod is a distributed application-level key-value cache that supports declaratively defined, incrementally maintained, dynamic, partially-materialized views. These views, which we call *cache joins*, can simplify application development by shifting the burden of view maintenance onto the cache. Cache joins define relationships among key ranges; using cache joins, Pequod calculates views on demand, incrementally updates them as required, and in many cases improves performance by reducing client communication. To build Pequod, we had to design a view abstraction for volatile, relationless key-value caches and make it work across servers in a distributed system. Pequod performs as well as other in-memory key-value caches and, like those caches, outperforms databases with view support.

1 Introduction

Web developers use application-level key-value caches such as memcached [2] to improve the performance of database-backed sites. Caches can store *base* data, meaning copies of database records; this improves performance by offloading reads from a bottleneck persistent store. More benefit can be gained by caching *computed* data, which derives from base data but is organized more conveniently for readers. Unfortunately, neither approach is easy to program. Twitter and Facebook, for instance, organize their persistent stores by posting user and time [12, 20], but their primary read operations combine and filter many users' data streams into "timelines" based on user subscriptions. A base-data cache for this access pattern would make reads difficult: the application would basically design and execute a query plan on the cache. A computed-data cache that stored the results of complex timeline queries would simplify reads, but complicate writes: developers would have to invalidate or update cached computed results as necessary to maintain freshness. Developers have even responded to these challenges by building application-specific caching systems [12].

Pequod is a general-purpose distributed key-value cache that transparently keeps computed results up to date. Its central idea is the *cache join* abstraction, which compactly expresses the transformations required to turn input cached data into computed results. The developer installs cache joins in advance. Pequod uses cache joins both to compute data requested by clients, and to update cached results in response to updates. Pequod handles the complex task of updating cached computed data with little developer effort.

A cache join is a materialized view [22] implemented in a distributed key-value cache rather than a relational database. It declaratively defines computed data in terms of simple transformations of base data. We show that materialization, where the cache stores computed data and keeps it up to date, is important for performance. Since cached data is by definition partial, Pequod's materialized views must be both partial and dynamic [28, 30]. Cache joins also offer control over data freshness, supporting both periodic snapshots and incremental updates that keep computed data fully up to date [19]. Pequod thus combines many advanced features, supporting in one system distributed, incrementally maintained, both eager and lazy, dynamic, partially-materialized views. Although Pequod builds on implementation strategies from database materialized views, the non-relational, distributed, key-value-cache context required changes both to the cache join abstraction and to its implementation.

Cache joins help Pequod improve application performance. Computing a result often involves reading extra data; executing this computation in servers, rather than clients, reduces network traffic. Pequod also uses dependency records and hints to efficiently maintain its cache in response to updates.

Our work offers several contributions. We observe that simple joins, filters, and aggregations can express relationships among cached data for many applications, and that can be applied to a distributed, ordered key-value cache. We describe the *cache join* abstraction for key-value materialized-view-like queries and query execution plans. We provide efficient policies for computing these cache joins based on application queries. We design a distributed system that supports cache joins using cross-server data subscriptions and update notifications. Finally, we evaluate Pequod on two example applications inspired by popular websites, a Twitter-like microblogging service and a Hacker News-like news aggregator with user karma. We compare our system with existing technologies and find that Pequod preserves key-value cache performance, despite the addition of cache join execution. We show that moving computation into

cache servers can improve overall system performance, and demonstrate that a deployment of Pequod can be scaled to handle Web-class workloads.

2 Design

Pequod is an ordered key-value cache with string keys and values. It supports four basic operations: get(*k*) returns a value; put(*k*, *v*) updates a value; remove(*k*) removes a value; and the ordered scan(*first*, *last*) operation returns a lexicographically-ordered list of those key-value pairs with keys in the given range. Pequod is not a database and, as is usual for key-value caches, it doesn't support multi-key transactions.

To support freshness, base data in a Pequod cache must be kept up to date relative to the persistent backing store (typically a database). A convenient way to do this is to connect Pequod with a database shard, instructing Pequod that some keys can be found in the database and instructing the database that updates to relevant tables should be forwarded to Pequod (e.g., using Postgres's `notify` statement). If a request is made for a database-sourced key, Pequod will query the database and cache the result, and the database will keep Pequod abreast of any changes. Pequod thus acts as a write-around cache: application writes go directly to the database, and applications access Pequod only for reads. We describe the design of Pequod using a write-around deployment, though other deployments are possible (such as write-through or lookaside caching).

We describe Pequod with reference to Twip, an application that models the core of Twitter, but its ideas apply to many applications that use materialized views.

2.1 Caching Twip

A Twip user can *post* tweets, *follow* other users (subscribe to their tweets), and check her *timeline*. This last operation is the most complex: when user `ann` checks her timeline, Twip returns, in a time-sorted list, all recent posts made by any user `ann` follows.

A Twip database store might use two tables, `p` for posts and `s` for subscriptions. `p`'s columns would be `poster` (the user ID of the posting user), `time`, and `tweet`; `s`'s columns would be `user` (the ID of the subscribing user) and `poster` (the ID of the followed user). This query satisfies a timeline check by user `ann` for all interesting tweets posted after time 100:

```
select p.time, p.poster, p.tweet from s, p
   where s.user='ann' and s.poster=p.poster
     and p.time>=100 order by p.time;
```

Timeline checks are frequent, routinely outnumbering new posts by a factor of 100 [20]. They are also expensive. As the query makes clear, a single timeline check must collect information from all a user's subscriptions, and Twitter users average more than 100 subscriptions each [9]. Executing frequent, expensive, latency-sensitive queries on a persistent database is inadvisable. Timeline checks must be cached.

The cache could simply hold base data. For example, the key-value pair `p|bob|100` ↦ `Hi` might represent user `bob`'s tweet of `Hi` at time 100, and the pair `s|ann|bob` ↦ `1` might represent `ann`'s subscription to `bob`. However, a base-data-only cache shifts the query planning burden onto the application. To check `ann`'s timeline, Twip must read her subscriptions with a scan on the `s|ann|` range, and then, for each subscription, scan the corresponding `p|` range for tweets. This involves at least two rounds of RPCs (and, for typical users who follow many others, hundreds of RPCs total).

To simplify timeline checks, the cache could store computed data. Keys starting with `t|ann|` could store `ann`'s timeline, with copies of tweets by the users `ann` follows. For instance, the `bob` tweet above would correspond to the key-value pair `t|ann|100|bob` ↦ `Hi`. The order of components in this key is semantically important and follows from the lexicographic order of the scan operation. Since each user has their own timeline and tweets are sorted by time, the user and post time must follow the `t|` prefix in that order. We assume multiple followed users might post tweets at the same time, so the key includes the poster to disambiguate them. `Ann`'s timeline check at time 100 would be implemented by a single scan on the half-open range [`t|ann|100|`, `t|ann|`$^+$).[1] This resembles Redis's demonstration Twitter [4] and to some extent the actual Twitter [20], though these services store user timelines as Redis list values (since Redis is not an ordered store) and Twitter stores tweet IDs rather than texts.

Timeline checks are certainly simpler when computed data is cached, but posts and subscriptions are more complicated. Any post must update all relevant timelines. Performance will be good—shifting overhead from read operations to writes is often useful—but implementation remains complex. Furthermore, since the cache might evict a timeline, the application must still contain code to construct timelines from posts.

2.2 Basic cache joins

Pequod aims to help applications avoid this complexity. A Twip application using Pequod can write posts to the database and read timelines directly from the cache without worrying about subscriptions, and with performance as high as that of a typical application cache. The key

[1] The notation `t|ann|`$^+$ represents the upper bound of the `t|ann|` range: [`t|ann|`, `t|ann|`$^+$) contains exactly those keys starting with `t|ann|`. In the Pequod implementation, this upper bound is implemented by the unsightly string `t|ann}`.

idea is Pequod's declarative *cache join* specifications, which relate computed data (such as timelines) to base data (such as posts and subscriptions). The Twip **timeline cache join**

```
t|user|time|poster = check s|user|poster
    copy p|poster|time;
```

defines the value of `t|`*user*`|`*time*`|`*poster* as a copy of the value of `p|`*poster*`|`*time* whenever `s|`*user*`|`*poster* exists. Cache join semantics are that of a simple SPJ database query; the syntax is inspired by Datalog [14] "`t`(*user*, *time*, *poster*, *tweet*) :- `s`(*user*, *poster*), `p`(*poster*, *time*, *tweet*)." However, where Datalog and database queries treat all columns alike, cache joins distinguish keys and values. The poster's tweet, which is the *tweet* column in Datalog and SQL, is implicitly referenced in Pequod by the cache join's `copy` operator.

Pequod computes cache joins using strategies effective for application caches. When the timeline cache join is installed, Pequod contains no `t|` keys. Instead, Pequod dynamically materializes the required results in response to scans of the `t|` range. A timeline scan on [`t|ann|100`, `t|ann|`$^+$) would (in a write-around deployment) ensure that the `s|ann` subscriptions and all relevant `p|` posts were cached, then join those keys to produce the relevant timeline. In addition to returning to the client, Pequod caches the computed timeline and installs updaters that keeps it up to date. If `bob` tweets again at time 120, the database will notify Pequod, which will put the new tweet into key `p|bob|120`. This put triggers a process that automatically copies the tweet to key `t|ann|120|bob`. Since Pequod is eagerly and incrementally maintaining `ann`'s timeline, her later timeline checks can execute without additional computation. Pequod also handles subscription changes: modifications to the `s|ann|` range cause incremental recomputation.

2.3 Advanced cache joins

Cache joins can specify query plans and types of maintenance as required by the application. For example, Twitter celebrities can have tens of millions of followers. Copying their posts into so many timelines will use a lot of memory. Since there are relatively few such celebrities and relatively few celebrity posts, it can be more memory-efficient to handle their posts in a different way [26]. For instance:

```
ct|time|poster = copy cp|poster|time;
t|user|time|poster = check s|user|poster
    copy p|poster|time;      // (1) non-celeb, same as above
t|user|time|poster = pull copy ct|time|poster
    check s|user|poster;     // (2) celebrity
```

Here each user's timeline is calculated from *two* cache joins with different execution strategies. The first, for non-celebrities, is eagerly maintained. The second, for celebrities, is not: the `pull` annotation tells Pequod to recompute the join on each request without caching the results. Celebrity users store their posts under `cp|` keys, rather than the usual `p|`; a helper range, `ct|`, combines all celebrity posts in time-primary order. To compute the celebrity portion of a timeline, Pequod checks the `ct` range for celebrity posts, then filters the results through the user's subscriptions. If no celebrities have posted in a time range (a common case), Pequod will complete the celebrity join with a single fast scan. In our tests, celebrity timelines don't offer performance advantages, but they do save memory.

Cache joins can also colocate different classes of values into the same range of keys. This powerful use case doesn't fit easily into a relational model. For instance, consider a Hacker News-like news application with user karma [1]; we call our version Newp. Users can author new articles, comment and vote on articles, and read article pages. An article page shows the article's vote count, its comments, and the "karma" for each user who commented on the article, where a user's karma is the count of votes on the articles that user authored. A cache for Newp might store articles in one key range, comments in another, and votes in another. Karma, which involves an expensive sum over all of a user's votes, should be precalculated and cached as well. Thus, the data necessary to render an article would be spread over many key ranges. But in Pequod, separate cache joins can bring these disparate objects into a single `page|` scan range, as shown in Figure 1. As a result, Newp can issue one scan on [`page|bob|101`, `page|bob|101|`$^+$) to retrieve all of the disparate data needed to render an article page. We call cache joins like this *interleaved* since they interleave results from logically different computations in a single output range.

Newp also shows that Pequod cache joins can aggregate input data. The `karma` join uses a `count` operator to reduce a range of inputs (all keys starting with `vote|`*author*) into a single value, namely the count of the number of inputs. Pequod supports several simple aggregation functions, including `sum`, `count`, `min`, and `max`. Aggregated data is kept up to date just like copied data, and aggregations are easy to add.

2.4 Distributed Pequod

Distributed Pequod supports workloads too large, or too compute-intensive, for a single server to handle. Base data is replicated across servers as necessary to support the maintenance of computed data (cache join outputs). Each base key has a *home server* to which updates are directed (a partition function maps key ranges to home servers). When a base key k is read from a server S other than its home server H, S requests k's value from

```
karma|author = count vote|author|id|voter;
rank|author|id = count vote|author|id|voter;
page|author|id|a = copy article|author|id;
page|author|id|r = copy rank|author|id;
page|author|id|c|cid|commenter =
  copy comment|author|id|cid|commenter;
page|author|id|k|cid|commenter =
  check comment|author|id|cid|commenter
  copy karma|commenter
```

Figure 1: Interleaved cache joins bring the data necessary to render a Newp article into one contiguous range. Key tags like `|a` and `|r` help the application distinguish types of data.

H. In addition to returning the value, *H* installs a *subscription* for *S* to *k*. When *H* receives an update to *k*'s value, it will send the new value to *S*. Pequod thus maintains eventually-consistent replicas of base data. Computed data is distributed across servers according to client requests. To compute a cache join, a server first fetches all relevant base data into memory (possibly accessing home servers or the backing store, and possibly working through intermediate cache joins), then runs without further communication.

Since cache joins can execute anywhere, adding servers to a Pequod deployment increases its computational capacity. Base data subscriptions also make replication-based load balancing possible: directing reads for popular data ranges to multiple Pequod servers establishes incrementally-maintained replicas that can distribute query load. Adding more servers to a deployment also increases the system's storage capacity, but due to the data duplication and subscription overhead inherent in our design, a Pequod cache's storage capacity does not necessarily rise linearly with the number of servers. Data duplication is reduced when clients send their requests to appropriate servers. Twip clients avoid redundant timeline storage by sending all timeline checks for user *u* to a specific server $S(u)$. This is especially important since timelines make up the large majority of system data (each tweet is stored once in the poster's `p|` range and many, many times in followers' `t|` ranges). But some duplication is unavoidable—for instance, a popular user's tweets are copied to all servers, inducing network overhead—and other applications might be unpartitionable.

Our initial design goal for distributed Pequod has been simplicity rather than completeness, and we do not focus on consistency or resilience to failure. Pequod is eventually consistent: every update to base data eventually becomes visible to all interested servers, but since update propagation is asynchronous, different servers might see updates at different times. The maximum update delay depends on network properties, and is relatively low for our expected deployments (several servers within a single data center). Many Web applications are tolerant of this kind of inconsistency. For some applications, the current Pequod prototype can also support "read-your-own-writes" consistency, where writes made by a client are always visible to later reads by the same client. To achieve this level of consistency, clients must read from and write to a single Pequod cache server (base data are written directly to Pequod to avoid the asynchrony of database notification).

2.5 Eviction

Pequod can evict data under memory pressure. Three types of data can be evicted: data computed by a cache join, remote data copied from another Pequod server via subscription, and cached base data, which in our expected deployment is loaded on demand from a database. Eviction requires modifying the store and invalidating all dependent computed data (which can have transitive effects). At present, an overloaded Pequod server simply evicts the least recently used data ranges. This could be improved by considering the expected costs of reloading a range (the latency of fetching base data from the database, the CPU cost of recomputing dependent ranges, and so forth).

2.6 Discussion

Pequod simplifies application design by adding intelligence to the caching layer. But databases already support that intelligence; why not just use one? Applications couldn't afford to use the main persistent database, but perhaps, as in DBProxy [7, 8], another relational database could be used as a cache. This kind of deployment would be an excellent choice if it performed well. Since Web applications already rely on key-value cache performance, and since some of that performance derives from the key-value model, we chose instead to add intelligence to a key-value cache.

3 Cache joins

A cache join declares how to calculate some output key-value pairs from input key-value pairs, which we call *sources*. Since cache joins are expressed in terms of key-value pairs, which have a single meaningful index ordering (namely, lexicographic key ordering), cache joins also expose meaningful server performance properties, making them resemble both database views and query plans.

A cache join specification has four parts. (1) An *output pattern*, such as `t|`*user*`|`*time*`|`*poster*, defines the format of output keys. (2) One or more *source patterns*, such as `copy p|`*poster*`|`*time*, select keys whose values are used to compute results, and define the operators applied to these keys. (3) Optional *performance annotations* (including the order of source patterns) guide query exe-

```
<cachejoin> ::= <key> "=" ["push" | "pull" |
                "snapshot <T>"] <sources>;
<sources>   ::= <source> | <sources> <source>;
<source>    ::= <operator> <key>;
<operator>  ::= "copy" | "min" | "max" | "count"
                | "sum" | "check";
```

Figure 2: Cache join grammar.

cution; see §3.4. (4) *Slot definitions* tell Pequod how to unpack a key into its component slots—for example, by looking for vertical bars, or by taking fixed numbers of bytes. We don't explain slot definitions further.

Pequod supports several source operators. `Copy` tells Pequod to copy the corresponding source's value to the output key. `Check` marks sources whose values aren't interesting. (For example, in the timeline join, subscriptions like `s|`*user*`|`*poster* are used only for the contents of their keys.) There are also several aggregate functions; like SQL's aggregate functions, they combine many source values into a single output.

Unlike a database query, a cache join exposes the performance properties of key ordering (in relational databases index structure is specified elsewhere), and exposes more performance properties through annotations, but has less flexibility. For instance, cache joins must not be recursive (a cache join's output cannot be used as one of its sources), and relationships between "tables" must be expressed entirely through keys.

Users define cache joins in textual form (Figure 2 summarizes the grammar) and install them using an "add-join" RPC. Multiple cache joins may be installed over the same range of keys. One cache join can act as a source for another; this can be useful, for example to permute keys into a more convenient order, but risks cascading invalidations. Performance annotations such as `snapshot` *T* (§3.4) can mitigate this problem somewhat. Users should not install circular cache joins.

Pequod checks for errors (such as recursive queries) at cache join installation time, but some errors are difficult to identify in advance. For instance, consider the timeline join variant `t|`*user*`|`*time* `= check s|`*user*`|`*poster* `copy p|`*poster*`|`*time*. Since this lacks the "`|`*poster*" suffix for timeline keys, it produces undefined results when two or more posters post tweets at the same time. (A corresponding database query would produce one tuple per relevant tweet, but Pequod values are strings, not tuples, and the `copy` operator doesn't know how to combine multiple values into a single string.) But it's not necessarily appropriate to reject such joins out of hand; perhaps the application ensures that no two tweets have the same time. Thus Pequod's users are left responsible for avoiding ambiguous cache joins, either by preventing output collisions or by using aggregate functions with well-defined behavior.

We currently impose additional technical requirements on cache joins. For instance, in a join with n sources, exactly $n-1$ of the operators must equal `check`, and we constrain the use of `count` and `sum` operators to simple cases.

3.1 Cache join query execution

When Pequod receives a scan(*first*, *last*) or get(*key*) request that overlaps with one or more cache joins, it must execute the queries they represent. This section describes the semantics and implementation of cache join query execution. We focus on forward query execution, which starts from base data. The algorithm used by Pequod is a key-value variant of the classical nested-loop implementation for database join queries. The next section describes how incremental maintenance works.

Cache join execution logically enumerates all tuples of source keys and selects those that match the join's constraints. The selection step ensures that there's one key per source, that the key formats match the source patterns, and that slots common to multiple source keys have consistent values. Pequod then combines the selected values according to the join's operators and installs the results.

For the timeline join `t|`*user*`|`*time*`|`*poster* = `check s|`*user*`|`*poster* `copy p|`*poster*`|`*time*, the keys ⟨`s|ann|bob`,`p|bob|100`⟩` match the source patterns: the first key matches the first source, the second key matches the second source, and the shared *poster* slot has a consistent value in both keys (`bob`). An output key can be derived from the output pattern and the source keys; here, that key would be `t|ann|100|bob`. The join's operators (`check` for the `s` source and `copy` for the `p` source) indicate that `p|bob|100`'s value should be copied to `t|ann|100|bob`.

The main optimization strategy for nested-loop queries moves selection operators as early as possible, since this avoids enumerating irrelevant tuples. In relational databases, selection operators are functions on columns. In Pequod, selection operators are functions on *ranges*, especially as they map to the "columns" defined by pattern slots. For example, when given a scan query over the timeline range `[t|ann|100,t|ann|200)`, Pequod can limit its examination of subscriptions to the range `[s|ann|,s|ann|`$^+$`)`; and for each resulting subscription `s|ann|`*poster*, it can examine the limited post range `[p|`*poster*`|100,p|`*poster*`|200)`.

Figure 3 outlines our query execution algorithm; though our implementation is generic, the pseudocode uses the timeline join for concreteness. Two related concepts, slot sets and containing ranges, help move selection early. A *slot set* is a set of slot assignments derived from a cache join and a key or key range. For example, in

compute timeline(*first*, *last*):
  ss := *timelinejoin*.slotset(t, *first*, *last*)
  $[ks^-, ks^+)$:= *ss*.containingrange(s, *first*, *last*)
  for each $ks \mapsto vs$ where $ks \in [ks^-, ks^+)$
      and *ks* matches s|*user*|*poster*:
    ss' := *ss*.addslots(s, *ks*)
    $[kp^-, kp^+)$:= ss'.containingrange(p, *first*, *last*)
    for each $kp \mapsto vp$ where $kp \in [kp^-, kp^+)$
        and *kp* matches p|*poster*|*time*:
      yield t|*user*|*time*|*poster* $\mapsto$ *vp*

Figure 3: Query execution for the timeline cache join.

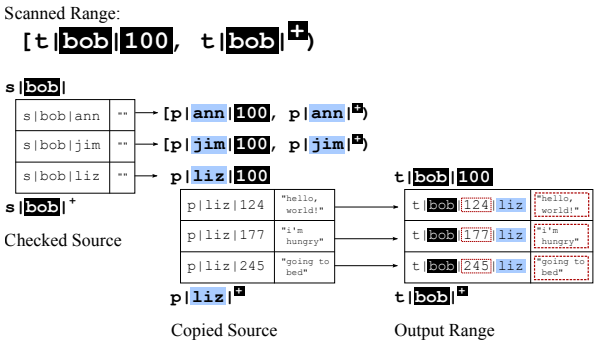


Figure 4: Example query execution of the timeline join. The scanned range provides context used when scanning source ranges. The keys and values in the output key range comprise elements of the original scan range and both source ranges.

compute timeline(*first*, *last*):
  for each subrange $[x^-, x^+) \subset [first, last)$ where
      joinstatus($[x^-, x^+)$) $\neq$ VALID:
    ss := *timelinejoin*.slotset(t, x^-, x^+)
    $[ks^-, ks^+)$:= *ss*.containingrange(s, x^-, x^+)
    js := new join status range for $[x^-, x^+)$
    add updater from $[ks^-, ks^+)$ to *js*
    for each $ks \mapsto vs$ where $ks \in [ks^-, ks^+)$
        and *ks* matches s|*user*|*poster*:
      ss' := *ss*.addslots(s, *ks*)
      $[kp^-, kp^+)$:= ss'.containingrange(p, x^-, x^+)
      add updater from $[kp^-, kp^+)$ to *js*
      for each $kp \mapsto vp$ where $kp \in [kp^-, kp^+)$
          and *kp* matches p|*poster*|*time*:
        yield t|*user*|*time*|*poster* $\mapsto$ *vp*
    install *js* as VALID

Figure 5: Query execution for the timeline cache join, including installation of data structures for later incremental maintenance. Changes from Figure 3 are in black.

the timeline join, the key s|ann|bob corresponds to the slot set {*user* $\mapsto$ ann, *poster* $\mapsto$ bob}. Query execution begins by deriving a slot set from the requested output key range; for example, given scan(t|ann|100, t|ann|+), Pequod creates the slot set {*user* $\mapsto$ ann}. Slot sets are augmented with additional slot assignments as Pequod works through source keys. A *containing range* is effectively the inverse of a slot set. Given a slot set, a source pattern, and the requested output key range, Pequod can calculate a minimal range of source keys that might affect the scan's results. For example, given the scan request above and the slot set {*user* $\mapsto$ ann, *poster* $\mapsto$ bob}, the minimal containing range for the p source would be [p|bob|100, p|bob|+). Any post outside that containing range would either not match the required *poster*, or not map to an output key in the requested output key range. But even with containing ranges, the algorithm must compare the source range keys with their patterns. As a schema-free key-value store, Pequod might have keys in the range that don't match the source patterns. Figure 4 shows a sample execution of this algorithm for the timeline join.

Since Pequod should support any application and provide general key-value cache semantics, we took care to handle any range query. Thus, for example, we correctly implement queries like [t|ann|100, t|bob|200) and [t|a, t|b) that cross multiple timelines. Correct and minimal containing ranges are generated in each case.

3.2 Incremental maintenance

Pequod can keep computed data up to date as sources change. Eager incremental maintenance transfers work from read queries to writes and saves computation on exact re-requests. The view selection problem [22] is easy in Pequod: since all queries are range scans, precomputed ranges naturally benefit queries that partially or completely overlap.

Pequod implements incremental maintenance through two auxiliary data structures. A *join status range* indicates whether a range of keys is up to date with respect to the cache joins whose outputs overlap that range. Join status ranges are logically attached to output ranges and form a disjoint cover of key space (every key is associated with exactly one join status range). *Updaters*, in contrast, logically attach to source ranges. An updater links a range of source keys with a *context*—a cache join, a slot set, and a join status range. The context provides the information required to maintain its join status range. For instance, a range of posts [p|bob|100, p|bob|+) might have an updater for the timeline join with slot set {*user* $\mapsto$ ann}; this *user* value lets Pequod map source key p|bob|200 to output key t|ann|200|bob. Many updaters can apply to a given key, so we store updaters in an interval tree. Whenever Pequod modifies its store, it finds all updaters applicable to the modified key and runs the indicated incremental maintenance for each. Figure 5 outlines how join status ranges and updaters are installed during forward query execution.

The notification provided to an updater includes the modified source key, the new value, the old value, and the type of change (insert new key, update existing key, or remove existing key). The updater reacts by modifying either the attached join status or the cache's value for some key. The form of this modification depends on the relevant operator; for example, the updater for `copy` operators calculates the appropriate output key and inserts the value there.

Pequod also supports *invalidations*, which provide a form of *lazy* view maintenance [29]. Unlike an eager updater, which updates the cache upon notification, an invalidating updater just marks its join status range as INVALID. The invalidity will be detected and corrected when the output range is queried. There are two kinds of invalidation. Complete invalidation removes installed updaters and requires that a range be recomputed from scratch. Partial invalidation instead logs the source modification into an entry on the relevant join status range. The logged modification—or a subset of it—will be applied later, when the output range is queried [29]. Lazy maintenance, and especially partial invalidation, shifts some of the burden of view maintenance back onto read operations from write operations. Our prototype uses lazy maintenance (invalidations) for `check` sources and eager maintenance for all other sources, a choice that performs well for our applications. For example, Twip subscription changes logically shift many tweets into or out of a timeline. Thanks to lazy maintenance, however, Pequod shifts only those tweets strictly required by queries. Since most timeline checks are updates, rather than loads of past tweets, this can perform much less work than eager maintenance would. Our policy would not work equally well for all applications, however, and we would like to offer users more control over maintenance type.

Several important optimizations improved Pequod's performance by large factors. Updaters frequently overlap; for example, a Twip user's posts have one updater per subscriber. It was especially important to combine such updaters whenever possible. If a new updater is installed for the same source range as an existing updater—or for an overlapping range—Pequod reduces memory usage and the size of the updater tree by appending information about the new updater to the existing one. Other important performance improvements were obtained by compressing or eliminating the context information stored with updaters, since in many cases Pequod can derive an output key completely from the source key and the relevant join status range. (Consider a timeline-join updater on source range [`p|ren|200`,`p|ren|`$^+$) associated with join status range [`t|ann|100|`,`t|ann|`$^+$). The join status range uniquely determines the *user* slot, and the source key uniquely determines the *poster* and *time* slots.)

3.3 Resolving missing data

The join query execution algorithm in Figure 5 assumes that all required source data is present in the cache. However, this assumption need not hold. A source range might overlap with the output of another cache join; the source range might exist in the persistent backing store, but have expired from the cache; and in distributed Pequod, the source range might be stored on a different server. Pequod detects these cases and loads the data as required before executing the join query. The first case can be handled with a recursive query execution. In the second and third cases, the data is loaded and metadata is installed to indicate its presence.

Pequod reduces query latency by loading missing base data in parallel. When base data is missing, Pequod initiates an asynchronous fetch request (to the database or to a home server) and attaches a *restart context* to the current join status range. It then continues to execute the query using any cache-resident data. When all required fetches complete, the restart contexts are used to restart the query. The restarted query behaves as if executed from scratch, so every server query produces results consistent with a snapshot of that server's state. However, Pequod doesn't recompute any parts of the query that were already completed and haven't been invalidated since. In general, a query execution is iteratively evaluated until there are no outstanding restart contexts and the join status range is marked VALID; in most cases, this requires at most one round of fetches and a cleanup execution that fills in gaps. The resulting output range is scanned to construct a client response.

We always resolve missing base data by loading it into server memory. This allows us to execute cache joins locally. Other strategies, such as executing cache joins in partitioned fashion and aggregating the results [5], could benefit some applications by reducing data movement.

3.4 Performance annotations

Cache joins contain annotations that affect evaluation performance. First, the *maintenance type* defines whether and how a join is kept up to date. The default `push` type asks for the incremental maintenance described in §3.2. A `pull` join, in contrast, is calculated from scratch on each query using the procedure in §3.1. As we saw for Twitter celebrities (§2.3), this can save memory for some applications and data patterns. Finally, a `snapshot` *T* join implements deferred view maintenance. Pequod calculates the join from scratch, but caches the result—without further updates—for *T* seconds. Snapshot joins induce less maintenance overhead than push joins and less computation overhead than pull joins. Second, the order of sources is a performance annotation. Pequod examines the source descriptions in a cache join in order, and different source orders can perform quite differ-

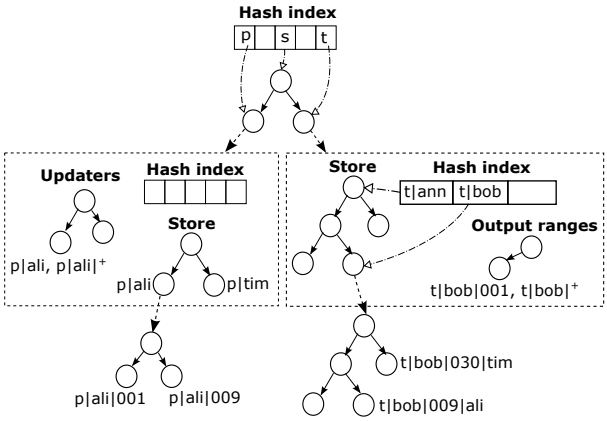


Figure 6: Pequod internal data structure for Twip. The logical store divides into tables (the rectangle layer) and, when appropriate, subtables (the lowest layer).

ently. Joins are generally more efficient when the smallest ranges are examined first, since this reduces the data to be examined later. It's usually best to arrange sources so that slots are encountered in the same order as in the output key, since this leads to small containing ranges. The user may be able choose an even better order using their knowledge of key counts and update frequencies.

4 Implementation

Pequod is a single-threaded, event-driven C++ program. Pequod uses red-black trees to store key-value pairs and bookkeeping information, such as updaters and join status ranges. Several optimizations use auxiliary data structures such as hash tables to reduce tree lookups, improving performance significantly for some workloads. This section describes Pequod's implementation, focusing on these optimizations and when they are useful. Figure 6 shows the overall arrangement of Pequod's internal data structures as configured for Twip.

4.1 Structure

Pequod stores data in several layers of tree visible to clients as a single ordered key-value store. The first tree layer separates logical *tables*, such as `p|` and `t|`, into separate subtrees. Each table stores the relevant key-value pairs and bookkeeping structures (an interval tree of updaters and a tree of join status ranges). By separating concerns for different ranges, this design sped up Pequod significantly relative to a one-level store.

Tables can themselves be subdivided. Many applications have natural key boundaries across which scans are rare; for example, Twip scans mostly lie within a timeline range. If developers mark these boundaries, Pequod will use them to break the store into *subtables*. Thanks to a hash table that indexes the subtables, operations that lie entirely within a subtable can jump to that subtable in $O(1)$ time (rather than $O(\log N)$). However, the entire key-value store is still ordered, and operations that cross subtable boundaries will execute as expected. The use of subtables improves the runtime of our Twip benchmark by a factor of 1.55x, but increases memory consumption by a factor of 1.17x, a consequence of additional bookkeeping.

4.2 Output hints

In many of our applications, each update to a join status range either modifies the same key as the previous update (as is common for the `count` operator) or inserts a new key immediately after the previous update (as when inserting a fresh post into a Twip timeline). Both types of modification can be performed in $O(1)$ amortized time given a pointer to the last-updated key. Each join status range therefore maintains a pointer to its last updated key. We call this pointer the *output hint*. A reference counting scheme ensures that the hint stays valid even if the underlying key-value pair is removed from the tree. This optimization avoids tree lookups in our Twip benchmark, and improves its performance by a factor of 1.11x.

4.3 Value sharing

The `copy` operator often requires Pequod to install multiple copies of a value into multiple output ranges. For example, Twip inserts a copy of each tweet into each of the interested followers' timelines. To reduce memory overhead, Pequod allows multiple output ranges to share the source's value. This optimization fits in naturally with server computation and might not work as naturally if sharing was entirely directed by application clients. This optimization reduces memory consumption by a factor of 1.14x on our Twip benchmark. Value sharing is only useful for `copy` joins, but it introduces no overhead on other joins.

5 Evaluation

This section evaluates Pequod's performance. Pequod performs well compared with related systems, its materialization strategy works well on our workloads, and unconventional features of its data model (interleaved cache joins) can benefit real applications. Furthermore, distributed Pequod can scale across multiple servers to handle large workloads.

5.1 Experimental setup

We evaluate Pequod using two hardware configurations, a multiprocessor and a cluster of Amazon EC2 virtual machines. The multiprocessor is an Amazon EC2 `cr1.8xlarge` instance with 32 logical processors and 244GB of RAM running Ubuntu Linux 13.04. The Amazon EC2 cluster, used to evaluate scalability, consists

of `cc2.8xlarge` and `cr1.8xlarge` VM instances connected by a 10Gbps network. Each VM has 32 cores, 60-244GB of RAM, and runs Amazon Linux 2013.09.2.

Application clients communicate with Pequod servers using RPC. Experiments on the multicore machine use TCP over the loopback interface for RPC invocation. Clients are event-driven processes that keep many RPCs outstanding. We run enough clients to saturate the Pequod servers.

In most of our experiments, Pequod is configured to run Twip. The underlying data is derived from a Twitter social graph obtained in 2009 [21]. The full graph, which contains 40 million users and 1.4 billion relationships, is used in the scalability experiment (§5.5). All other Twip experiments use a sampled subgraph containing 1.8M users and 72M relationships.

Our clients model the actions of individual Twip users. Each modeled user (1) "logs in," obtaining a list of many recent tweets; (2) repeatedly checks for new tweets, subscribes to other users, and posts tweets of their own; and (3) logs out (though they may log in again later). The incremental timeline updates in step (2) return many fewer tweets than the initial scans at login time. These events occur in the rough ratio 5% initial timeline scans, 9% new subscriptions, 85% incremental timeline updates, and 1% posts, which we derived using information on the real Twitter [20]. Users post with different likelihoods. The probability that a user posts a message is proportional to the log of their follower count, so more popular users tweet more often. In one common workload, 70% of users are active (the remainder never check their timelines) and each active user checks their timeline 50 times. This results in approximately 62M timeline checks, 6.2M new relationships, and 620K new posts over the course of the experiment.

We do not evaluate database interaction or eviction. Pequod is deployed as a look-aside cache: applications send it updates directly. Notification bottlenecks in the database made the performance of our write-around deployment uninteresting. Although we enable eviction, it never triggers in our experiments.

We ran most experiments several times and observed little to no performance variability. Confidence intervals would not be visible on our graphs.

5.2 System comparison

Pequod aims to improve the programmability of application-level caches by offering developers more interesting semantics. These semantics do not compromise performance: Pequod performs no worse than comparable caches.

We evaluate two Twip implementations. The first, "Pequod," is the Twip application described above; timelines are fetched via the timeline join. In the second, "client Pequod," application clients are responsible for maintaining timelines. There are no cache joins. After making a post, the posting client sends a timeline update for every subscribed user. Client Pequod lets us evaluate the performance impact of server-managed computation in isolation. We also evaluate the Redis (version 2.8.5) and memcached (version 1.4.16) key-value caches and a traditional database, PostgreSQL (version 9.1). Each system runs the same workload to completion as fast as possible. Redis and memcached don't support server-side computation, so as in client Pequod, their clients actively manage user timelines; Redis stores timelines as sorted sets of tweets, memcached as a string to which tweets are appended. PostgreSQL, in contrast, does support server-side computation. Although our test version lacks automatically-updated materialized views, we use triggers to get a similar effect. Each system is given six cores in our multicore machine. PostgreSQL runs a single process with multiple threads, while the other systems partition the store and use one process per core. The machine's remaining cores run client processes; for each system, we used the number of client processes that produced the best system runtime. We configure all systems so that data is stored in memory and consistency is relaxed as much as possible.[2]

System	Runtime
Pequod	197.06 s (1.00x)
Redis	262.62 s (1.33x)
Client Pequod	323.29 s (1.64x)
memcached	784.43 s (3.98x)
PostgreSQL	1882.78 s (9.55x)

Figure 7: Time to process a Twip experiment to completion using Pequod and related systems. Smaller numbers are better.

Figure 7 shows the results. Pequod, which uses materialized views, runs a factor of 1.64x faster than client Pequod, which doesn't. The penalty is roughly equally divided between RPC overhead (client Pequod makes many more RPCs) and insertion overhead (client Pequod doesn't benefit from output hints or value sharing). Although a more optimized client-managed caching system could beat Pequod (perhaps by implementing Pequod-like functionality specialized for the application), RPC overhead and program complexity remain as challenges for any client-managed or special-purpose system. Pequod runs a factor of 1.33x faster than Redis: join support does not sacrifice the performance advantages of key-

[2] We disable Redis disk checkpoints and avoid triggering eviction in memcached by configuring the amount of available memory. For PostgreSQL, we allocate a shared memory buffer large enough to hold our entire data set, place the data store in an in-memory file system, and tune for performance: we disable *fsync*, *synchronous commit*, and *full page writes* and set *bgwriter lru maxpages* to zero.

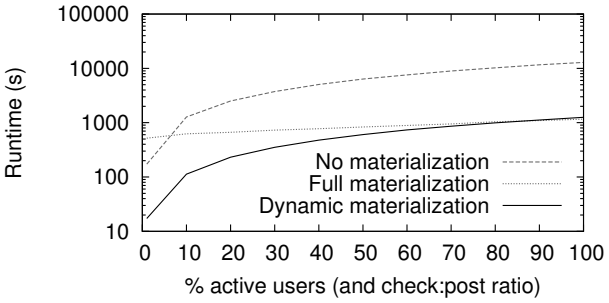


Figure 8: Pequod's dynamically materialized views generally outperform other strategies on the Twip benchmark.

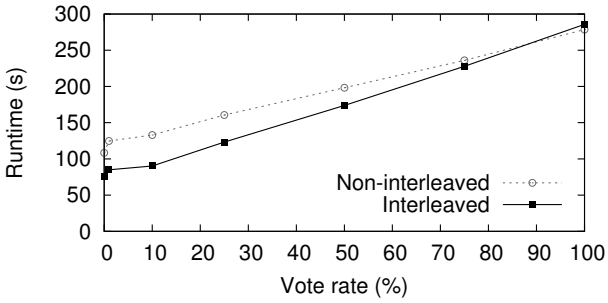


Figure 9: Newp interleaved cache joins perform better than fetching article data in separate RPCs, except when writes are very common.

value caches. Redis runs a factor of 1.23x faster than client Pequod, however. This difference is due to Redis's hash table data structure, which offers $O(1)$ lookups. Though tree optimizations could speed up client Pequod somewhat, unordered stores offer fundamental performance advantages over ordered stores. memcached runs a factor of 3x slower than Redis: the Twip workload has more writes than memcached prefers. The traditional database, despite running in memory with relaxed ACID guarantees, is not a suitable replacement for an application-level cache. Pequod outperforms PostgreSQL by nearly an order of magnitude (9.55x).[3]

To summarize, Pequod performs no worse than widely available key-value caches; for this workload, it even offers a small performance improvement. The additional semantics provided by Pequod simplify application development without compromising performance.

5.3 Materialization strategy

Pequod implements cache joins using a dynamic materialization strategy: queries are computed on demand, but recently-accessed ranges are eagerly and incrementally updated. We compare this strategy with the obvious alternatives, namely no materialization (where no ranges are cached) and full materialization (where all ranges are cached and kept up to date). We create a Twip workload comprising only timeline check and post operations. 1 million posts are distributed among all 1.8M users as described above (proportionally to the log of the follower count). We then vary p, the percentage of "active" users, between 1 and 100. Each workload performs p million timeline checks spread uniformly across the $1.8\text{M} \times p/100$ active users, resulting in a check:post ratio between 1:1 and 100:1. We use five clients and one server, run the workload to completion as fast as possible, and measure the elapsed time.

Figure 8 shows the results. As expected, the no-materialization strategy performs relatively well with few active users, but as timeline scans increase, materialization quickly becomes important for performance. Because it avoids materializing data in which no one is interested, Pequod's dynamic materialization outperforms full materialization up to approximately 90% active users. After that, full materialization performs slightly better (a factor of 1.08x better at 100% active users). This performance difference is due to the join computation dynamic materialization must perform when a user first logs in. Full materialization keeps all timelines up to date at all times; though this avoids login overhead, it inevitably uses more memory when users can be inactive.

[3]Widely-available databases with true materialized view support were also evaluated; they performed similarly to PostgreSQL.

5.4 Cache join choice

Pequod leaves view selection and query planning to the application developer; this flexibility, and the design flexibility offered by the key-value context, can improve application performance.

We evaluate two versions of the Hacker News-like Newp application that use different joins. The first uses separate ranges for aggregate data (karma and vote counts); constructing an article requires many RPCs in two round trips. The second uses the interleaved cache join from §2.3 to colocate this data. Reading an article requires a single scan, but more server computation and storage overhead is incurred (upon each vote aggregate values are copied into each `page|` range). The shared workload has three types of operation: reading an article, commenting, and voting. The Pequod data store is pre-populated with 100K articles, 50K users, 1M comments, and 2M votes. We simulate 20M user sessions; each user reads a random article; with a varying chance votes on the article; and independently with a 1% chance comments on the article. The experiment is run using a single server and multiple clients. We expect the interleaved approach to perform well when article reads far outnumber votes and comments.

The results, shown in Figure 9, indicate that interleaved cache joins are superior for most vote rates tested.

The non-interleaved implementation issues many gets per article (e.g., for karma), each of which incurs overheads including an $O(\log N)$ lookup. The interleaved join improves overall performance until the cost of precomputation outweighs the cost of processing many gets (90% vote rate). Storing votes and karma in a hash table might shift the crossover to the left, but the interleaved join would preserve its main advantage, namely code simplicity.

5.5 Scalability

Our evaluation of distributed Pequod focuses on a problem inherent in cache joins, namely the CPU overhead of cache join execution. Pequod can do more work per request than would a simpler cache, putting pressure on server CPUs. We show that distributed Pequod scales well enough that adding servers reduces this pressure.

Our experimental setup involves a large backing store, which holds the full 2009 Twitter social graph, and a variable number of Pequod compute servers, which execute the timeline join in response to client timeline checks. Both backing store and compute servers are Pequod processes, but the backing store absorbs all writes while the compute servers absorb all reads. Each experiment runs the same Twip workload: 28M active users issue 1.4B timeline checks, make 140M new subscriptions, and generate 14M new posts. We run the workload as fast as possible to ensure that the bottleneck is within Pequod. Measurements indicate that for the server setups measured, in each case, the bottleneck is Pequod compute server CPU. All of a user's compute requests are directed to the same compute server, minimizing unnecessary data duplication. To warm the cache servers, each active user is logged into the system prior to the experiment, ensuring that a join status range exists, base data are present, updaters are installed, and subscriptions are established between the compute and base servers. We use up to 30 virtual machines on the Amazon EC2 testbed. 6 VMs run 32 Pequod servers for the backing store and up to 48 client processes, while the remaining 24 VMs run between 12 and 48 Pequod compute servers.

Figure 10 shows the result. Throughput increases by 3x (from 1.42M to 4.27M qps) as the number of compute servers increases from 12 to 48. Perfect scalability would increase throughput by 4x; unfortunately, some Pequod overheads (such as base data required per compute server) do not drop linearly with the number of compute servers. Total memory consumption increased from 290GB to 297GB at the base servers, a consequence of storing duplicate subscription information. The compute servers stored more total data thanks to base data duplication; total memory used by all compute servers increased from 1.2TB to 1.5TB. Likewise, a larger fraction of the consumed network bandwidth is dedicated to inter-server subscription maintenance, increasing from roughly 10% to 16% between 12 and 48 compute servers (the rest is client communication). Though these overheads are potential bottlenecks to system scalability, Pequod still performs and scales well overall.

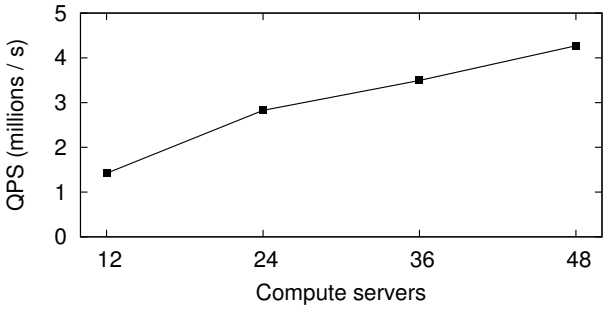


Figure 10: Adding computational capacity results in a speedup for a fixed Twip workload.

6 Related work

Application caches Pequod's key-value design was inspired by existing application-level caches [2, 3].

Pequod is an ordered store. This helped make cache joins simple and useful, but unordered hash table stores, such as Redis and memcached, offer faster $O(1)$ operations. Several of our implementation tricks and optimizations—some of them enabled by the cache join abstraction—reduce the cost of tree walking. To speed up Pequod further, we could replace its binary trees with more cache-efficient structures [24] or, better, investigate cache join variants for unordered stores. (This would probably require structured values à la Redis.)

Previous work has tracked dependencies among cached data in application-level caches [15, 31]. The application developer specifies these dependencies explicitly, either by item or by item class; the systems respond to updates by invalidating dependent objects. In TxCache [25], dependency tracking is automatic, rather than explicit. TxCache provides transactional consistency, which Pequod does not, and can invalidate cached objects that are computed by arbitrary pure functions, rather than SPJ queries. Compared to these systems, Pequod's cache joins define dependencies in a particularly natural way, and Pequod can update dependent objects, which we found faster than invalidating them.

DBProxy [7, 8] is a distributed database cache that can store partial query results in edge cache nodes and service later queries from those caches. The caches are incrementally maintained by a master backend database; new results are produced by re-executing queries (with an exception for some aggregates, which are also stored as exact-match results). Pequod, in contrast, uses eager view maintenance to avoid costly computation on the

read path. DBProxy transparently populates its cache by inspecting queries. Pequod is not transparent: developers decide what data is cached and describe how to maintain that data.

Materialized views Pequod borrows joins and materialized views [11, 16] from relational databases. Many production databases offer materialized views for data warehousing, replication, and storing results of expensive computations. Most views are meant to be transactionally consistent with the underlying data, and are kept up to date either by synchronous updates when base data changes (eager update) or by refreshing the view when it is read (lazy update). We borrow from work on efficiently maintaining views with incremental updates and batch processing [10, 13, 18, 27–29, 32].

Materialized views in Pequod are eventually consistent with respect to base data and are maintained through asynchronous, incremental updates. Views in Pequod can also be partially and dynamically materialized [28, 30], a relatively advanced feature.

Several research systems have applied one or more of these techniques. Agrawal et al. [5] added materialized views to PNUTS [17], a distributed key-value store. Like Pequod, views are implemented as partitioned tables, are eventually consistent, and are maintained with asynchronous, incremental computation. However, this work did not support partial materialization or some of Pequod's performance annotations. Interestingly, the authors use a different execution strategy for aggregate joins, which use a distributed query to reduce data movement. Pequod might benefit from a similar strategy.

Dynamic materialized views (DMV) [30] can partially materialize a view based on data access patterns. In DMV, the selection of rows to materialize can be specified manually or handled at runtime by a feedback loop with policies for admission and eviction. Pequod's dynamic materialized views borrow from DMV, but implement a simple selection policy based on access time.

DBToaster [6] presents a method for deriving incremental update triggers from relational view queries, but only works on aggregate queries and does not partially materialize views.

Luo's partial materialized views (PMVs) cache portions of frequently-executed queries with the goal of allowing early access to partial query results [23]. PMVs are restricted—for instance, they do not support insertions on base data—but a similar feature might be useful for some Pequod applications.

Applications The real Twitter service actively updates the timelines of logged-in users as tweets arrive [20]; this was one inspiration for Pequod's hybrid pull/push architecture. The load on Twitter's service is high: Twitter has more than 150M active users that generate 300K timeline reads per second on average. On a typical day Twitter handles 4-6K new tweets per second (340M per day), resulting in 300K deliveries per second (2.6B per day) to user timelines [20]. Twitter scales its timeline service by partitioning its users amongst an expandable set of cache servers. Our Twip application uses the same strategy. Twip does not support other Twitter features, such as search; we have not investigated whether these features would benefit from Pequod.

Silberstein et al. [26] describe the importance of balancing "pull" and "push" strategies in social networking services, an insight we borrow for celebrity join. Given a Twitter-like application, their system determines at runtime which tweets should be materialized into followers' timelines and which should not. A more complex join operator could conceivably support their algorithm in Pequod.

7 Conclusion

Pequod is a distributed key-value cache that uses a new abstraction, the cache join, to automatically rearrange and transform cached data in ways useful for applications. By understanding how cached data is computed, Pequod is able to keep cached data fresh and provide performance benefits while presenting a simple API to users. As future work, we hope to improve Pequod further by optimizing its data structure design and exploring options for configuration changes and recovery from server failure.

Acknowledgements

We thank the SOSP and NSDI reviewers and our shepherd, Ali Ghodsi, for many helpful comments. This work was partially supported by a Microsoft Research New Faculty Fellowship, by the National Science Foundation (awards 0834415 and 0915164), and by Quanta Computer.

References

[1] Hacker News FAQ. http://ycombinator.com/newsfaq.html.

[2] memcached. http://memcached.org.

[3] Redis. http://redis.io.

[4] Retwis. http://redis.io/topics/twitter-clone.

[5] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for VLSD databases. In *Proc. SIGMOD 2009, ACM SIGMOD Int'l Conf. on Management of Data*, pages 179–192. ACM, June 2009.

[6] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proc. VLDB Endowment*, 5(10): 968–979, June 2012.

[7] K. Amiri, S. Park, and R. Tewari. A self-managing data cache for edge-of-network web applications. In *Proc. CIKM'02, 11th Int'l Conf. on Information and Knowledge Management*, pages 177–185. ACM, Nov. 2002.

[8] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Dbproxy: a dynamic data cache for web applications. In *Proc. ICDE 2003, 19rd Int'l Conf. on Data Engineering*, pages 821–831. IEEE Computer Society, Mar. 2003.

[9] Beevolve, Inc. An exhaustive study of Twitter users across the world. http://www.beevolve.com/twitter-statistics/#b2, Oct. 2012.

[10] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. SIGMOD'86, 1986 ACM SIGMOD Int'l Conf. on Management of Data*, pages 61–71. ACM, May 1986.

[11] R. F. Boyce, D. D. Chamberlin, M. M. Hammer, and W. F. King. Specifying queries as relational expressions. In *Proc. 1973 Meeting on Programming Languages and Information Retrieval*, pages 31–47. ACM, 1973.

[12] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proc. 2013 USENIX Annual Technical Conf.*, pages 49–60. USENIX, June 2013.

[13] M. W. Cain. Creating and using materialized query tables (MQT) in IBM DB2 for i5/OS (version 2.0). Technical report, IBM, Sept. 2006. http://public.dhe.ibm.com/partnerworld/pub/pdf/courses/438a.pdf.

[14] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. Knowledge and Data Engineering*, 1(1): 146–166, Mar. 1989.

[15] J. Challenger, A. Iyengar, and P. Dantzig. A scalable system for consistently caching dynamic web data. In *Proc. INFOCOM'99, 18th Joint Conf. of the IEEE Computer and Communications Societies*, volume 1, pages 294–303, Mar. 1999.

[16] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[17] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endowment*, 1(2):1277–1288, August 2008.

[18] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. In A. Gupta and I. S. Mumick, editors, *Materialized Views*, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

[19] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD'93, 1993 ACM SIGMOD Int'l Conf. on Management of Data*, pages 157–166. ACM, May 1993.

[20] R. Krikorian. Real-time delivery architecture at Twitter. Talk at QCon New York. http://www.infoq.com/presentations/Real-Time-Delivery-Twitter, Oct. 2012.

[21] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proc. WWW 2010, 19th Int'l World Wide Web Conf.*, pages 591–600. ACM, Apr. 2010.

[22] P.-Å. Larson and H. Yang. Computing queries from derived relations. In *Proc. VLDB'85, 11th Int'l Conf. on Very Large Data Bases*, pages 259–269. VLDB Endowment, Aug. 1985.

[23] G. Luo. Partial materialized views. In *Proc. ICDE'07, 23rd Int'l Conf. on Data Engineering*, pages 756–765. IEEE Computer Society, Apr. 2007.

[24] Y. Mao, E. Kohler, and R. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys'12, 7th ACM European Conf. on Computer Systems*, pages 183–196. ACM, 2012.

[25] D. R. K. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov. Transactional consistency and automatic management in an application data cache. In *Proc. OSDI'10, 9th USENIX Conf. on Operating Systems Design and Implementation*, pages 1–15. USENIX, Oct. 2010.

[26] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *Proc. SIGMOD 2010, ACM SIGMOD Int'l Conf. on Management of Data*, pages 831–842. ACM, June 2010.

[27] F. W. Tompa and J. A. Blakeley. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, October 1988. URL `http://dx.doi.org/10.1016/0306-4379(88)90005-1`.

[28] J. Zhou, P.-Å. Larson, and J. Goldstein. Partially materialized views. Technical Report MSR-TR-2005-77, Microsoft Research.

[29] J. Zhou, P.-Å. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *Proc. VLDB'07, 33rd Int'l Conf. on Very Large Data Bases*, pages 231–242. VLDB Endowment, Sept. 2007.

[30] J. Zhou, P.-Å. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. In *Proc. ICDE'07, 23rd Int'l Conf. on Data Engineering*, pages 526–535. IEEE Computer Society, Apr. 2007.

[31] H. Zhu and T. Yang. Class-based cache management for dynamic Web content. In *Proc. INFOCOM'01, 20th Joint Conf. of the IEEE Computer and Communications Societies*, volume 3, pages 1215–1224, 2001.

[32] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. SIGMOD'95, 1995 ACM SIGMOD Int'l Conf. on Management of Data*, pages 316–327. ACM, May 1995.

MICA: A Holistic Approach to Fast In-Memory Key-Value Storage

Hyeontaek Lim,[1] Dongsu Han,[2] David G. Andersen,[1] Michael Kaminsky[3]
[1]*Carnegie Mellon University,* [2]*KAIST,* [3]*Intel Labs*

Abstract

MICA is a scalable in-memory key-value store that handles 65.6 to 76.9 million key-value operations per second using a single general-purpose multi-core system. MICA is over 4–13.5x faster than current state-of-the-art systems, while providing consistently high throughput over a variety of mixed read and write workloads.

MICA takes a holistic approach that encompasses all aspects of request handling, including parallel data access, network request handling, and data structure design, but makes unconventional choices in each of the three domains. First, MICA optimizes for multi-core architectures by enabling parallel access to partitioned data. Second, for efficient parallel data access, MICA maps client requests directly to specific CPU cores at the server NIC level by using client-supplied information and adopts a light-weight networking stack that bypasses the kernel. Finally, MICA's new data structures—circular logs, lossy concurrent hash indexes, and bulk chaining—handle both read- and write-intensive workloads at low overhead.

1 Introduction

In-memory key-value storage is a crucial building block for many systems, including popular social networking sites (e.g., Facebook) [36]. These storage systems must provide high performance when serving many small objects, whose total volume can grow to TBs and more [5].

While much prior work focuses on high performance for read-mostly workloads [15, 30, 32, 37], in-memory key-value storage today must also handle write-intensive workloads, e.g., to store frequently-changing objects [2, 5, 36]. Systems optimized only for reads often waste resources when faced with significant write traffic; their inefficiencies include lock contention [32], expensive updates to data structures [15, 30], and complex memory management [15, 32, 36].

In-memory key-value storage also requires low-overhead network communication between clients and servers. Key-value workloads often include a large number of small key-value items [5] that require key-value storage to handle short messages efficiently. Systems using standard socket I/O, optimized for bulk communication, incur high network stack overhead at both kernel- and user-level. Current systems attempt to batch requests at the client to amortize this overhead, but batching increases latency, and large batches are unrealistic in large cluster key-value stores because it is more difficult to accumulate multiple requests being sent to the same server from a single client [36].

MICA (Memory-store with Intelligent Concurrent Access) is an in-memory key-value store that achieves high throughput across a wide range of workloads. MICA can provide either store semantics (no existing items can be removed without an explicit client request) or cache semantics (existing items may be removed to reclaim space for new items). Under write-intensive workloads with a skewed key popularity, a single MICA node serves 70.4 million small key-value items per second (Mops), which is 10.8x faster than the next fastest system. For skewed, read-intensive workloads, MICA's 65.6 Mops is at least 4x faster than other systems even after modifying them to use our kernel bypass. MICA achieves 75.5–76.9 Mops under workloads with a uniform key popularity. MICA achieves this through the following techniques:

Fast and scalable parallel data access: MICA's data access is fast and scalable, using data partitioning and exploiting CPU parallelism within and between cores. Its EREW mode (Exclusive Read Exclusive Write) minimizes costly inter-core communication, and its CREW mode (Concurrent Read Exclusive Write) allows multiple cores to serve popular data. MICA's techniques achieve consistently high throughput even under skewed workloads, one weakness of prior partitioned stores.

Network stack for efficient request processing: MICA interfaces with NICs directly, bypassing the kernel, and uses client software and server hardware to direct remote key-value requests to appropriate cores where the requests can be processed most efficiently. The network stack achieves zero-copy packet I/O and request processing.

New data structures for key-value storage: New memory allocation and indexing in MICA, optimized for store and cache separately, exploit properties of key-value workloads to accelerate write performance with simplified memory management.

2 System Goals

In this section, we first clarify the non-goals and then discuss the goals of MICA.

Non-Goals: We do not change the *cluster* architecture. It can still shard data and balance load across nodes, and perform replication and failure recovery.

We do not aim to handle large items that span multiple packets. Most key-value items will fit comfortably in a single packet [5]. Clients can store a large item in a traditional key-value system and put a pointer to that system in MICA. This only marginally increases total latency; one extra round-trip time for indirection is smaller than the transfer time of a large item sending multiple packets.

We do not strive for durability: All data is stored in DRAM. If needed, log-based mechanisms such as those from RAMCloud [37] would be needed to allow data to persist across power failures or reboots.

MICA instead strives to achieve the following goals:

High single-node throughput: Sites such as Facebook replicate some key-value nodes purely to handle load [36]. Faster nodes may reduce cost by requiring fewer of them overall, reducing the cost and overhead of replication and invalidation. High-speed nodes are also more able to handle load spikes and popularity hot spots. Importantly, using fewer nodes can also reduce job latency by reducing the number of servers touched by client requests. A single user request can create more than 500 key-value requests [36], and when these requests go to many nodes, the time until all replies arrive increases, delaying completion of the user request [10]. Having fewer nodes reduces fan-out, and thus, can improve job completion time.

Low end-to-end latency: The end-to-end latency of a remote key-value request greatly affects performance when a client must send back-to-back requests (e.g., when subsequent requests are dependent). The system should minimize both local key-value processing latency and the number of round-trips between the client and server.

Consistent performance across workloads: Real workloads often have a Zipf-distributed key popularity [5], and it is crucial to provide fast key-value operations regardless of skew. Recent uses of in-memory key-value storage also demand fast processing for write-intensive workloads [2, 36].

Handle small, variable-length key-value items: Most key-value items are small [5]. Thus, it is important to process requests for them efficiently. Ideally, key-value request processing over the network should be as fast as packet processing in software routers—40 to 80 Gbps [12, 19]. Variable-length items require careful memory management to reduce fragmentation that can waste substantial space [5].

Key-value storage interface and semantics: The system must support standard single-key requests (e.g., `GET(key)`, `PUT(key,value)`, `DELETE(key)`) that are common in systems such as Memcached. In cache mode, the system performs automatic cache management that may evict stored items at its discretion (e.g., LRU); in store mode, the system must not remove any stored items without clients' permission while striving to achieve good memory utilization.

Commodity hardware: Using general-purpose hardware reduces the cost of development, equipment, and operation. Today's server hardware can provide high-speed I/O [12, 22], comparable to that of specialized hardware such as FPGAs and RDMA-enabled NICs.

Although recent studies tried to achieve some of these goals, none of their solutions comprehensively address them. Some systems achieve high throughput by supporting only small fixed-length keys [33]. Many rely on client-based request batching [15, 30, 33, 36] to amortize high network I/O overhead, which is less effective in a large installation of key-value stores [14]; use specialized hardware, often with multiple client-server round-trips and/or no support for item eviction (e.g., FPGAs [7, 29], RDMA-enabled NICs [35]); or do not specifically address remote request processing [45]. Many focus on uniform and/or read-intensive workloads; several systems lack evaluation for skewed workloads [7, 33, 35], and some systems have lower throughput for write-intensive workloads than read-intensive workloads [30]. Several systems attempt to handle memory fragmentation explicitly [36], but there are scenarios where the system never reclaims fragmented free memory, as we describe in the next section. The fast packet processing achieved by software routers and low-overhead network stacks [12, 19, 20, 41, 43] set a bar for how fast a key-value system *might* operate on general-purpose hardware, but do not teach how their techniques apply to the higher-level processing of key-value requests.

3 Key Design Choices

Achieving our goals requires rethinking how we design *parallel data access*, the *network stack*, and *key-value data structures*. We make an unconventional choice for each; we discuss how we overcome its potential drawbacks to achieve our goals. Figure 1 depicts how these components fit together.

3.1 Parallel Data Access

Exploiting the parallelism of modern multi-core systems is crucial for high performance. The most common access models are concurrent access and exclusive access:

Concurrent access is used by most key-value systems [15, 30, 36]. As in Figure 2 (a), multiple CPU cores can access the shared data. The integrity of the data structure must be maintained using mutexes [36], optimistic locking [15, 30], or lock-free data structures [34].

Unfortunately, concurrent writes scale poorly: they incur frequent cache line transfer between cores, because only one core can hold the cache line of the same memory location for writing at the same time.

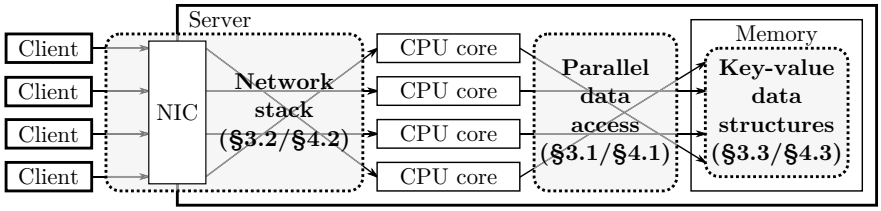
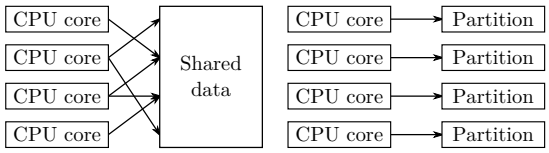


Figure 1: Components of in-memory key-value stores. MICA's key design choices in §3 and their details in §4.



(a) Concurrent access (b) Exclusive access

Figure 2: Parallel data access models.

Exclusive access has been explored less often for key-value storage [6, 25, 33]. Only one core can access part of the data, as in Figure 2 (b). By partitioning the data ("sharding"), each core exclusively accesses its own partition in parallel without inter-core communication.

Prior work observed that partitioning *can* have the best throughput and scalability [30, 45], but cautions that it lowers performance when the load between partitions is imbalanced, as happens under skewed key popularity [15, 30, 45]. Furthermore, because each core can access only data within its own partition, *request direction* is needed to forward requests to the appropriate CPU core.

MICA's parallel data access: MICA partitions data and mainly uses exclusive access to the partitions. MICA exploits CPU caches and packet burst I/O to disproportionately speed more loaded partitions, nearly eliminating the penalty from skewed workloads. MICA can fall back to concurrent reads if the load is extremely skewed, but avoids concurrent writes, which are always slower than exclusive writes. Section 4.1 describes our data access models and partitioning scheme.

3.2 Network Stack

This section discusses how MICA avoids network stack overhead and directs packets to individual cores.

3.2.1 Network I/O

Network I/O is one of the most expensive processing steps for in-memory key-value storage. TCP processing alone may consume 70% of CPU time on a many-core optimized key-value store [33].

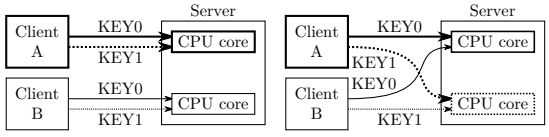
The **socket I/O** used by most in-memory key-value stores [15, 30, 33, 45] provides portability and ease of development. However, it underperforms in packets per second because it has high per-`read()` overhead. Many systems therefore often have clients include a batch of requests in a single larger packet to amortize I/O overhead.

Direct NIC access is common in software routers to achieve line-rate packet processing [12, 19]. This raw access to NIC hardware bypasses the kernel to minimize the packet I/O overhead. It delivers packets in bursts to efficiently use CPU cycles and the PCIe bus connecting NICs and CPUs. Direct access, however, precludes useful TCP features such as retransmission, flow control, and congestion control.

MICA's network I/O uses direct NIC access. By targeting only small key-value items, it needs fewer transport-layer features. Clients are responsible for retransmitting packets if needed. Section 4.2 describes such issues and our design in more detail.

3.2.2 Request Direction

Request direction delivers client requests to CPU cores for processing.[1] Modern NICs can deliver packets to specific cores for load balancing or core affinity using hardware-based packet classification and multi-queue support.



(a) Flow-level core affinity (b) Object-level core affinity

Figure 3: Request direction mechanisms.

Flow-level core affinity is available using two methods: Receive-Side Scaling (RSS) [12, 19] sends packets to cores based by hashing the packet header 5-tuple to identify which RX queue to target. Flow Director (FDir) [41] can more flexibly use different parts of the packet header plus a user-supplied table to map header values to RX queues. Efficient network stacks use affinity to reduce inter-core contention for TCP control blocks [20, 41].

Flow affinity reduces only *transport layer* contention, not *application-level* contention [20], because a single transport flow can contain requests for any objects (Figure 3 (a)). Even for datagrams, the benefit of flow affinity is small due to a lack of locality across datagrams [36].

Object-level core affinity distributes requests to cores based upon the application's partitioning. For example, requests sharing the same key would all go to the core handling that key's partition (Figure 3 (b)).

[1] Because we target small key-value requests, we will use requests and packets interchangeably.

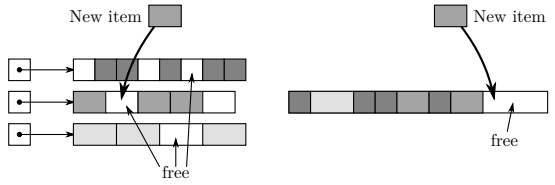
Systems using exclusive access require object-level core affinity, but commodity NIC hardware cannot directly parse and understand application-level semantics. Software request redirection (e.g., message passing [33]) incurs inter-core communication, which the exclusive access model is designed to avoid.

MICA's request direction uses Flow Director [23, 31]. Its *clients* then encode object-level affinity information in a way Flow Director can understand. Servers, in turn, inform clients about the object-to-partition mapping. Section 4.2 describes how this mechanism works.

3.3 Key-Value Data Structures

This section describes MICA's choice for two main data structures: allocators that manage memory space for storing key-value items and indexes to find items quickly.

3.3.1 Memory Allocator



(a) Dynamic object allocator (b) Append-only log structure

Figure 4: Memory allocators.

A **dynamic object allocator** is a common choice for storing variable-length key-value items (Figure 4 (a)). Systems such as Memcached typically use a slab approach: they divide object sizes into classes (e.g., 48-byte, 56-byte, ..., 1-MiB[2]) and maintain separate ("segregated") memory pools for these classes [15, 36]. Because the amount of space that each class uses typically varies over time, the systems use a global memory manager that allocates large memory blocks (e.g., 1 MiB) to the pools and dynamically rebalances allocations between classes.

The major challenge for dynamic allocation is the memory fragmentation caused when blocks are not fully filled. There may be no free blocks or free objects for some size classes while blocks from other classes are partly empty after deletions. Defragmentation packs objects of each object tightly to make free blocks, which involves expensive memory copy. This process is even more complex if the memory manager performs rebalancing concurrently with threads accessing the memory for other reads and writes.

Append-only log structures are write-friendly, placing new data items at the end of a linear data structure called a "log" (Figure 4 (b)). To update an item, it simply inserts a new item to the log that overrides the previous value. Inserts and updates thus access memory sequentially, incurring fewer cache and TLB misses, making logs particularly suited for bulk data writes. This approach is common in flash memory stores due to the high cost of random flash writes [3, 4, 28], but has been used in only a few in-memory key-value systems [37].

Garbage collection is crucial to space efficiency. It reclaims space occupied by overwritten and deleted objects by moving live objects to a new log and removing the old log. Unfortunately, garbage collection is costly and often reduces performance because of the large amount of data it must copy, trading memory efficiency against request processing speed.

MICA's memory allocator: MICA uses separate memory allocators for cache and store semantics. Its cache mode uses a log structure with inexpensive garbage collection and in-place update support (Section 4.3.1). MICA's allocator provides fast inserts and updates, and exploits cache semantics to eliminate log garbage collection and drastically simplify free space defragmentation. Its store mode uses segregated fits [42, 47] that share the unified memory space to avoid rebalancing size classes (Section 4.3.3).

3.3.2 Indexing: Read-oriented vs. Write-friendly

Read-oriented index: Common choices for indexing are hash tables [15, 33, 36] or tree-like structures [30]. However, conventional data structures are much slower for writes compared to reads; hash tables examine many slots to find a space for the new item [15], and trees may require multiple operations to maintain structural invariants [30].

Write-friendly index: Hash tables using *chaining* [33, 36] can insert new items without accessing many memory locations, but they suffer a time-space tradeoff: by having long chains (few hash buckets), an item lookup must follow a long chain of items, this requiring multiple random dependent memory accesses; when chains are short (many hash buckets), memory overhead to store chaining pointers increases. *Lossy data structures* are rather unusual in in-memory key-value storage and studied only in limited contexts [7], but it is the standard design in hardware indexes such as CPU caches [21].

MICA's index: MICA uses new index data structures to offer both high-speed read and write. In cache mode, MICA's lossy index also leverages the cache semantics to achieve high insertion speed; it evicts an old item in the hash table when a hash collision occurs instead of spending system resources to resolve the collision. By using the memory allocator's eviction support, the MICA lossy index can avoid evicting recently-used items (Section 4.3.2). The MICA lossless index uses *bulk chaining*, which allocates cache line-aligned space to a bucket for each chain segment. This keeps the chain length short and space efficiency high (Section 4.3.3).

[2]Binary prefixes (powers of 2) end with an "i" suffix, whereas SI prefixes (powers of 10) have no "i" suffix.

4 MICA Design

This section describes each component in MICA and discusses how they operate together to achieve its goals.

4.1 Parallel Data Access

This section explains how CPU cores access data in MICA, but assumes that cores process only the requests for which they are responsible. Later in Section 4.2, we discuss how MICA assigns remote requests to CPU cores.

4.1.1 Keyhash-Based Partitioning

MICA creates one or more partitions per CPU core and stores key-value items in a partition determined by their key. Such horizontal partitioning is often used to shard *across* nodes [4, 11], but some key-value storage systems also use it across cores within a node [6, 25, 33].

MICA uses a *keyhash* to determine each item's partition. A keyhash is the 64-bit hash of an item's key calculated by the client and used throughout key-value processing in MICA. MICA uses the first few high order bits of the keyhash to obtain the partition index for the item.

Keyhash partitioning uniformly maps keys to partitions, reducing the request distribution imbalance. For example, in a Zipf-distributed population of size 192×2^{20} (192 Mi) with skewness 0.99 as used by YCSB [9],[3] the most popular key is 9.3×10^6 times more frequently accessed than the average; after partitioning keys into 16 partitions, however, the most popular partition is only 53% more frequently requested than the average.

MICA retains high throughput under this remaining partition-level skew because it can process requests in "hot" partitions more efficiently, for two reasons. First, a partition is popular *because* it contains "hot" items; these hot items naturally create locality in data access. With high locality, MICA experiences fewer CPU cache misses when accessing items. Second, the skew causes packet I/O to be more efficient for popular partitions (described in Section 4.2.1). As a result, throughput for the Zipf-distributed workload is 86% of the uniformly-distributed workload, making MICA's partitioned design practical even under skewed workloads.

4.1.2 Operation Modes

MICA can operate in EREW (Exclusive Read Exclusive Write) or CREW (Concurrent Read Exclusive Write). *EREW* assigns a single CPU core to each partition for all operations. No concurrent access to partitions eliminates synchronization and inter-core communication, making MICA scale linearly with CPU cores. *CREW* allows any core to read partitions, but only a single core can write. This combines the benefit of concurrent read and exclusive write; the former allows all cores to process read requests, while the latter still reduces expensive cache line transfer. CREW handles reads efficiently under highly skewed load, at the cost of managing read-write conflicts. MICA minimizes the synchronization cost with efficient optimistic locking [48] (Section 4.3.2).

Supporting cache semantics in CREW, however, raises a challenge for read (`GET`) requests: During a `GET`, the cache may need to update cache management information. For example, policies such as LRU use bookkeeping to remember recently used items, which can cause conflicts and cache-line bouncing among cores. This, in turn, defeats the purpose of using exclusive writes.

To address this problem, we choose an approximate approach: MICA counts reads only from the exclusive-write core. Clients round-robin CREW reads across all cores in a NUMA domain, so this is effectively a sampling-based approximation to, e.g., LRU replacement as used in MICA's item eviction support (Section 4.3.1).

To show performance benefits of EREW and CREW, our MICA prototype also provides the CRCW (Concurrent Read Concurrent Write) mode, in which MICA allows multiple cores to read and write any partition. This effectively models concurrent access to the shared data in non-partitioned key-value systems.

4.2 Network Stack

The network stack in MICA provides *network I/O* to transfer packet data between NICs and the server software, and *request direction* to route requests to an appropriate CPU core to make subsequent key-value processing efficient.

Exploiting the small key-value items that MICA targets, request and response packets use UDP. Despite clients not benefiting from TCP's packet loss recovery and flow/congestion control, UDP has been used widely for read requests (e.g., `GET`) in large-scale deployments of in-memory key-value storage systems [36] for low latency and low overhead. Our protocol includes sequence numbers in packets, and our application relies on the idempotency of `GET` and `PUT` operations for simple and stateless application-driven loss recovery, if needed: some queries may not be useful past a deadline, and in many cases, the network is provisioned well, making retransmission rare and congestion control less crucial [36].

4.2.1 Direct NIC Access

MICA uses Intel's DPDK [22] instead of standard socket I/O. This allows our user-level server software to control NICs and transfer packet data with minimal overhead. MICA differs from general network processing [12, 19, 41] that has used direct NIC access in that MICA is an application that processes high-level key-value requests.

In NUMA (non-uniform memory access) systems with multiple CPUs, NICs may have different affinities to CPUs. For example, our evaluation hardware has two

[3] i-th key constitutes $1/(i^{0.99} H_{n,0.99})$ of total requests, where $H_{n,0.99} = \sum_{i=1}^{n} (1/i^{0.99})$ and n is the total number of keys.

CPUs, each connected to two NICs via a direct PCIe bus. MICA uses NUMA-aware memory allocation so that each CPU and NIC only accesses packet buffers stored in their respective NUMA domains.

MICA uses NIC multi-queue support to allocate a dedicated RX and TX queue to each core. Cores exclusively access their own queues without synchronization in a similar way to EREW data access. By directing a packet to an RX queue, the packet can be processed by a specific core, as we discuss in Section 4.2.2.

Burst packet I/O: MICA uses the DPDK's burst packet I/O to transfer multiple packets (up to 32 in our implementation) each time it requests packets from RX queues or transmits them to TX queues. Burst I/O reduces the per-packet cost of accessing and modifying the queue, while adding only trivial delay to request processing because the burst size is small compared to the packet processing rate.

Importantly, burst I/O helps handle skewed workloads. A core processing popular partitions spends more time processing requests, and therefore performs packet I/O less frequently. The lower I/O frequency increases the burst size, reducing the per-packet I/O cost (Section 5.2). Therefore, popular partitions have more CPU available for key-value processing. An unpopular partition's core has higher per-packet I/O cost, but handles fewer requests.

Zero-copy processing: MICA avoids packet data copy throughout RX/TX and request processing. MICA uses MTU-sized packet buffers for RX even if incoming requests are small. Upon receiving a request, MICA avoids memory allocation and copying by reusing the request packet to construct a response: it flips the source and destination addresses and ports in the header and updates only the part of the packet payload that differs between the request and response.

4.2.2 Client-Assisted Hardware Request Direction

Modern NICs help scale packet processing by directing packets to different RX queues using hardware features such as Receiver-Side Scaling (RSS) and Flow Director (FDir) [12, 19, 41] based on the packet header.

Because each MICA key-value request is an individual packet, we wish to use *hardware* packet direction to directly send packets to the appropriate queue based upon the key. Doing so is much more efficient than redirecting packets in software. Unfortunately, the NIC alone cannot provide key-based request direction: RSS and FDir cannot classify based on the packet payload, and cannot examine variable length fields such as request keys.

Client assistance: We instead take advantage of the opportunity to co-design the client and server. The client caches information from a server directory about the operation mode (EREW or CREW), number of cores, NUMA domains, and NICs, and number of partitions.

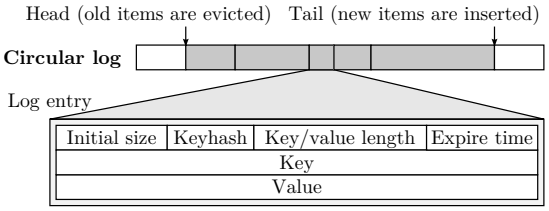


Figure 5: Design of a circular log.

The client then embeds the request direction information in the packet header: If the request uses exclusive data access (read/write on EREW and write on CREW), the client calculates the partition index from the keyhash of the request. If the request can be handled by any core (a CREW read), it picks a server core index in a round-robin way (across requests, but in the same NUMA domain (Section 4.2.1)). Finally, the client encodes the partition or core index as the UDP destination port.[4] The server programs FDir to use the UDP destination port, without hashing, ("perfect match filter" [23]), as an index into a table mapping UDP port numbers to a destination RX queue. Key hashing only slightly burdens clients. Using fast string hash functions such as CityHash [8], a single client machine equipped with dual 6-core CPUs on our testbed can generate over 40 M requests/second with client-side key hashing. Clients include the keyhash in requests, and servers reuse the embedded keyhash when they need a keyhash during the request processing to benefit from offloaded hash computation.

Client-assisted request direction using NIC hardware allows efficient request processing. Our results in Section 5.5 show that an optimized software-based request direction that receives packets from any core and distributes them to appropriate cores is significantly slower than MICA's hardware-based approach.

4.3 Data Structure

MICA, in cache mode, uses *circular logs* to manage memory for key-value items and *lossy concurrent hash indexes* to index the stored items. Both data structures exploit cache semantics to provide fast writes and simple memory management. Each MICA partition consists of a single circular log and lossy concurrent hash index.

MICA provides a store mode with straightforward extensions using segregated fits to allocate memory for key-value items and *bulk chaining* to convert the lossy concurrent hash indexes into lossless ones.

4.3.1 Circular Log

MICA stores items in its circular log by appending them to the *tail* of the log (Figure 5). This results in a space-efficient packing. It updates items in-place as long as the

[4] To avoid confusion between partition indices and the core indices, we use different ranges of UDP ports; a partition may be mapped to a core whose index differs from the partition index.

new size of the key+value does not exceed the size of the item when it was first inserted. The size of the circular log is bounded and does not change, so to add a new item to a full log, MICA evicts the oldest item(s) at the *head* of the log to make space.

Each entry includes the key and value length, key, and value. To locate the next item in the log and support item resizing, the entry contains the initial item size, and for fast lookup, it stores the keyhash of the item. The entry has an expire time set by the client to ignore stale data.

Garbage collection and defragmentation: The circular log eliminates the expensive garbage collection and free space defragmentation that are required in conventional log structures and dynamic memory allocators. Previously deleted items in the log are automatically collected and removed when new items enter the log. Almost all free space remains contiguously between the tail and head.

Exploiting the eviction of live items: Items evicted at the head are not reinserted to the log even if they have not yet expired. In other words, the log may delete items without clients knowing it. This behavior is valid in cache workloads; a key-value store must evict items when it becomes full. For example, Memcached [32] uses LRU to remove items and reserve space for new items.

MICA uses this item eviction to implement common eviction schemes at low cost. Its "natural" eviction is FIFO. MICA can provide LRU by reinserting any requested items at the tail because only the least recently used items are evicted at the head. MICA can approximate LRU by reinserting requested items selectively—by ignoring items recently (re)inserted and close to the tail; this approximation offers eviction similar to LRU without frequent reinserts, because recently accessed items remain close to the tail and far from the head.

A second challenge for conventional logs is that any reference to an evicted item becomes dangling. MICA does not store back pointers in the log entry to discover all references to the entry; instead, it provides detection, and removes dangling pointers incrementally (Section 4.3.2).

Low-level memory management: MICA uses hugepages and NUMA-aware allocation. Hugepages (2 MiB in x86-64) use fewer TLB entries for the same amount of memory, which significantly reduces TLB misses during request processing. Like the network stack, MICA allocates memory for circular logs such that cores access only local memory.

Without explicit range checking, accessing an entry near the end of the log (e.g., at $2^{34}-8$ in the example below) could cause an invalid read or segmentation fault by reading off the end of the range. To avoid such errors without range checking, MICA manually maps the virtual memory addresses right after the end of the log to the same physical page as the first page of the log, making the entire log appear locally contiguous:

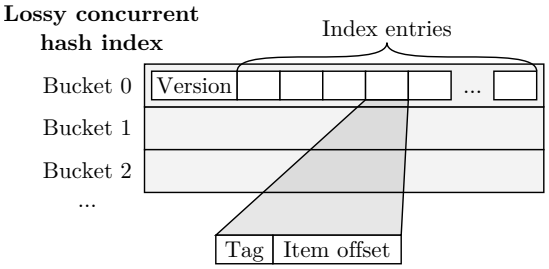
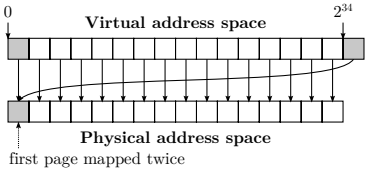


Figure 6: Design of a lossy concurrent hash index.



Our MICA prototype implements this scheme in userspace by mapping a pool of hugepages to virtual addresses using the `mmap()` system call.

4.3.2 Lossy Concurrent Hash Index

MICA's hash index locates key-value items in the log using a set-associative cache similar to that used in CPU caches. As shown in Figure 6, a hash index consists of multiple buckets (configurable for the workload), and each bucket has a fixed number of index entries (configurable in the source code; 15 in our prototype to occupy exactly two cache lines). MICA uses a portion of the keyhashes to determine an item's bucket; the item can occupy any index entry of the bucket unless there is a duplicate.

Each index entry contains partial information for the item: a tag and the item offset within the log. A tag is another portion of the indexed item's keyhash used for filtering lookup keys that do not match: it can tell whether the indexed item will never match against the lookup key by comparing the stored tag and the tag from the lookup keyhash. We avoid using a zero tag value by making it one because we use the zero value to indicate an empty index entry. Items are deleted by writing zero values to the index entry; the entry in the log will be automatically garbage collected.

Note that the parts of keyhashes used for the partition index, the bucket number, and the tag do not overlap. Our prototype uses 64-bit keyhashes to provide sufficient bits.

Lossiness: The hash index is lossy. When indexing a new key-value item into a full bucket of the hash index, the index evicts an index entry to accommodate the new item. The item evicted is determined by its age; if the item offset is most behind the tail of the log, the item is the oldest (or least recently used if the log is using LRU), and the associated index entry of the item is reclaimed.

This lossy property allows fast insertion. It avoids expensive resolution of hash collisions that lossless indexes of other key-value stores require [15, 33]. As a result,

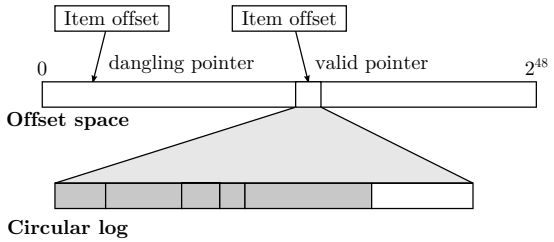


Figure 7: Offset space for dangling pointer detection.

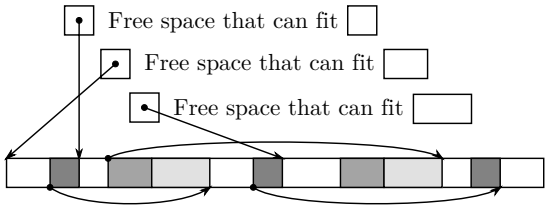


Figure 8: Segregated free lists for a unified space.

MICA's insert speed is comparable to lookup speed.

Handling dangling pointers: When an item is evicted from the log, MICA does not delete its index entry. Although it is possible to store back pointers in the log entry, updating the hash index requires a random memory write and is complicated due to locking if the index is being accessed concurrently, so MICA does not. As a result, index pointers can "dangle," pointing to invalid entries.

To address this problem, MICA uses large pointers for head/tail and item offsets. As depicted in Figure 7, MICA's index stores log offsets that are wider than needed to address the full size of the log (e.g., 48-bit offsets vs 34 bits for a 16 GiB log). MICA detects a dangling pointer before using it by checking if the difference between the log tail and the item offset is larger than the actual log size.[5] If the tail wraps around the 48-bit size, however, a dangling pointer may appear valid again, so MICA scans the index incrementally to remove stale pointers.

This scanning must merely complete a full cycle before the tail wraps around in its wide offset space. The speed at which it wraps is determined by the increment rate of the tail and the width of the item offset. In practice, full scanning is infrequent even if writes occur very frequently. For example, with 48-bit offsets and writes occurring at 2^{30} bytes/second (millions of operations/second), the tail wraps every 2^{48-30} seconds. If the index has 2^{24} buckets, MICA must scan only 2^6 buckets per second, which adds negligible overhead.

Supporting concurrent access: MICA's hash index must behave correctly if the system permits concurrent operations (e.g., CREW). For this, each bucket contains a 32-bit version number. It performs reads optimistically using this version counter to avoid generating memory writes while satisfying `GET` requests [15, 30, 48]. When accessing an item, MICA checks if the initial state of the version number of the bucket is even-numbered, and upon completion of data fetch from the index and log, it reads the version number again to check if the final version number is equal to the initial version number. If either check fails, it repeats the read request processing from the beginning. For writes, MICA increments the version number by one before beginning, and increments the version number by one again after finishing all writes. In CRCW mode, which allows multiple writers to access the same bucket, a writer also spins until the initial version number is even (i.e., no other writers to this bucket) using a compare-swap operation instruction.

Our MICA prototype uses different code to optimize locking. It uses conventional instructions to manipulate version numbers to exploit memory access ordering on the x86 architecture [48] in CREW mode where there is only one writer. EREW mode does not require synchronization between cores, so MICA ignores version numbers. Because of such a hard-coded optimization, the current prototype lacks support for runtime switching between the operation modes.

Multi-stage prefetching: To retrieve or update an item, MICA must perform request parsing, hash index lookup, and log entry retrieval. These stages cause random memory access that can significantly lower system performance if cores stall due to CPU cache and TLB misses.

MICA uses multi-stage prefetching to interleave computation and memory access. MICA applies memory prefetching for random memory access done at each processing stage in sequence. For example, when a burst of 8 RX packets arrives, MICA fetches packets 0 and 1 and *prefetches* packets 2 and 3. It decodes the requests in packets 0 and 1, and prefetches buckets of the hash index that these requests will access. MICA continues packet payload prefetching for packets 4 and 5. It then prefetches log entries that may be accessed by the requests of packets 0 and 1 while prefetching the hash index buckets for packets 2 and 3, and the payload of packet 6 and 7. MICA continues this pipeline until all requests are processed.

4.3.3 Store Mode

The store mode of MICA uses segregated fits [42, 47] similar to fast malloc implementations [27], instead of the circular log. Figure 8 depicts this approach. MICA defines multiple size classes incrementing by 8 bytes covering all supported item sizes, and maintains a freelist for each size class (a linked list of pointers referencing unoccupied memory regions that are at least as large as the size class). When a new item is inserted, MICA chooses the smallest size class that is at least as large as the item size and has any free space. It stores the item in the free space, and inserts any unused region of the free space into a freelist that matches that region's size. When an item is deleted, MICA coalesces any adjacent free regions using boundary

[5] $(\text{Tail} - \text{ItemOffset} + 2^{48}) \bmod 2^{48} > \text{LogSize}$.

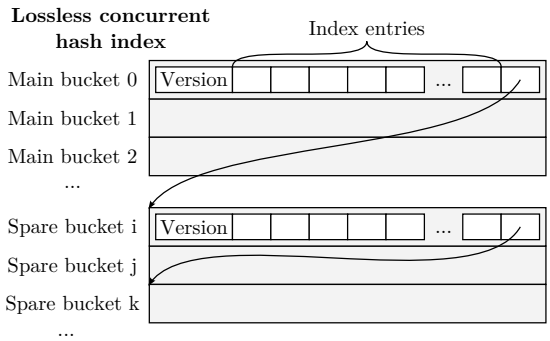


Figure 9: Bulk chaining in MICA's lossless hash index.

tags [26] to recreate a large free region.

MICA's segregated fits differ from the simple segregated storage used in Memcached [15, 32]. MICA maintains a unified space for all size classes; on the contrary, Memcached's SLAB allocator dynamically assigns memory blocks to size classes, which effectively partitions the memory space according to size classes. The unified space of MICA eliminates the need to rebalance size classes unlike the simple segregated storage. Using segregated fits also makes better use of memory because MICA already has partitioning done with keyhashes; a SLAB allocator introducing another partitioning would likely waste memory by allocating a whole block for only a few items, resulting in low memory occupancy.

MICA converts its lossy concurrent hash index into a lossless hash index by using *bulk chaining*. Bulk chaining is similar to the traditional chaining method in hash tables; it adds more memory space to the buckets that contain an excessive number of items.

Figure 9 shows the design of the lossless hash index. MICA uses the lossy concurrent hash index as the main buckets and allocates space for separate spare buckets that are fewer than the main buckets. When a bucket experiences an overflow, whether it is a main bucket or spare bucket, MICA adds an unused spare bucket to the full bucket to form a bucket chain. If there are no more spare buckets available, MICA rejects the new item and returns an out-of-space error to the client.

This data structure is friendly to memory access. The main buckets store most of items (about 95%), keeping the number of random memory read for an index lookup close to 1; as a comparison, cuckoo hashing [39] used in improved Memcached systems [15] would require 1.5 random memory accesses per index lookup in expectation. MICA also allows good memory efficiency; because the spare buckets only store overflow items, making the number of spare buckets 10% of the main buckets allows the system to store the entire dataset of 192 Mi items in our experiments (Section 5).

5 Evaluation

We answer four questions about MICA in this section:

- *Does it perform well under diverse workloads?*
- *Does it provide good latency?*
- *How does it scale with more cores and NIC ports?*
- *How does each component affect performance?*

Our results show that MICA has consistently high throughput and low latency under a variety of workloads. It scales nearly linearly, using CPU cores and NIC ports efficiently. Each component of MICA is needed. MICA achieves 65.6–76.9 million operations/second (Mops), which is over 4–13.5x faster than the next fastest system; the gap widens as the fraction of write requests increases.

MICA is written in 12 K lines of C and runs on x86-64 GNU/Linux. Packet I/O uses the Intel DPDK 1.4.1 [22].

Compared systems: We use custom versions of open-source Memcached [32], MemC3 [15], Masstree [30], and RAMCloud [37]. The revisions of the original code we used are: Memcached: 87e2f36; MemC3: an internal version; Masstree: 4ffb946; RAMCloud: a0f6889.

Note that the compared systems often offer additional capabilities compared to others. For example, Masstree can handle range queries, and RAMCloud offers low latency processing on InfiniBand; on the other hand, these key-value stores do not support automatic item eviction as Memcached systems do. Our evaluation focuses on the performance of the standard features (e.g., single key queries) common to all the compared systems, rather than highlighting the potential performance impact from these semantic differences.

Modifications to compared systems: We modify the compared systems to use our lightweight network stack to avoid using expensive socket I/O or special hardware (e.g., InfiniBand). When measuring Memcached's baseline latency, we use its original network stack using the kernel to obtain the latency distribution that typical Memcached deployments would experience. Our experiments do not use any client-side request batching. We also modified these systems to invoke memory allocation functions though our framework if they use hugepages, because the DPDK requests all hugepages from the OS at initialization and would make the unmodified systems inoperable if they request hugepages from the OS; we kept other memory allocations using no hugepages as-is. Finally, while running experiments, we found that statistics collection in RAMCloud caused lock contention, so we disabled it for better multi-core performance.

5.1 Evaluation Setup

Server/client configuration: MICA server runs on a machine equipped with dual 8-core CPUs (Intel Xeon E5-2680 @2.70 GHz), 64 GiB of total system memory, and eight 10-Gb Ethernet ports (four Intel X520-T2's). Each

CPU has 20 MiB of L3 cache. We disabled logical processor support ("Hyper-Threading"). Each CPU accesses the 32 GiB of the system memory that resides in its local NUMA domain over a quad-channel DDR3-1600 bus. Each CPU socket is directly connected to two NICs using PCIe gen2. Access to hardware resources in the remote NUMA domain uses an interconnect between two CPUs (Intel QuickPath).

We reserved the half of the memory (16 GiB in each NUMA domain) for hugepages regardless of how MICA and the compared systems use hugepages.

MICA allocates 16 partitions in the server, and these partitions are assigned to different cores. We configured the cache version of MICA to use approximate LRU to evict items; MICA reinserts any recently accessed item at the tail if the item is closer to the head than to the tail of the circular log.

Two client machines with dual 6-core CPUs (Intel Xeon L5640 @2.27 GHz) and two Intel X520-T2's generate workloads. The server and clients are directly connected without a switch. Each client is connected to the NICs from both NUMA domains of the server, allowing a client to send a request to any server CPU.

Workloads: We explore different aspects of the systems by varying the item size, skew, and read-write ratio.

We use three datasets as shown in the following table:

Dataset	Key Size (B)	Value Size (B)	Count
Tiny	8	8	192 Mi
Small	16	64	128 Mi
Large	128	1024	8 Mi

We use two workload types: *uniform* and *skewed*. Uniform workloads use the same key popularity for all requests; skewed workloads use a non-uniform key popularity that follows a Zipf distribution of skewness 0.99, which is the same as YCSB's [9].

Workloads have a *varied ratio between* `GET` *and* `PUT`. 50% `GET` (50% `PUT`) workloads are write-intensive, and 95% `GET` (5% `PUT`) workloads are read-intensive. They correspond to YCSB's A and B workloads, respectively.

Workload generation: We use our custom key-value request generator that uses similar techniques to our lightweight network stack to send more than 40 Mops of key-value requests per machine to saturate the link.[6] It uses approximation techniques of Zipf distribution generation [17, 38] for fast skewed workload generation.

To find the maximum *meaningful* throughput of a system, we adjust the workload generation rate to allow only marginal packet losses ($< 1\%$ at any NIC port). We could generate requests at the highest rate to cause best-effort request processing (which can boost measured throughput more than 10%), as is commonly done in throughput measurement of software routers [12, 19], but we avoid this method because we expect that real deployments of in-memory key-value stores would not tolerate excessive packet losses, and such flooding can distort the intended skew in the workload by causing biased packet losses at different cores.

The workload generator does not receive every response from the server. On our client machines, receiving packets whose size is not a multiple of 64 bytes is substantially slower due to an issue in the PCIe bus [18].

The workload generator works around this slow RX by sampling responses to perform fewer packet RX from NIC to CPU. It uses its real source MAC addresses for only a fraction of requests, causing its NIC to drop the responses to the other requests. By looking at the sampled responses, the workload generator can validate that the server has correctly processed the requests. Our server is unaffected from this issue and performs full packet RX.

5.2 System Throughput

We first compare the full-system throughput. MICA uses EREW with all 16 cores. However, we use a different number of cores for the other systems to obtain their best throughput because some of them (Memcached, MemC3, and RAMCloud) achieve higher throughput with fewer cores (Section 5.4). The throughput numbers are calculated from the actual number of responses sent to the clients after processing the requests at the server. We denote the cache version of MICA by MICA-c and the store version of MICA by MICA-s.

Figure 10 (top) plots the experiment result using *tiny key-value items*. MICA performs best, regardless of the skew or the `GET` ratio. MICA's throughput reaches 75.5–76.9 Mops for uniform workloads and 65.6–70.5 Mops for skewed ones; its parallel data access does not incur more than a 14% penalty for skewed workloads. MICA uses 54.9–66.4 Gbps of network bandwidth at this processing speed—this speed is very close to 66.6 Gbps that our network stack can handle when doing packet I/O only. The next best system is Masstree at 16.5 Mops, while others are below 6.1 Mops. All systems except MICA suffer noticeably under write-intensive 50% `GET`.

Small key-value items show similar results in Figure 10 (middle). However, the gap between MICA and the other systems shrinks because MICA becomes network bottlenecked while the other systems never saturate the network bandwidth in our experiments.

Large key-value items, shown in Figure 10 (bottom), exacerbates the network bandwidth bottleneck, further limiting MICA's throughput. MICA achieves 12.6–14.6 Mops for 50% `GET` and 8.6–9.4 Mops for 95% `GET`; note that MICA shows high throughput with lower `GET` ratios,

[6] MICA clients are still allowed to use standard socket I/O in cases where the socket overhead on the client machines is acceptable because the MICA server and clients use the plain UDP protocol.

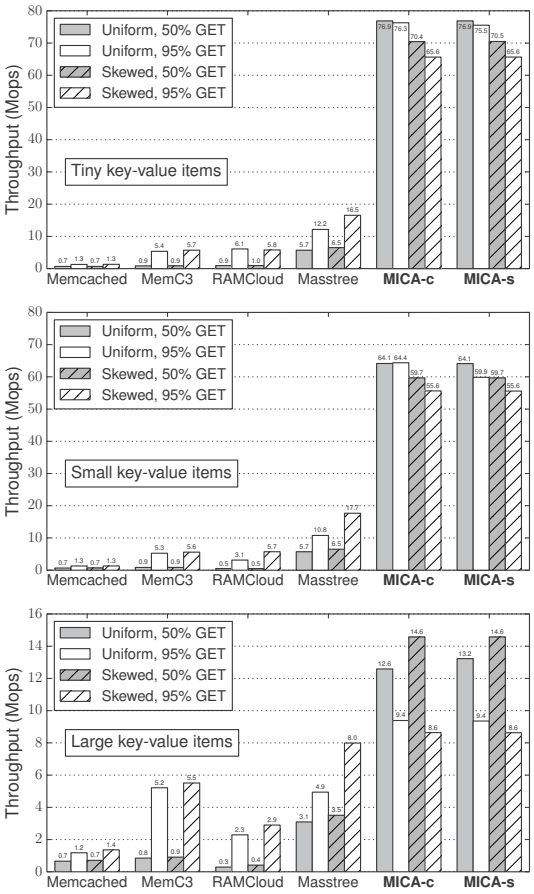


Figure 10: End-to-end throughput of in-memory key-value systems. All systems use our lightweight network stack that does not require request batching. The bottom graph (large key-value items) uses a different Y scale from the first two graphs'.

which require less network bandwidth as the server can omit the key and value from the responses. Unlike MICA, however, all other systems achieve higher throughput under 95% `GET` than under 50% `GET` because these systems are bottleneck locally, not by the network bandwidth.

In those measurements, MICA's cache and store modes show only minor differences in the performance. We will refer to the cache version of MICA as MICA in the rest of the evaluation for simplicity.

Skew resistance: Figure 11 compares the per-core throughput under uniform and skewed workloads of 50% `GET` with tiny items. MICA uses EREW. Several cores process more requests under the skewed workload than under the uniform workload because they process requests more efficiently. The skew in the workload increases the RX burst size of the most loaded core from 10.2 packets per I/O to 17.3 packets per I/O, reducing its per-packet I/O cost, and the higher data locality caused by the workload skew improves the average cache hit ratio of all cores from

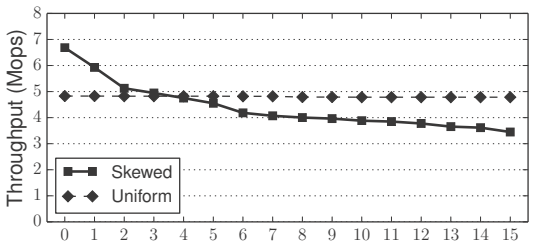


Figure 11: Per-core breakdown of end-to-end throughput.

Figure 12: Local throughput of key-value data structures.

67.8% to 77.8%. A local benchmark in Figure 12 (without network processing) also shows that skewed workloads grant good throughput for local key-value processing due to the data locality. These results further justify the partitioned design of MICA and explains why MICA retains high throughput under skewed workloads.

Summary: MICA's throughput reaches 76.9 Mops, at least 4x faster than the next best system. MICA delivers consistent performance across different skewness, write-intensiveness, and key-value sizes.

5.3 Latency

To show that MICA achieves comparably low latency while providing high throughput, we compare MICA's latency with that of the original Memcached implementation that uses the kernel network stack. To measure the end-to-end latency, clients tag each request packet with the current timestamp. When receiving responses, clients compare the current timestamp and the previous timestamp echoed back in the responses. We use uniform 50%

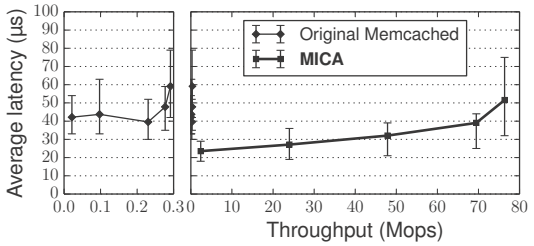


Figure 13: End-to-end latency of the original Memcached and MICA as a function of throughput.

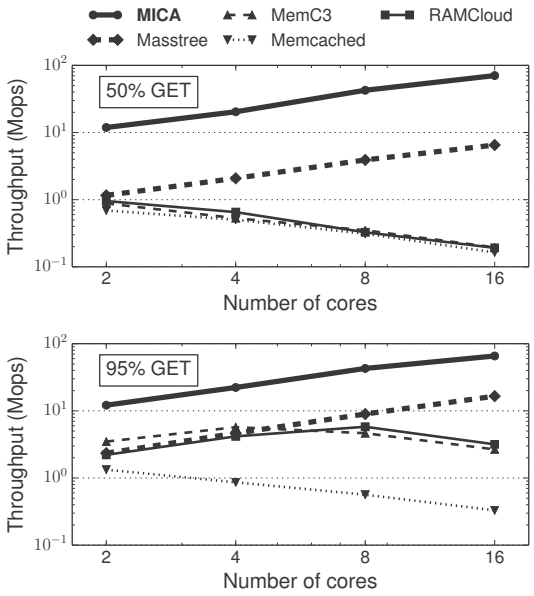


Figure 14: End-to-end throughput of in-memory key-value systems using a varying number of cores. All systems use our lightweight network stack.

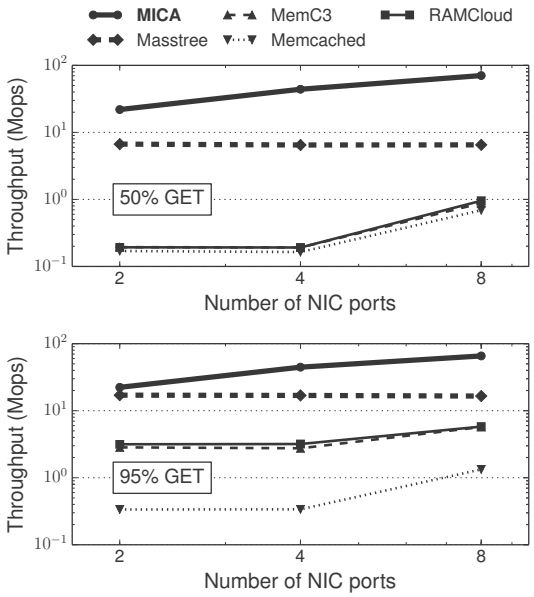


Figure 15: End-to-end throughput of in-memory key-value systems using a varying number of NIC ports. All systems use our lightweight network stack.

`GET` workloads on tiny items. MICA uses EREW. The client varies the request rate to observe the relationship between throughput and latency.

Figure 13 plots the end-to-end latency as a function of throughput; the error bars indicate 5th- and 95th-percentile latency. The original Memcached exhibits almost flat latency up to certain throughput, whereas MICA shows varied latency depending on the throughput it serves. MICA's latency lies between 24–52 µs. At the similar latency level of 40 µs, MICA shows 69 Mops—more than two orders of magnitude faster than Memcached.

Because MICA uses a single round-trip per request unlike RDMA-based systems [35], we believe that MICA provides best-in-class low-latency key-value operations.

Summary: MICA achieves both high throughput and latency near the network minimum.

5.4 Scalability

CPU scalability: We vary now the number of CPU cores and compare the end-to-end throughput. We allocate cores evenly to both NUMA domains so that cores can efficiently access NICs connected to their CPU socket. We use skewed workloads on tiny items because it is generally more difficult for partitioned stores to handle skewed workloads. MICA uses EREW.

Figure 14 (upper) compares core scalability of systems with 50% `GET`. Only MICA and Masstree perform better with more cores. Memcached, MemC3, and RAMCloud scale poorly, achieving their best throughput at 2 cores.

The trend continues for 95% `GET` requests in Figure 14 (lower); MICA and Masstree scale well as before. The rest also achieve higher throughput, but still do not scale. Note that some systems scale differently from their original papers. For example, MemC3 achieves 5.7 Mops at 4 cores, while the original paper shows 4.4 Mops at 16 cores [15]. This is because using our network stack instead of their network stack reduces I/O cost, which may expose a different bottleneck (e.g., key-value data structures) that can change the optimal number of cores for the best throughput.

Network scalability: We also change the available network bandwidth by varying the number of NIC ports we use for request processing. Figure 15 shows that MICA again scales well with high network bandwidth, because MICA can use almost all available network bandwidth for request processing. The `GET` ratio does not affect the result for MICA significantly. This result suggests that MICA can possibly scale further with higher network bandwidth (e.g., multiple 40 Gbps NICs). MICA and Masstree achieve similar performance under the 95% `GET` workload when using 2 ports, but Masstree and other systems do not scale well with more ports.

Summary: MICA scales well with more CPU cores and more network bandwidth, even under write-intensive workloads where other systems tend to scale worse.

5.5 Necessity of the Holistic Approach

In this section, we demonstrate how each component of MICA contributes to its performance. Because MICA is a coherent system that exploits the synergy between its

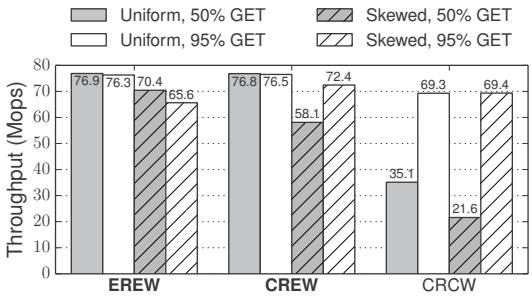


Figure 16: End-to-end performance using MICA's EREW, CREW, and CRCW.

Method	Workload	Throughput
Software-only	Uniform	33.9 Mops
	Skewed	28.1 Mops
Client-assisted hardware-based	Uniform	76.9 Mops
	Skewed	70.4 Mops

Table 1: End-to-end throughput of different request direction methods.

components, we compare different approaches for one component while keeping the other components the same.

Parallel data access: We use end-to-end experiments to measure how different data access modes affect the system performance. We use tiny items only. Figure 16 shows the end-to-end results. EREW shows consistently good performance. CREW achieves slightly higher throughput with high `GET` ratios on skewed workloads compared to EREW (white bars at 95% `GET`) because despite the overheads from bucket version management, CREW can use multiple cores to read popular items without incurring excessive inter-core communication. While CRCW performs better than any other compared systems (Section 5.2), CRCW offers no benefit over EREW and CREW; this suggests that we should avoid CRCW.

Network stack: As shown in Section 5.2, switching Masstree to our network stack resulted in much higher throughput (16.5 Mops without request batching) than the throughput from the original paper (8.9 Mops with request batching [30]); this indicates that our network stack provides efficient I/O for key-value processing.

The next question is how important it is to use hardware to direct requests for exclusive access in MICA. To compare with MICA's client-assisted hardware request direction, we implemented software-only request direction: clients send requests to any server core in a round-robin way, and the server cores direct the received requests to the appropriate cores for EREW data access. We use Intel DPDK's queue to implement message queues between cores. We use 50% `GET` on tiny items.

Table 1 shows that software request direction achieves only 40.0–44.1% of MICA's throughput. This is due to the inter-core communication overhead of software request direction. Thus, MICA's request direction is crucial for realizing the benefit of exclusive access.

Method	Workload	Throughput
Partitioned Masstree	50% `GET`	5.8 Mops
	95% `GET`	17.9 Mops
MICA	50% `GET`	70.4 Mops
	95% `GET`	65.6 Mops

Table 2: End-to-end throughput comparison between partitioned Masstree and MICA using skewed workloads.

Key-value data structures: MICA's circular logs, lossy concurrent hash indexes, and bulk chaining permit high-speed read and write operations with simple memory management. Even CRCW, the slowest data access mode of MICA, outperforms the second best system, Masstree (Section 5.2).

We also demonstrate that partitioning existing data structures does not simply grant MICA's high performance. For this, we compare MICA with "partitioned" Masstree, which uses one Masstree instance per core, with its support for concurrent access disabled in the source code. This is similar to MICA's EREW. We also use the same partitioning and request direction scheme.

Table 2 shows the result with skewed workloads on tiny items. Partitioned Masstree achieves only 8.2–27.3% of MICA's performance, with the throughput for 50% `GET` even lower than non-partitioned Masstree (Section 5.2). This indicates that to make best use of MICA's parallel data access and network stack, it is important to use key-value data structures that perform high-speed writes and to provide high efficiency with data partitioning.

In conclusion, the holistic approach is essential; any missing component significantly degrades performance.

6 Related Work

Most DRAM stores are not partitioned: Memcached [32], RAMCloud [37], MemC3 [15], Masstree [30], and Silo [45] all have a single partition for each server node. Masstree and Silo show that partitioning can be efficient under some workloads but is slow under workloads with a skewed key popularity and many cross-partition transactions. MICA exploits burst I/O and locality so that even in its exclusive EREW mode, loaded partitions run faster. It can do so because the simple key-value requests that it targets do not cross partitions.

Partitioned systems are fast with well-partitioned data. Memcached on Tilera [6], CPHash [33], and Chronos [25] are partitioned in-memory key-value systems that exclusively access partitioned hash tables to minimize lock contention and cache movement, similar to MICA's EREW partitions. These systems lack support for other partitioning such as MICA's CREW that can provide higher throughput under read-intensive skewed workloads.

H-Store [44] and VoltDB [46] use single-threaded execution engines that access their own partition exclusively, avoiding expensive concurrency control. Because workload skew can reduce system throughput, they require careful data partitioning, even using machine learning methods [40], and dynamic load balancing [25]. MICA achieves similar throughput under both uniform and skewed workloads without extensive partitioning and load balancing effort because MICA's keyhash-based partitioning mitigates the skew using and its request processing for popular partitions exploits burst packet I/O and cache-friendly memory access.

Several in-memory key-value systems focus on low latency request processing. RAMCloud achieves 4.9–15.3 µs end-to-end latency for small objects [1], and Chronos exhibits average latency of 10 µs and a 99th-percentile latency of 30 µs, on low latency networks such as InfiniBand and Myrinet. Pilaf [35] serves read requests using one-sided RDMA reads on a low-latency network. Our MICA prototype currently runs on 10-Gb Ethernet NIC whose base latency is much higher [16]; we plan to evaluate MICA on a low-latency network such.

Prior work studies providing a high performance reliable transport service using low-level unreliable datagram services. The Memcached UDP protocol relies on application-level packet loss recovery [36]. Low-overhead user-level implementations for TCP such as mTCP [24] can offer reliable communication to Memcached applications without incurring high performance penalties. Low-latency networks such as InfiniBand often implement hardware-level reliable datagrams [35].

Affinity-Accept [41] uses Flow Director on the commodity NIC hardware to load balance TCP connections across multiple CPU cores. Chronos [25] directs remote requests to server cores using client-supplied information, similar to MICA; however, Chronos uses software-based packet classification whose throughput for small key-value requests is significantly lower than MICA's hardware-based classification.

Strict or complex item eviction schemes in key-value stores can be so costly that it can reduce system throughput significantly. MemC3 [15] replaces Memcached [32]'s original LRU with a CLOCK-based approximation to avoid contention caused by LRU list management. MICA's circular log and lossy concurrent hash index use its lossy property to support common eviction schemes at low cost; the lossy concurrent hash index is easily extended to support lossless operations by using bulk chaining.

A worthwhile area of future work is applying MICA's techniques to semantically richer systems, such as those that are durable [37], or provide range queries [13, 30] or multi-key transactions [45]. Our results show that existing systems such as Masstree can benefit considerably simply by moving to a lightweight network stack; nevertheless, operations in these systems may cross partitions, it remains to be seen how to best harness the speed of exclusively accessed partitions.

7 Conclusion

MICA is an in-memory key-value store that provides high-performance, scalable key-value storage. It provides consistently high throughput and low latency for read/write-intensive workloads with a uniform/skewed key popularity. We demonstrate high-speed request processing with MICA's parallel data access to partitioned data, efficient network stack that delivers remote requests to appropriate CPU cores, and new lossy and lossless data structures that exploit properties of key-value workloads to provide high-speed write operations without complicating memory management.

Acknowledgments

This work was supported by funding from the National Science Foundation under awards CCF-0964474 and CNS-1040801, Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and Basic Science Research Program through the National Research Foundation of Korea funded by MSIP (NRF-2013R1A1A1076024). Hyeontaek Lim was supported in part by the Facebook Fellowship. We would like to thank Nick Feamster, John Ousterhout, Dong Zhou, Yandong Mao, Wyatt Lloyd, and our NSDI reviewers for their valuable feedback, and Prabal Dutta for shepherding this paper.

References

[1] Ramcloud project wiki: clusterperf November 12, 2012, 2012. https://ramcloud.stanford.edu/wiki/display/ramcloud/clusterperf+November+12%2C+2012.

[2] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of the fifth ACM international conference on Web search and data mining*, Feb. 2012.

[3] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. 7th USENIX NSDI*, Apr. 2010.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012.

[6] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store.

http://gigaom2.files.wordpress.com/2011/07/facebook-tilera-whitepaper.pdf, 2011.

[7] M. Blott, K. Karras, L. Liu, K. Vissers, J. Bär, and Z. István. Achieving 10Gbps line-rate key-value stores with FPGAs. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Cloud Computing*, June 2013.

[8] CityHash. http://code.google.com/p/cityhash/, 2014.

[9] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[10] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[11] 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 *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[12] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[13] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. In *Proc. ACM SIGCOMM*, Aug. 2012.

[14] Facebook's memcached multiget hole: More machines != more capacity. http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html, 2009.

[15] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. 10th USENIX NSDI*, Apr. 2013.

[16] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, June 2013.

[17] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, May 1994.

[18] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste. XIA: Efficient support for evolvable internetworking. In *Proc. 9th USENIX NSDI*, Apr. 2012.

[19] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, Aug. 2010.

[20] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, Oct. 2012.

[21] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12): 1612–1630, Dec. 1989.

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

[23] Intel 82599 10 Gigabit Ethernet Controller: Datasheet. http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html, 2014.

[24] E. 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. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[25] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, Oct. 2012.

[26] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997. First edition published in 1968.

[27] D. Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 2000.

[28] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[29] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with Smart Pipes: Designing SoC accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, June 2013.

[30] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. 7th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2012.

[31] Mellanox ConnectX-3 product brief. http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN_Card.pdf, 2013.

[32] A distributed memory object caching system. http://memcached.org/, 2014.

[33] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHash: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2012.

[34] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, July 2002.

[35] 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 2013 conference on USENIX Annual technical conference*, June 2013.

[36] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX NSDI*, Apr. 2013.

[37] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[38] Optimized approximative pow() in C / C++. http://martin.ankerl.com/2012/01/25/optimized-approximative-pow-in-c-and-cpp/, 2012.

[39] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[40] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD '12: Proceedings of the 2012 international conference on Management of Data*, May 2012.

[41] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems*, Apr. 2012.

[42] P. Purdom, S. Stigler, and T.-O. Cheam. Statistical investigation of three storage allocation algorithms. *BIT Numerical Mathematics*, 11(2), 1971.

[43] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, June 2012.

[44] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *Proc. VLDB*, Sept. 2007.

[45] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[46] VoltDB, the NewSQL database for high velocity applications. http://voltdb.com/, 2014.

[47] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science*, 1995.

[48] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013.

NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms

Jinho Hwang[†] K. K. Ramakrishnan[*] Timothy Wood[†]
[†]*The George Washington University* [*]*WINLAB, Rutgers University*

Abstract

NetVM brings virtualization to the Network by enabling high bandwidth network functions to operate at near line speed, while taking advantage of the flexibility and customization of low cost commodity servers. NetVM allows customizable data plane processing capabilities such as firewalls, proxies, and routers to be embedded within virtual machines, complementing the control plane capabilities of Software Defined Networking. NetVM makes it easy to dynamically scale, deploy, and reprogram network functions. This provides far greater flexibility than existing purpose-built, sometimes proprietary hardware, while still allowing complex policies and full packet inspection to determine subsequent processing. It does so with dramatically higher throughput than existing software router platforms.

NetVM is built on top of the KVM platform and Intel DPDK library. We detail many of the challenges we have solved such as adding support for high-speed inter-VM communication through shared huge pages and enhancing the CPU scheduler to prevent overheads caused by inter-core communication and context switching. NetVM allows true zero-copy delivery of data to VMs both for packet processing and messaging among VMs within a trust boundary. Our evaluation shows how NetVM can compose complex network functionality from multiple pipelined VMs and still obtain throughputs up to 10 Gbps, an improvement of more than 250% compared to existing techniques that use SR-IOV for virtualized networking.

1 Introduction

Virtualization has revolutionized how data center servers are managed by allowing greater flexibility, easier deployment, and improved resource multiplexing. A similar change is beginning to happen within communication networks with the development of virtualization of network functions, in conjunction with the use of software defined networking (SDN). While the migration of network functions to a more software based infrastructure is likely to begin with edge platforms that are more "control plane" focused, the flexibility and cost-effectiveness obtained by using common off-the-shelf hardware and systems will make migration of other network functions attractive. One main deterrent is the achievable performance and scalability of such virtualized platforms compared to purpose-built (often proprietary) networking hardware or middleboxes based on custom ASICs.

Middleboxes are typically hardware-software packages that come together on a special-purpose appliance, often at high cost. In contrast, a high throughput platform based on virtual machines (VMs) would allow network functions to be deployed dynamically at nodes in the network with low cost. Further, the shift to VMs would let businesses run network services on existing cloud platforms, bringing multiplexing and economy of scale benefits to network functionality. Once data can be moved to, from and between VMs at line rate for all packet sizes, we approach the long-term vision where the line between data centers and network resident "boxes" begins to blur: both software and network infrastructure could be developed, managed, and deployed in the same fashion.

Progress has been made by network virtualization standards and SDN to provide greater configurability in the network [1–4]. SDN improves flexibility by allowing software to manage the network control plane, while the performance-critical data plane is still implemented with proprietary network hardware. SDN allows for new flexibility in how data is forwarded, but the focus on the control plane prevents dynamic management of many types of network functionality that rely on the data plane, for example the information carried in the packet payload.

This limits the types of network functionality that can be "virtualized" into software, leaving networks to continue to be reliant on relatively expensive network appliances that are based on purpose-built hardware.

Recent advances in network interface cards (NICs) allow high throughput, low-latency packet processing using technologies like Intel's Data Plane Development Kit (DPDK) [5]. This software framework allows end-host applications to receive data directly from the NIC, eliminating overheads inherent in traditional interrupt driven OS-level packet processing. Unfortunately, the DPDK framework has a somewhat restricted set of options for support of virtualization, and on its own cannot support the type of flexible, high performance functionality that network and data center administrators desire.

To improve this situation, we have developed NetVM, a platform for running complex network functionality at line-speed (10Gbps) using commodity hardware. NetVM takes advantage of DPDK's high throughput packet processing capabilities, and adds to it abstractions that enable in-network services to be flexibly created,

chained, and load balanced. Since these "virtual bumps" can inspect the full packet data, a much wider range of packet processing functionality can be supported than in frameworks utilizing existing SDN-based controllers manipulating hardware switches. As a result, NetVM makes the following innovations:

1. A virtualization-based platform for flexible network service deployment that can meet the performance of customized hardware, especially those involving complex packet processing.
2. A shared-memory framework that truly exploits the DPDK library to provide zero-copy delivery to VMs and between VMs.
3. A hypervisor-based switch that can dynamically adjust a flow's destination in a state-dependent (e.g., for intelligent load balancing) and/or data-dependent manner (e.g., through deep packet inspection).
4. An architecture that supports high speed inter-VM communication, enabling complex network services to be spread across multiple VMs.
5. Security domains that restrict packet data access to only trusted VMs.

We have implemented NetVM using the KVM and DPDK platforms—all the aforementioned innovations are built on the top of DPDK. Our results show how NetVM can compose complex network functionality from multiple pipelined VMs and still obtain line rate throughputs of 10Gbps, an improvement of more than 250% compared to existing SR-IOV based techniques. We believe NetVM will scale to even higher throughputs on machines with additional NICs and processing cores.

2 Background and Motivation

This section provides background on the challenges of providing flexible network services on virtualized commodity servers.

2.1 Highspeed COTS Networking

Software routers, SDN, and hypervisor based switching technologies have sought to reduce the cost of deployment and increase flexibility compared to traditional network hardware. However, these approaches have been stymied by the performance achievable with commodity servers [6–8]. These limitations on throughput and latency have prevented software routers from supplanting custom designed hardware [9–11].

There are two main challenges that prevent commercial off-the-shelf (COTS) servers from being able to process network flows at line speed. First, network packets arrive at unpredictable times, so interrupts are generally used to notify an operating system that data is ready for processing. However, interrupt handling can be expensive because modern superscalar processors use

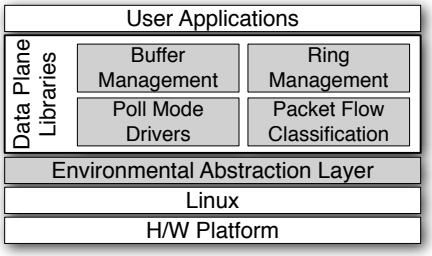


Figure 1: DPDK's run-time environment over Linux.

long pipelines, out-of-order and speculative execution, and multi-level memory systems, all of which tend to increase the penalty paid by an interrupt in terms of cycles [12, 13]. When the packet reception rate increases further, the achieved (receive) throughput can drop dramatically in such systems [14]. Second, existing operating systems typically read incoming packets into kernel space and then copy the data to user space for the application interested in it. These extra copies can incur an even greater overhead in virtualized settings, where it may be necessary to copy an additional time between the hypervisor and the guest operating system. These two sources of overhead limit the the ability to run network services on commodity servers, particularly ones employing virtualization [15, 16].

The Intel DPDK platform tries to reduce these overheads by allowing user space applications to directly poll the NIC for data. This model uses Linux's huge pages to pre-allocate large regions of memory, and then allows applications to DMA data directly into these pages. Figure 1 shows the DPDK architecture that runs in the application layer. The poll mode driver allows applications to access the NIC card directly without involving kernel processing, while the buffer and ring management systems resemble the memory management systems typically employed within the kernel for holding `sk_buffs`.

While DPDK enables high throughput user space applications, it does not yet offer a complete framework for constructing and interconnecting complex network services. Further, DPDK's passthrough mode that provides direct DMA to and from a VM can have significantly lower performance than native IO[1]. For example, DPDK supports Single Root I/O Virtualization (SR-IOV[2]) to allow multiple VMs to access the NIC, but packet "switching" (i.e., demultiplexing or load balancing) can only be performed based on the L2 address. As depicted in Figure 2(a), when using SR-IOV, packets are switched on

[1] Until Sandy-bridge, the performance was close to half of native performance, but with the next generation Ivy-bridge processor, the claim has been that performance has improved due to IOTLB (I/O Translation Lookaside Buffer) super page support [17]. But no performance results have been released.

[2] SR-IOV makes it possible to logically partition a NIC and expose to each VM a separate PCI-based NIC called a "Virtual Function" [18].

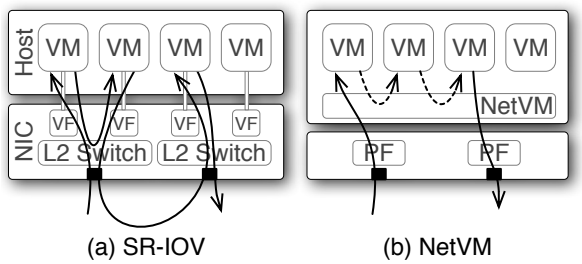


Figure 2: DPDK uses per-port switching with SR-IOV, whereas NetVM provides a global switch in the hypervisor and shared-memory packet transfer (dashed lines).

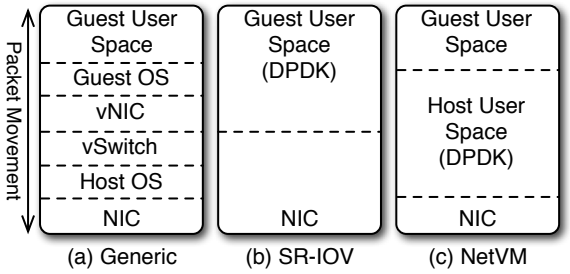


Figure 3: Architectural Differences for Packet Delivery in Virtualized Platform.

a per-port basis in the NIC, which means a second data copy is required if packets are forwarded between VMs on a shared port. Even worse, packets must go out of the host and come back via an external switch to be transmitted to a VM that is connected to another port's virtual function. Similar overheads appear for other VM switching platforms, e.g., Open vSwitch [19] and VMware's vNetwork distributed switch [20]. We seek to overcome this limitation in NetVM by providing a flexible switching capability without copying packets as shown in Figure 2(b). This improves performance of communication between VMs, which plays an important role when chained services are deployed.

Intel recently released an integration of DPDK and Open vSwitch [21] to reduce the limitations of SR-IOV switching. However, the DPDK vSwitch still requires copying packets between the hypervisor and the VM's memory, and does not support directly-chained VM communication. NetVM's enhancements go beyond DPDK vSwitch by providing a framework for flexible state- or data-dependent switching, efficient VM communication, and security domains to isolate VM groups.

2.2 Flexible Network Services

While platforms like DPDK allow for much faster processing, they still have limits on the kind of flexibility they can provide, particularly for virtual environments. The NIC based switching supported by DPDK + SR-IOV is not only expensive, but is limited because the NIC only has visibility into Layer 2 headers. With current techniques, each packet with a distinct destination MAC can be delivered to a different destination VM. However, in a network resident box (such as a middlebox acting as a firewall, a proxy, or even if the COTS platform is acting as a router), the destination MAC of incoming packets is the same. While advances in NIC design could reduce these limitations, a hardware based solution will never match the flexibility of a software-based approach.

By having the hypervisor perform the initial packet switching, NetVM can support more complex and dynamic functionality. For example, each application that supports a distinct function may reside in a separate VM, and it may be necessary to exploit flow classification to properly route packets through VMs based on mechanisms such as shallow (header-based) or deep (data-based) packet analysis. At the same time, NetVM's switch may use state-dependent information such as VM load levels, time of day, or dynamically configured policies to control the switching algorithm. Delivery of packets based on such rules is simply not feasible with current platforms.

2.3 Virtual Machine Based Networking

Network providers construct overall network functionality by combining middleboxes and network hardware that typically have been built by a diverse set of vendors. While NetVM can enable fast packet processing in software, it is the use of virtualization that will permit this diverse set of services to "play nice" with each other—virtualization makes it trivial to encapsulate a piece of software and its OS dependencies, dramatically simplifying deployment compared to running multiple processes on one bare-metal server. Running these services within VMs also could permit user-controlled network functions to be deployed into new environments such as cloud computing platforms where VMs are the norm and isolation between different network services would be crucial.

The consolidation and resource management benefits of virtualization are also well known. Unlike hardware middleboxes, VMs can be instantiated on demand when and where they are needed. This allows NetVM to multiplex one server for several related network functions, or to dynamically spawn VMs where new services are needed. Compared to network software running on bare metal, using a VM for each service simplifies resource allocation and improves performance isolation. These characteristics are crucial for network services that often have strict performance requirements.

3 System Design

Figure 3 compares two existing, commonly implemented network virtualization techniques against NetVM. In the first case, representing traditional virtualization plat-

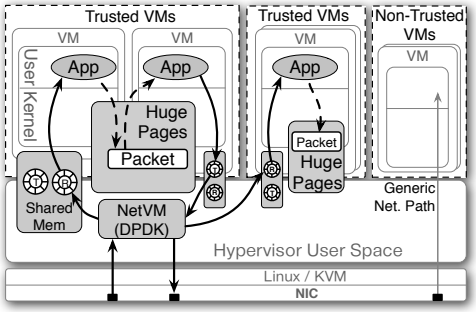


Figure 4: NetVM only requires a simple descriptor to be copied via shared memory (solid arrows), which then gives the VM direct access to packet data stored in huge pages (dashed arrow).

forms, packets arrive at the NIC and are copied into the hypervisor. A virtual switch then performs L2 (or a more complex function, based on the full 5-tuple packet header) switching to determine which VM is the recipient of the packet and notifies the appropriate virtual NIC. The memory page containing the packet is then either copied or granted to the Guest OS, and finally the data is copied to the user space application. Not surprisingly, this process involves significant overhead, preventing line-speed throughput.

In the second case (Figure 3(b)), SR-IOV is used to perform L2 switching on the NIC itself, and data can be copied directly into User Space of the appropriate VM. While this minimizes data movement, it does come at the cost of limited flexibility in how packets are routed to the VM, since the NIC must be configured with a static mapping and packet header information other than the MAC address cannot be used for routing.

The architecture of NetVM is shown in Figure 3(c). It does not rely on SR-IOV, instead allowing a user space application in the hypervisor to analyze packets and decide how to forward them. However, rather than copy data to the Guest, we use a shared memory mechanism to directly allow the Guest user space application to read the packet data it needs. This provides both flexible switching and high performance.

3.1 Zero-Copy Packet Delivery

Network providers are increasingly deploying complex services composed of routers, proxies, video transcoders, etc., which NetVM could consolidate onto a single host. To support fast communication between these components, NetVM employs two communication channels to quickly move data as shown in Figure 4. The first is a small, shared memory region (shared between the hypervisor and each individual VM) that is used to transmit packet descriptors. The second is a huge page region shared with a group of trusted VMs that allows chained applications to directly read or write packet data. Memory sharing through a "grant" mechanism is commonly used to transfer control of pages between the hypervisor and guest; by expanding this to a region of memory accessible by all trusted guest VMs, NetVM can enable efficient processing of flows traversing multiple VMs.

NetVM Core, running as a DPDK enabled user application, polls the NIC to read packets directly into the huge page area using DMA. It decides where to send each packet based on information such as the packet headers, possibly content, and/or VM load statistics. NetVM inserts a descriptor of the packet in the ring buffer that is setup between the individual destination VM and hypervisor. Each individual VM is identified by a "role number"−a representation of each network function, that is assigned by the VM manager. The descriptor includes a mbuf location (equivalent to a sk_buff in the Linux kernel) and huge page offset for packet reception. When transmitting or forwarding packets, the descriptor also specifies the action (transmit through the NIC, discard, or forward to another VM) and role number (i.e., the destination VM role number when forwarding). While this descriptor data must be copied between the hypervisor and guest, it allows the guest application to then directly access the packet data stored in the shared huge pages.

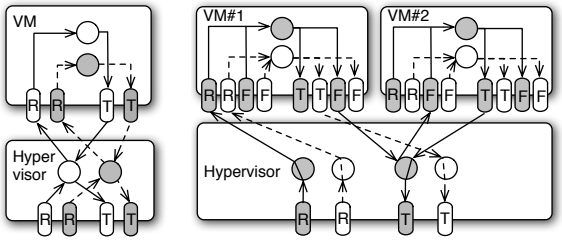
After the guest application (typically implementing some form of network functionality like a router or firewall) analyzes the packet, it can ask NetVM to forward the packet to a different VM or transmit it over the network. Forwarding simply repeats the above process—NetVM copies the descriptor into the ring buffer of a different VM so that it can be processed again; the packet data remains in place in the huge page area and never needs to be copied (although it can be independently modified by the guest applications if desired).

3.2 Lockless Design

Shared memory is typically managed with locks, but locks inevitably degrade performance by serializing data accesses and increasing communication overheads. This is particularly problematic for high-speed networking: to maintain full 10 Gbps throughput independent of packet size, a packet must be processed within 67.2 ns [22], yet context switching for a contested lock takes on the order of micro-seconds [23, 24], and even an uncontested lock operation may take tens of nanoseconds [25]. Thus a single context switch could cause the system to fall behind, and thus may result in tens of packets being dropped.

We avoid these issues by having parallelized queues with dedicated cores that service them. When working with NICs that have multiple queues and Receive Side Scaling (RSS) capability[1], the NIC receives pack-

[1] Modern NICs support RSS, a network driver technology to allow packet receive processing to be load balanced across multiple processors or cores [26].



(a) Single VM (b) Multiple VMs (Inter-VM)

Figure 5: Lockless and NUMA-Aware Queue/Thread Management (R = Receive Queue, T = Transmit Queue, and F = Forward Queue).

ets from the link and places them into one of several flow queues based on a configurable (usually an n-tuple) hash [27]. NetVM allows only two threads to manipulate this shared circular queue—the (producer) DPDK thread run by a core in the hypervisor and the (consumer) thread in the guest VM that performs processing on the packet. There is only a single producer and a single consumer, so synchronization is not required since neither will read or write simultaneously to the same region.

Our approach eliminates the overhead of locking by dedicating cores to each queue. This still permits scalability, because we can simply create additional queues (each managed by a pair of threads/cores). This works with the NIC's support for RSS, since incoming flows can automatically be load balanced across the available queues. Note that synchronization is not required to manage the huge page area either, since only one application will ever have control of the descriptor containing a packet's address.

Figure 5(a) depicts how two threads in a VM deliver packets without interrupting each other. Each core (marked as a circle) in the hypervisor receives packets from the NIC and adds descriptors to the tail of its own queue. The guest OS also has two dedicated cores, each of which reads from the head of its queue, performs processing, and then adds the packet to a transmit queue. The hypervisor reads descriptors from the tail of these queues and causes the NIC to transmit the associated packets. This thread/queue separation guarantees that only a single entity accesses the data at a time.

3.3 NUMA-Aware Design

Multi-processor systems exhibit NUMA characteristics, where memory access time depends on the memory location relative to a processor. Having cores on different sockets access memory that maps to the same cache line should be avoided, since this will cause expensive cache invalidation messages to ping pong back and forth between the two cores. As a result, ignoring the NUMA aspects of modern servers can cause significant performance degradation for latency sensitive tasks like network processing [28, 29].

Quantitatively, a last-level-cache (L3) hit on a 3GHz Intel Xeon 5500 processor takes up to 40 cycles, but the miss penalty is up to 201 cycles [30]. Thus if two separate sockets in NetVM end up processing data stored in nearby memory locations, the performance degradation can potentially be up to five times, since cache lines will end up constantly being invalidated.

Fortunately, NetVM can avoid this issue by carefully allocating and using huge pages in a NUMA-aware fashion. When a region of huge pages is requested, the memory region is divided uniformly across all sockets, thus each socket allocates a total of *(total huge page size / number of sockets)* bytes of memory from DIMMs that are local to the socket. In the hypervisor, NetVM then creates the same number of receive/transmit threads as there are sockets, and each is used only to process data in the huge pages local to that socket. The threads inside the guest VMs are created and pinned to the appropriate socket in a similar way. This ensures that as a packet is processed by either the host or the guest, it always stays in a local memory bank, and cache lines will never need to be passed between sockets.

Figure 5 illustrates how two sockets (gray and white) are managed. That is, a packet handled by gray threads is never moved to white threads, thus ensuring fast memory accesses and preventing cache coherency overheads. This also shows how NetVM pipelines packet processing across multiple cores—the initial work of handling the DMAed data from the NIC is performed by cores in the hypervisor, then cores in the guest perform packet processing. In a multi-VM deployment where complex network functionality is being built by chaining together VMs, the pipeline extends to an additional pair of cores in the hypervisor that can forward packets to cores in the next VM. Our evaluation shows that this pipeline can be extended as long as there are additional cores to perform processing (up to three separate VMs in our testbed).

3.4 Huge Page Virtual Address Mapping

While each individual huge page represents a large contiguous memory area, the full huge page region is spread across the physical memory both because of the per-socket allocations described in Section 3.3, and because it may be necessary to perform multiple huge page allocations to reach the desired total size if it is bigger than the default unit of huge page size−the default unit size can be found under /proc/meminfo. This poses a problem since the address space layout in the hypervisor is not known by the guest, yet guests must be able to find packets in the shared huge page region based on the address in the descriptor. Thus the address where a packet is placed by the NIC is only meaningful to the hypervisor; the address must be translated so that the guest will

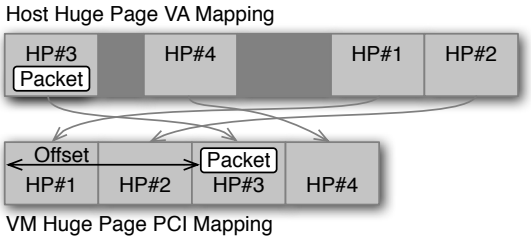


Figure 6: The huge pages spread across the host's memory must be contiguously aligned within the VM. NetVM must be able to quickly translate the address of a new packet from the host's virtual address space to an offset within the VM's address space.

be able to access it in the shared memory region. Further, looking up these addresses must be as fast as possible in order to perform line-speed packet processing.

NetVM overcomes the first challenge by mapping the huge pages into the guest in a contiguous region, as shown in Figure 6. NetVM exposes these huge pages to guest VMs using an emulated PCI device. The guest VM runs a driver that polls the device and maps its memory into user space, as described in Section 4.3. In effect, this shares the entire huge page region among all trusted guest VMs and the hypervisor. Any other untrusted VMs use a regular network interface through the hypervisor, which means they are not able to see the packets received from NetVM.

Even with the huge pages appearing as a contiguous region in the guest's memory space, it is non-trivial to compute where a packet is stored. When NetVM DMAs a packet into the huge page area, it receives a descriptor with an address *in the hypservisor's virtual address space*, which is meaningless to the guest application that must process the packet. While it would be possible to scan through the list of allocated huge pages to determine where the packet is stored, that kind of processing is simply too expensive for high-speed packet rates because every packet needs to go through this process. To resolve this problem, NetVM uses only bit operations and precomputed lookup tables; our experiments show that this improves throughput by up to 10% (with 8 huge pages) and 15% (with 16 huge pages) in the worst case compared to a naive lookup.

When a packet is received, we need to know which huge page it belongs to. Firstly, we build up an index map that converts a packet address to a huge page index. The index is taken from the upper 8 bits of its address (31^{st} bit to 38^{th} bit). The first 30 bits are the offset in the corresponding huge page, and the rest of the bits (left of the 38^{th} bit) can be ignored. We denote this function as $IDMAP(h) = (h >> 30)\&0xFF$, where h is a memory address. This value is then used as an index into an array $HMAP[i]$ to determine the huge page number.

To get the address base (i.e., a starting address of each huge page in the ordered and aligned region) of the huge page where the packet belongs to, we need to establish an accumulated address base. If all the huge pages have the same size, we do not need this address base−instead, just multiplying is enough, but since there can be different huge page sizes, we need to keep track of an accumulated address base. A function $HIGH(i)$ keeps a starting address of each huge page index i. Lastly, the residual address is taken from last 30 bits of a packet address using $LOW(a) = a\&0x3FFFFFFF$. $OFFSET(p) = HIGH(HMAP[IDMAP(p)]) \mid LOW(p)$ returns an address offset of contiguous huge pages in the emulated PCI.

3.5 Trusted and Untrusted VMs

Security is a key concern in virtualized cloud platforms. Since NetVM aims to provide zero-copy packet transmission while also having the flexibility to steer flows between cooperating VMs, it shares huge pages assigned in the hypervisor with multiple guest VMs. A malicious VM may be able to guess where the packets are in this shared region to eavesdrop or manipulate traffic for other VMs. Therefore, there must be a clear separation between trusted VMs and non-trusted VMs. NetVM provides a group separation to achieve the necessary security guarantees. When a VM is created, it is assigned to a trust group, which determines what range of memory (and thus which packets) it will have access to.

While our current implementation supports only trusted or untrusted VMs, it is possible to subdivide this further. Prior to DMAing packet data into a huge page, DPDK's classification engine can perform a shallow analysis of the packet and decide which huge page memory pool to copy it to. This would, for example, allow traffic flows destined for one cloud customer to be handled by one trust group, while flows for a different customer are handled by a second NetVM trust group on the same host. In this way, NetVM enables not only greater flexibility in network function virtualization, but also greater security when multiplexing resources on a shared host.

Figure 4 shows a separation between trusted VM groups and a non-trusted VM. Each trusted VM group gets its own memory region, and each VM gets a ring buffer for communication with NetVM. In constrast, non-trusted VMs only can use generic network paths such as those in Figure 3 (a) or (b).

4 Implementation Details

NetVM's implementation includes the NetVM Core Engine (the DPDK application running in the hypervisor), a NetVM manager, drivers for an emulated PCI device, modifications to KVM's CPU allocation policies, and NetLib (our library for building in-network functional-

ity in VM's userspace). Our implementation is built on QEMU 1.5.0 (KVM included), and DPDK 1.4.1.

KVM and QEMU allow a regular Linux host to run one or more VMs. Our functionality is split between code in the guest VM, and code running in user space of the host operating system. We use the terms host operating system and hypervisor interchangeably in this discussion.

4.1 NetVM Manager

The NetVM manager runs in the hypervisor and provides a communication channel so that QEMU can pass information to the NetVM core engine about the creation and destruction of VMs, as well as their trust level. When the NetVM manager starts, it creates a server socket to communicate with QEMU. Whenever QEMU starts a new VM, it connects to the socket to ask the NetVM Core to initialize the data structures and shared memory regions for the new VM. The connection is implemented with a socket-type chardev with "-chardev socket,path=<path>,id=<id>" in the VM configuration. This is a common approach to create a communication channel between a VM and an application running in the KVM host, rather than relying on hypervisor-based messaging [31].

NetVM manager is also responsible for storing the configuration information that determines VM trust groups (i.e., which VMs should be able to connect to NetVM Core) and the switching rules. These rules are passed to the NetVM Core Engine, which implements these policies.

4.2 NetVM Core Engine

The NetVM Core Engine is a DPDK userspace application running in the hypervisor. NetVM Core is initialized with user settings such as the processor core mapping, NIC port settings, and the configuration of the queues. These settings determine how many queues are created for receiving and transmitting packets, and which cores are allocated to each VM for these tasks. NetVM Core then allocates the Huge Page region and initializes the NIC so it will DMA packets into that area when polled.

The NetVM core engine has two roles: the first role is to receive packets and deliver/switch them to VMs (using zero-copy) following the specified policies, and the other role is to communicate with the NetVM manager to synchronize information about new VMs. The main control loop first polls the NIC and DMAs packets to huge pages in a burst (batch), then for each packet, NetVM decides which VM to notify. Instead of copying a packet, NetVM creates a tiny packet descriptor that contains the huge page address, and puts that into the private shared ring buffer (shared between the VM and NetVM Core). The actual packet data is accessible to the VM via shared memory, accessible over the emulated PCI device described below.

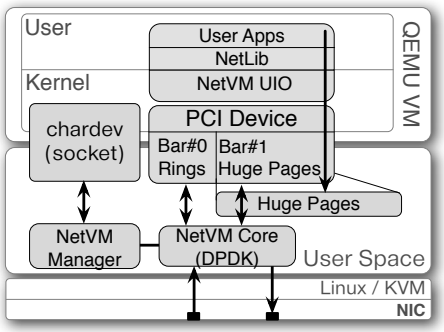


Figure 7: NetVM's architecture spans the guest and host systems; an emulated PCI device is used to share memory between them.

4.3 Emulated PCI

QEMU and KVM do not directly allow memory to be shared between the hypervisor and VMs. To overcome this limitation, we use an emulated PCI device that allows a VM to map the device's memory—since the device is written in software, this memory can be redirected to any memory location owned by the hypervisor. NetVM needs two seperate memory regions: a private shared memory (the address of which is stored in the device's BAR#0 register) and huge page shared memory (BAR#1). The private shared memory is used as ring buffers to deliver the status of user applications (VM $\rightarrow$ hypervisor) and packet descriptors (bidirectional). Each VM has this individual private shared memory. The huge page area, while not contiguous in the hypervisor, must be mapped as one contiguous chunk using the memory_region_add_subregion function. We illustrated how the huge pages map to virtual addresses, earlier in Section 3.4. In our current implementation, all VMs access the same shared huge page region, although this could be relaxed as discussed in 3.5.

Inside a guest VM that wishes to use NetVM's high-speed IO, we run a front-end driver that accesses this emulated PCI device using Linux's Userspace I/O framework (UIO). UIO was introduced in Linux 2.6.23 and allows device drivers to be written almost entirely in userspace. This driver maps the two memory regions from the PCI device into the guest's memory, allowing a NetVM user application, such as a router or firewall, to directly work with the incoming packet data.

4.4 NetLib and User Applications

Application developers do not need to know anything about DPDK or NetVM's PCI device based communication channels. Instead, our NetLib framework provides an interface between PCI and user applications. User applications only need to provide a structure containing

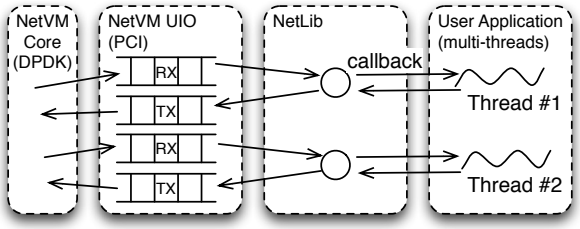


Figure 8: NetLib provides a bridge between PCI device and user applications.

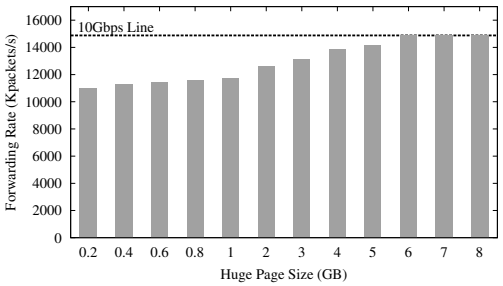


Figure 9: Huge page size can degrade throughput up to 26% (64-byte packets). NetVM needs 6GB to achieve the line rate speed.

configuration settings such as the number of cores, and a callback function. The callback function works similar to NetFilter in the linux kernel [32], a popular framework for packet filtering and manipulation. The callback function is called when a packet is received. User applications can read and write into packets, and decide what to do next. Actions include discard, send out to NIC, and forward to another VM. As explained in Section 4.1, user applications know the role numbers of other VMs. Therefore, when forwarding packets to another VM, user applications can specify the role number, not network addresses. This abstraction provides an easy way to implement communication channels between VMs.

Figure 8 illustrates a packet flow. When a packet is received from the hypervisor, a thread in NetLib fetches it and calls back a user application with the packet data. Then the user application processes the packet (read or/and write), and returns with an action. NetLib puts the action in the packet descriptor and sends it out to a transmit queue. NetLib supports multi-threading by providing each user thread with its own pair of input and output queues. There are no data exchanges between threads since NetLib provides a lockless model as NetVM does.

5 Evaluation

NetVM enables high speed packet delivery in-and-out of VMs and between VMs, and provides flexibility to steer traffic between function components that reside in distinct VMs on the NetVM platform. In this section, we evaluate NetVM with the following goals:

- Demonstrate NetVM's ability to provide high speed packet delivery with typical applications such as: Layer 3 forwarding, a userspace software router, and a firewall (§ 5.2),
- Show that the added latency with NetVM functioning as a middlebox is minimal (§ 5.3),
- Analyze the CPU time based on the task segment (§ 5.4), and
- Demonstrate NetVM's ability to steer traffic flexibly between VMs(§ 5.5).

In our experimental setup, we use two Xeon CPU X5650 @ 2.67GHz (2x6 cores) servers—one for the system under test and the other acting as a traffic generator—each of which has an Intel 82599EB 10G Dual Port NIC (with one port used for our performance experiments) and 48GB memory. We use 8GB for huge pages because Figure 9 shows that at least 6GB is needed to achieve the full line-rate (we have seen in Intel's performance reports setting 8GB as a default huge page size). The host OS is Red Hat 6.2 (kernel 2.6.32), and the guest OS is Ubuntu 12.10 (kernel 3.5). DPDK-1.4.1 and QEMU-1.5.0 are used. We use PktGen from WindRiver to generate traffic [33]. The base core assignment otherwise mentioned differently follows 2 cores to receive, 4 cores to transmit/forward, and 2 cores per VM.

We also compare NetVM with SR-IOV, the high performance IO pass-through system popularly used. SR-IOV allows the NIC to be logically partitioned into "virtual functions", each of which can be mapped to a different VM. We measure and compare the performance and flexibility provided by these architectures.

5.1 Applications

L3 Forwarder [34]: We use a simple layer-3 router. The forwarding function uses a hash map for the flow classification stage. Hashing is used in combination with a flow table to map each input packet to its flow at runtime. The hash lookup key is represented by a 5-tuple. The ID of the output interface for the input packet is read from the identified flow table entry. The set of flows used by the application is statically configured and loaded into the hash at initialization time (this simple layer-3 router is similar to the sample L3 forwarder provided in the DPDK library).

Click Userspace Router [10]: We also use Click, a more advanced userspace router toolkit to measure the performance that may be achieved by 'plugging in' an existing router implementation as-is into a VM, treating it as a 'container'. Click supports the composition of elements that each performs simple computations, but together can provide more advanced functionality such as IP routing. We have slightly modified Click by adding new receive and transmit elments that use Netlib for faster network IO. In total our changes comprise approximately 1000

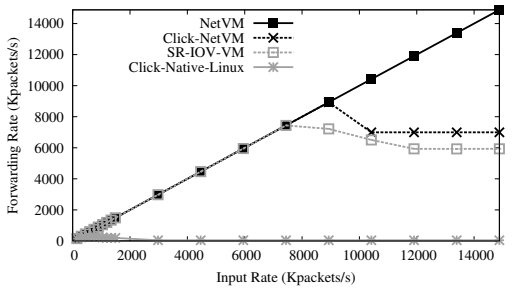


Figure 10: Forwarding rate as a function of input rate for NetVM, Click using NetVM, SR-IOV (DPDK in VM), and Native Linux Click-NetVM router (64-byte packets).

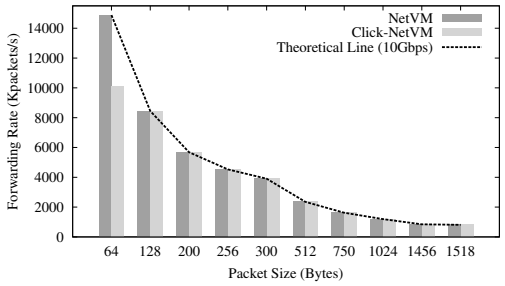


Figure 11: NetVM provides a line-rate speed regardless of packet sizes. Due to large application overhead, Click-NetVM achieves 6.8Gbps with 64-byte packet size.

lines of code. We test both a standard version of Click using Linux IO and our Netlib zero-copy version.

Firewall [35]: Firewalls control the flow of network traffic based on security policies. We use Netlib to build the foundational feature for firewalls—the packet filter. Firewalls with packet filters operate at layer 3, the network layer. This provides network access control based on several pieces of information in a packet, including the usual 5-tuple: the packet's source and destination IP address, network or transport protocol id, source and destination port; in addition its decision rules would also factor in the interface being traversed by the packet, and its direction (inbound or outbound).

5.2 High Speed Packet Delivery

Packet Forwarding Performance: NetVM's goal is to provide line rate throughput, despite running on a virtualized platform. To show that NetVM can indeed achieve this, we show the L3 packet forwarding rate vs. the input traffic rate. The theoretical value for the nominal 64-byte IP packet for a 10G Ethernet interface−with preamble size of 8 bytes, a minimum inter-frame gap 12 bytes−is 14,880,952 packets.

Figure 10 shows the input rate and the forwarded rate in packets/sec for three cases: NetVM's simple L3 forwarder, the Click router using NetVM (Click-NetVM), and Click router using native Linux (Click-Native-Linux). NetVM achieves the full line-rate, whereas Click-NetVM has a maximum rate of around 6Gbps. This is because Click has added overheads for scheduling elements (confirmed by the latency analysis we present subsequently in Table 1). Notice that increasing the input rate results in either a slight drop-off in the forwarding rate (as a result of wasted processing of packets that are ultimately dropped), or plateaus at that maximum rate. We believe Click-NetVM's performance could be further improved by either adding multi-threading support or using a faster processor, but SR-IOV can not achieve better performance this way. Not surprisingly, Click-Native-Linux performance is extremely poor (max 327Mbps), illustrating the dramatic improvement provided simply by zero-copy IO. [10].

With SR-IOV, the VM has two virtual functions associated with it and runs DPDK with two ports using two cores. SR-IOV achieves a maximum throughput of 5Gbps. We have observed that increasing the number of virtual functions or cores does not improve the maximum throughput. We speculate this limitation comes from the speed limitation on hardware switching.

Figure 11 now shows the forwarding rate as the packet size is varied. Since NetVM does not have further overheads as a consequence of the increased packet size (data is delivered by DMA), it easily achieves the full line-rate. Also, Click-NetVM also can provide the full line-rate for 128-byte and larger packet sizes.

Inter-VM Packet Delivery: NetVM's goal is to build complex network functionality by composing chains of VMs. To evaluate how pipelining VM processing elements affects throughput, we measure the achieved throughput when varying the number of VMs through which a packet must flow. We compare NetVM to a set of SR-IOV VMs, the state-of-the-art for virtualized networking.

Figure 12 shows that NetVM achieves a significantly higher base throughput for one VM, and that it is able to maintain nearly the line rate for chains of up to three VMs. After this point, our 12-core system does not have enough cores to dedicate to each VM, so there begins to be a processing bottleneck (e.g., four VMs require a total of 14 cores: 2 cores−one from each processor for NUMA-awareness−to receive packets in the host, 4 cores to transmit/forward between VMs, and 2 cores per VM for application-level processing). We believe that more powerful systems should easily be able to support longer chains using our architecture.

For a more realistic scenario, we consider a chain where 40% of incoming traffic is processed only by the first VM (an L2 switch) before being transmitted out the wire, while the remaining 60% is sent from the L2 switch VM through a Firewall VM, and then an L3 switch VM (e.g., a load balancer). In this case, our test machine has sufficient CPU capacity to achieve the line-rate for

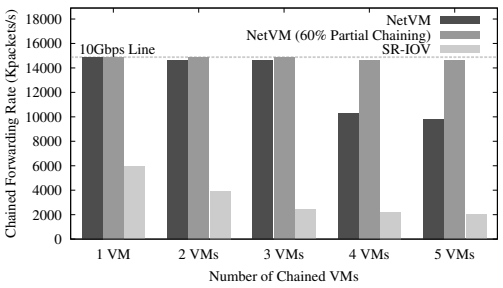


Figure 12: Inter-VM communication using NetVM can achieve a line-rate speed when VMs are well scheduled in different CPU cores (here, up to 3 VMs).

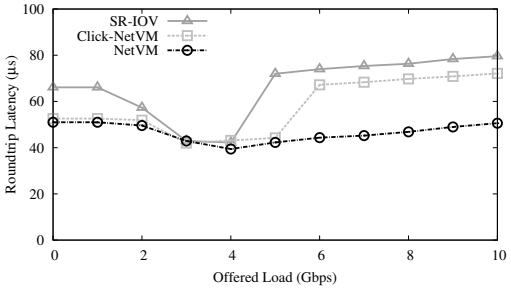


Figure 13: Average roundtrip latency for L3 forwarding.

the three VM chain, and sees only a small decrease if additional L3 switch VMs are added to the end of the chain. In contrast, SR-IOV performance is affected by the negative impact of IOTLB cache-misses, as well as a high data copy cost to move between VMs. Input/output memory management units (IOMMUs) use an IOTLB to speed up address resolution, but still each IOTLB cache-miss renders a substantial increase in DMA latency and performance degradation of DMA-intensive packet processing [36, 37].

5.3 Latency

While maintaining line-rate throughput is critical for in-network services, it is also important for the latency added by the processing elements to be minimized. We quantify this by measuring the average roundtrip latency for L3 forwarding in each platform. The measurement is performed at the traffic generator by looping back 64-byte packets sent through the platform. We include a timestamp on the packet transmitted. Figure 13 shows the roundtrip latency for the three cases: NetVM, Click-NetVM, and SR-IOV using identical L3 Forwarding function. Latency for Click-NetVM and SR-IOV increases especially at higher loads when there are additional packet processing delays under overload. We speculate that at very low input rates, none of the systems are able to make full benefit of batched DMAs and pipelining between cores, explaining the initially slightly worse performance for all approaches. After the offered load exceeds 5Gbps, SR-IOV and Click are unable to keep up, causing a significant portion of packets to be dropped. In this experiment, the queue lengths are relatively small, preventing the latency from rising significantly. The drop rate of SR-IOV rises to 60% at 10Gbps, while NetVM drops zero pockets.

5.4 CPU Time Breakdown

Table 1 breaks down the CPU cost of forwarding a packet through NetVM. Costs were converted to nanoseconds from the Xeon's cycle counters [38]. Each measurement is the average over a 10 second test. These measurements are larger than the true values because using Xeon cycle counters has significant overhead (the achieved throughput drops from 10Gbps to 8.7Gbps). Most of the tasks performed by a NetVM's CPU are included in the table.

"NIC → Hypervisor" measures the time it takes DPDK to read a packet from the NIC's receive DMA ring. Then NetVM decides which VM to send the packet to and puts a small packet descriptor in the VM's receive ring ("Hypervisor → VM"). Both of these actions are performed by a single core. "VM → APP" is the time NetVM needs to get a packet from a ring buffer and delivers it to the user application; the application then spends "APP (L3 Forwarding)" time; the forwarding application (NetVM or Click) sends the packet back to the VM ("APP → VM") and NetVM puts it into the VM's transmit ring buffer ("VM → Hypervisor"). Finally, the hypervisor spends "Hypervisor → NIC" time to send out a packet to the NIC's transmit DMA ring.

The Core# column demonstrates how packet descriptors are pipelined through different cores for different tasks. As was explained in Section 3.3, packet processing is restricted to the same socket to prevent NUMA overheads. In this case, only "APP (L3 Forwarding)" reads/writes the packet content.

5.5 Flexibility

NetVM allows for flexible switching capabilities, which can also help improve performance. Whereas Intel SR-IOV can only switch packets based on the L2 address, NetVM can steer traffic (per-packet or per-flow) to a spe-

Core#	Task	Time (ns/packet)	
		Simple	Click
0	NIC → Hypervisor	27.8	27.8
0	Hypervisor → VM	16.7	16.7
1	VM → APP	1.8	29.4
1	APP (L3 Forwarding)	37.7	41.5
1	APP → VM	1.8	129.0
1	VM → Hypervisor	1.8	1.8
2	Hypervisor → NIC	0.6	0.6
Total		88.3	246.8

Table 1: CPU Time Cost Breakdown for NetLib's Simple L3 router and Click L3 router.

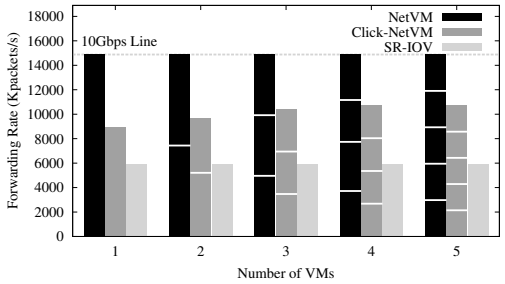


Figure 14: State-dependent (or data-dependent) load-balancing enables flexible steering of traffic. The graph shows a uniformly distributed load-balancing.

cific VM depending on system load (e.g., using the occupancy of the packet descriptor ring as an indication), shallow packet inspection (header checking), or deep packet inspection (header + payload checking) in the face of performance degradation. Figure 14 illustrates the forwarding rate when load-balancing is based on load of packets queued−the queue with the smallest number of packets has the highest priority. The stacked bars show how much traffic each VM receives and the total. NetVM is able to evenly balance load across VMs. Click-NetVM shows a significant performance improvement with multiple VMs (up to 20%) since additional cores are able to load balance the more expensive application-level processing. The SR-IOV system is simply unable to make use of multiple VMs in this way since the MAC addresses coming from the packet generator are all same. Adding more cores to the single SR-IOV VM does also not improve performance. We believe this will be a realistic scenario in the network (not just in our testbed) as the MAC addresses of incoming packets at a middlebox or a router will likely be the same across all packets.

We also have observed the same performance graph for NetVM's shallow packet inspection that load-balances based on the protocol type; deep-packet inspection overhead will depend on the amount of computation required while analyzing the packet. With many different network functions deployed, more dynamic workloads with SDN capability are left for the future works.

6 Discussion

We have shown NetVM's zero-copy packet delivery framework can effectively bring high performance for network traffic moving through a virtualized network platform. Here we discuss related issues, limitations, and future directions.

Scale to next generation machines: In this work, we have used the first CPU version (Nehalem architecture) that supports Intel's DPDK. Subsequent generations of processors from Intel, the Sandy-bridge and Ivy-bridge processors have significant additional hardware capabilities (i.e., cores), so we expect that this will allow both greater total throughput (by connecting to multiple NIC ports in parallel), and deeper VM chains. Reports in the commercial press and vendor claims indicate that there is almost a linear performance improvement with the number of cores for native Linux (i.e., non-virtualized). Since NetVM eliminates the overheads of other virtual IO techniques like SR-IOV, we also expect to see the same linear improvement by adding more cores and NICs.

Building Edge Routers with NetVM: We recognize that the capabilities of NetVM to act as a network element, such as an edge router in an ISP context, depends on having a large number of interfaces, albeit at lower speeds. While a COTS platform may have a limited number of NICs, each at 10Gbps, a judicious combination of a low cost Layer 2 (Ethernet) switch and NetVM will likely serve as an alternative to (what are generally high cost) current edge router platforms. Since the features and capabilities (in terms of policy and QoS) required on an edge router platform are often more complex, the cost of ASIC implementations tend to rise steeply. This is precisely where the additional processing power of the recent processors combined with the NetVM architecture can be an extremely attractive alternative. The use of the low cost L2 switch provides the necessary multiplexing/demultiplexing required to complement NetVM's ability to absorb complex functions, potentially with dynamic composition of those functions.

Open vSwitch and SDN integration: SDN allows greater flexibility for control plane management. However, the constraints of the hardware implementations of switches and routers often prevent SDN rules from being based on anything but simple packet header information. Open vSwitch has enabled greater network automation and reconfigurability, but its performance is limited because of the need to copy data. Our goal in NetVM is to build a base platform that can offer greater flexibility while providing high speed data movement underneath. We aim to integrate Open vSwitch capabilities into our NetVM Manager. In this way, the inputs that come from a SDN Controller using OpenFlow could be used to guide NetVM's management and switching behavior. NetVM's flexibility in demultiplexing can accommodate more complex rule sets, potentially allowing SDN control primitives to evolve.

Other Hypervisors: Our implementation uses KVM, but we believe the NetVM architecture could be applied to other virtualization platforms. For example, a similar setup could be applied to Xen; the NetVM Core would run in Domain-0, and Xen's grant table functionality would be used to directly share the memory regions used to store packet data. However, Xen's limited support for huge pages would have to be enhanced.

7 Related Work

The introduction of multi-core and multi-processor systems has led to significant advances in the capabilities of software based routers. The RouteBricks project sought to increase the speed of software routers by exploiting parallelism at both the CPU and server level [39]. Similarly, Kim et. al. [11] demonstrate how batching I/O and CPU operations can improve routing performance on multi-core systems. Rather than using regular CPU cores, PacketShader [28] utilizes the power of general purpose graphic processing units (GPGPU) to accelerate packet processing. Hyper-switch [40] on the other hand uses a low-overhead mechanism that takes into account CPU cache locality, especially in NUMA systems. All of these approaches demonstrate that the memory access time bottlenecks that prevented software routers such as Click [10] from performing line-rate processing are beginning to shift. However, none of these existing approaches support deployment of network services in virtual environments, a requirement that we believe is crucial for lower cost COTS platforms to replace purpose-built hardware and provide automated, flexible network function management.

The desire to implement network functions in software, to enable both flexibility and reduced cost because of running on COTS hardware, has recently taken concrete shape with a multitude of network operators and vendors beginning to work together in various industry forums. In particular, the work spearheaded by European Telecommunications Standards Institute (ETSI) on network function virtualization (NFV) has outlined the concept recently [41, 42]. While the benefits of NFV in reducing equipment cost and power consumption, improving flexibility, reduced time to deploy functionality and enabling multiple applications on a single platform (rather than having multiple purpose-specific network appliances in the network) are clear, there is still the outstanding problem of achieving high-performance. To achieve a fully capable NFV, high-speed packet delivery and low latency is required. NetVM provides the fundamental underlying platform to achieve this.

Improving I/O speeds in virtualized environments has long been a challenge. Santos et al. narrow the performance gap by optimzing Xen's driver domain model to reduce execution costs for gigabit Ethernet NICs [43]. vBalance dynamically and adaptively migrates the interrupts from a preempted vCPU to a running one, and hence avoids interrupt processing delays to improve the I/O performance for SMP-VMs [44]. vTurbo accelerates I/O processing for VMs by offloading that task to a designated core called a turbo core that runs with a much smaller time slice than the cores shared by production VMs [45]. VPE improves the performance of I/O device virtualization by using dedicated CPU cores [46]. However, none of these achieve full line-rate packet forwarding (and processing) for network links operating at 10Gbps or higher speeds. While we base our platform on DPDK, other approaches such as netmap [47] also provide highspeed NIC to userspace I/O.

Researchers have looked into middlebox virtualization on commodity servers. Split/Merge [48] describes a new abstraction (Split/Merge), and a system (FreeFlow), that enables transparent, balanced elasticity for stateful virtual middleboxes to have the ability to migrate flows dynamically. xOMB [6] provides flexible, programmable, and incrementally scalable middleboxes based on commodity servers and operating systems to achieve high scalability and dynamic flow management. CoMb [8] addresses key resource management and implementation challenges that arise in exploiting the benefits of consolidation in middlebox deployments. These systems provide flexible management of networks and are complementary to the the high-speed packet forwarding and processing capability of NetVM.

8 Conclusion

We have described a high-speed network packet processing platform, NetVM, built from commodity servers that use virtualization. By utilizing Intel's DPDK library, NetVM provides a flexible traffic steering capability under the hypervisor's control, overcoming the performance limitations of the existing, popular SR-IOV hardware switching techniques. NetVM provides the capability to chain network functions on the platform to provide a flexible, high-performance network element incorporating multiple functions. At the same time, NetVM allows VMs to be grouped into multiple trust domains, allowing one server to be safely multiplexed for network functionality from competing users.

We have demonstrated how we solve NetVM's design and implementation challenges. Our evaluation shows NetVM outperforms the current SR-IOV based system for forwarding functions and for functions spanning multiple VMs, both in terms of high throughput and reduced packet processing latency. NetVM provides greater flexibility in packet switching/demultiplexing, including support for state-dependent load-balancing. NetVM demonstrates that recent advances in multi-core processors and NIC hardware have shifted the bottleneck away from software-based network processing, even for virtual platforms that typically have much greater IO overheads.

Acknowledgments

We thank our shepherd, KyoungSoo Park, and reviewers for their help improving this paper. This work was supported in part by NSF grant CNS-1253575.

References

[1] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 29–42, Berkeley, CA, USA, 2013. USENIX Association.

[2] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[3] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 49–54, New York, NY, USA, 2012. ACM.

[4] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. Extending networking into the virtualization layer. In *8th ACM Workshop on Hot Topics inNetworks (HotNets-VIII).New YorkCity,NY(October 2009)*.

[5] Intel Corporation. Intel data plane development kit: Getting started guide. 2013.

[6] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xomb: extensible open middleboxes with commodity servers. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS '12, pages 49–60, New York, NY, USA, 2012. ACM.

[7] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev.*, 39(2):20–26, March 2009.

[8] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[9] Raffaele Bolla and Roberto Bruschi. Pc-based software routers: high performance and application service support. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '08, pages 27–32, New York, NY, USA, 2008. ACM.

[10] Eddie Kohler. The click modular router. *PhD Thesis*, 2000.

[11] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. The power of batching in the click modular router. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 14:1–14:6, New York, NY, USA, 2012. ACM.

[12] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. Hip: Hybrid interrupt-polling for the network interface. *ACM Operating Systems Reviews*, 35:50–60, 2001.

[13] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 3–3, Berkeley, CA, USA, 2012. USENIX Association.

[14] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15:217–252, 1997.

[15] Wenji Wu, Matt Crawford, and Mark Bowden. The performance analysis of linux networking - packet receiving. *Comput. Commun.*, 30(5):1044–1057, March 2007.

[16] Younggyun Koh, Calton Pu, Sapan Bhatia, and Charles Consel. Efficient packet processing in user-level os: A study of uml. In *in Proceedings of the 31th IEEE Conference on Local Computer Networks (LCN06)*, 2006.

[17] Intel Corporation. Intel virtualization technology for directed i/o. 2007.

[18] Intel Corp. Intel data plane development kit: Programmer's guide. 2013.

[19] Open vSwitch. http://www.openvswitch.org.

[20] VMWare White Paper. Vmware vnetwork distributed switch. *2013*.

[21] Intel Open Source Technology Center. https://01.org/packet-processing.

[22] Intel Corp. Intel data plane development kit: Getting started guide. 2013.

[23] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. Context switch overheads for linux on arm platforms. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[24] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[25] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. LADIS Keynote, 2009.

[26] Srihari Makineni, Ravi Iyer, Partha Sarangam, Donald Newell, Li Zhao, Ramesh Illikkal, and Jaideep Moses. Receive side coalescing for ac-

celerating tcp/ip processing. In *Proceedings of the 13th International Conference on High Performance Computing*, HiPC'06, pages 289–300, Berlin, Heidelberg, 2006. Springer-Verlag.

[27] Wind River White Paper. High-performance multicore networking software design options. 2013.

[28] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[29] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. Numa-aware algorithms: the case of data shuffling. *The biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[30] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500. 2013.

[31] A. Cameron Macdonell. Shared-memory optimizations for virtual machines. *PhD Thesis*.

[32] Rusty Russell and Harald Welte. Linux netfilter hacking howto. *http://www.netfilter.org/documentation/HOWTO /netfilter-hacking-HOWTO.html*.

[33] Wind River Technical Report. Wind river application acceleration engine. 2013.

[34] Intel Corp. Intel data plane development kit: Sample application user guide. 2013.

[35] Karen Scarfone and Paul Hoffman. Guidelines on firewalls and firewall policy. *National Institute of Standards and Technology*, 2009.

[36] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. Iommu: strategies for mitigating the iotlb bottleneck. In *Proceedings of the 2010 international conference on Computer Architecture*, ISCA'10, pages 256–274, Berlin, Heidelberg, 2012. Springer-Verlag.

[37] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. The price of safety: Evaluating iommu performance. In *In Proceedings of the 2007 Linux Symposium*, 2007.

[38] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual. 2013.

[39] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, page 1528, New York, NY, USA, 2009. ACM.

[40] Kaushik Kumar Ram, Alan L. Cox, Mehul Chadha, and Scott Rixner. Hyper-switch: A scalable software virtual switching architecture. *USENIX Annual Technical Conference (USENIX ATC)*, 2013.

[41] SDN and OpenFlow World Congress Introductory White Paper. Network functions virtualisation. *http://portal.etsi.org/NFV/NFV_White_Paper.pdf*, 2012.

[42] Frank Yue. Network functions virtualization - everything old is new again. *http://www.f5.com/pdf/white-papers/service-provider-nfv-white-paper.pdf*, 2013.

[43] Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[44] Luwei Cheng and Cho-Li Wang. vbalance: using interrupt load balance to improve i/o performance for smp virtual machines. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 2:1–2:14, New York, NY, USA, 2012. ACM.

[45] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. *USENIX Annual Technical Conference*, 2013.

[46] Jiuxing Liu and Bulent Abali. Virtualization polling engine (vpe): using dedicated cpu cores to accelerate i/o virtualization. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 225–234, New York, NY, USA, 2009. ACM.

[47] Luigi Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, Berkeley, CA, 2012. USENIX.

[48] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: system support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 227–240, Berkeley, CA, USA, 2013. USENIX Association.

ClickOS and the Art of Network Function Virtualization

Joao Martins[†], Mohamed Ahmed[†], Costin Raiciu[‡], Vladimir Olteanu[‡], Michio Honda[†], Roberto Bifulco[†], Felipe Huici[†]

[†] NEC Europe Ltd. [‡] University Politehnica of Bucharest

Abstract

Over the years middleboxes have become a fundamental part of today's networks. Despite their usefulness, they come with a number of problems, many of which arise from the fact that they are hardware-based: they are costly, difficult to manage, and their functionality is hard or impossible to change, to name a few.

To address these issues, there is a recent trend towards network function virtualization (NFV), in essence proposing to turn these middleboxes into software-based, virtualized entities. Towards this goal we introduce ClickOS, a high-performance, virtualized software middlebox platform. ClickOS virtual machines are small (5MB), boot quickly (about 30 milliseconds), add little delay (45 microseconds) and over one hundred of them can be concurrently run while saturating a 10Gb pipe on a commodity server. We further implement a wide range of middleboxes including a firewall, a carrier-grade NAT and a load balancer and show that ClickOS can handle packets in the millions per second.

1 Introduction

The presence of hardware-based network appliances (also known as middleboxes) has exploded, to the point where they are now an intrinsic and fundamental part of today's operational networks. They are essential to network operators, supporting a diverse set of functions ranging from security (firewalls, IDSes, traffic scrubbers), traffic shaping (rate limiters, load balancers), dealing with address space exhaustion (NATs) or improving performance (traffic accelerators, caches, proxies), to name a few. Middleboxes are ubiquitous: a third of access networks show symptoms of stateful middlebox processing [12] and in enterprise networks there are as many middleboxes deployed as routers and switches [37].

Despite their usefulness, recent reports and operator feedback reveal that such proprietary middleboxes come with a number of significant drawbacks [9]: middleboxes are expensive to buy and manage [37], and introducing new features means having to deploy new hardware at the next purchase cycle, a process which on average takes four years. Hardware middleboxes cannot easily be scaled up and down with shifting demand, and so must provisioned to cope with peak demand, which is wasteful. Finally, a considerable level of investment is needed to develop new hardware-based devices, which leaves potential small players out of the market and so raises innovation barriers.

To address these issues, Network Function Virtualization (NFV) has been recently proposed to shift middlebox processing from hardware appliances to software running on inexpensive, commodity hardware (e.g., x86 servers with 10Gb NICs). NFV has already gained a considerable momentum: seven of the world's leading telecoms network operators, along with 52 other operators, IT and equipment vendors and technology providers, have initiated a new standards group for the virtualization of network functions [8].

NFV platforms must support multi-tenancy, since they are intended to concurrently run software belonging to the operator and (potentially untrusted) third parties: colocated middleboxes should be isolated not only from a security but also a performance point of view [10]. Further, as middleboxes implement a large range of functionality, platforms should accommodate a wide range of OSes, APIs and software packages.

Is it possible to build a software-based virtualized middlebox platform that fits these requirements? Hypervisor-based technologies such as Xen or KVM are well established candidates and offer security and performance isolation out-of-the-box. However, they only support small numbers of tenants and their networking performance is unsatisfactory[1]. At a high-level, the reason for the poor performance is simple: neither the hypervisors (Xen or KVM), nor the guest OSes (e.g., Linux) have been optimized for middlebox processing.

In this paper we present the design, implementation and evaluation of ClickOS, a Xen-based software platform optimized for middlebox processing. To achieve high performance, ClickOS implements an extensive overhaul of Xen's I/O subsystem, including changes to the back-end switch, virtual net devices and back and front-end drivers. These changes enable ClickOS to significantly speed up networking in middleboxes running in Linux virtual machines: for simple packet generation, Linux throughput increases from 6.46 Gb/s to 9.68 Gb/s for 1500B packets and from 0.42 Gb/s to 5.73 Gb/s for minimum-sized packets.

This work was partly funded by the EU FP7 CHANGE (257422) project.

[1] In our tests, a Xen guest domain running Linux can only reach rates of 6.5 Gb/s on a 10Gb card for 1500-byte packets out-of-the-box; KVM reaches 7.5 Gb/s.

A key observation is that developing middleboxes as applications running over Linux (and other commodity OSes) is a complex task and uses few of the OS services beyond network connectivity. To allow ease of development, a much better choice is to use specialized frameworks to program middleboxes. Click [17] is a stand-out example as it allows users to build complex middlebox processing configurations by using simple, well known processing elements. Click is great for middlebox processing, but it currently needs Linux to function and so it inherits the overheads of commodity OSes.

To support fast, easily programmable middleboxes, ClickOS implements a minimalistic guest virtual machine that is optimized from the ground up to run Click processing at rates of millions of packets per second. ClickOS images are small (5MB), making it possible to run a large number of them (up to 400 in our tests). ClickOS virtual machines can boot and instantiate middlebox processing in under 30 milliseconds, and can saturate a 10Gb/s link for almost all packets sizes while concurrently running as many as 100 ClickOS virtual machines on a single CPU core.

2 Problem Statement

Our goal is to build a versatile, high performance software middlebox platform on commodity hardware. Such a platform must satisfy a number of performance and security requirements:

Flexibility to run different types of software middleboxes, relying on different operating systems or frameworks, coming from different vendors, and requested by the operator itself or potentially untrusted third-parties.

Isolation of memory, CPU, device access and performance to support multiple tenants on common hardware.

High Throughput and Low Delay: Middleboxes are typically deployed in operator environments so that it is common for them to have to handle large traffic rates (e.g., multiple 10Gb/s ports); the platform should be able to handle such rates, while adding only negligible delay to end-to-end RTTs.

Scalability: Running middleboxes for third-parties must be very efficient if it is to catch on. Ideally, the platform should ideally support a large number of middleboxes belonging to different third-parties, as long as only a small subset of them are seeing traffic at the same time. This implies that platforms must be able to quickly scale out processing with demand to make better use of additional resources on a server or additional servers, and to quickly scale down when demand diminishes.

How should middleboxes be programmed? The default today is to code them as applications or kernel changes on top of commodity OSes. This allows much **flexibility** in choosing the development tools and languages, at the cost of having to run one commodity OS to support a middlebox.

In addition, a large fraction of functionality is common across different middleboxes, making it important to support **code re-use** to reduce prototyping effort, and processing re-use to reduce overhead [36].

3 Related Work

There is plenty of related work we could leverage to build NFV platforms. Given that the goal is to isolate different middleboxes running on the same hardware, the choice is either containers (`chroot`, FreeBSD Jails, Solaris Zones, OpenVZ [44, 45, 27]) or hypervisors (VMWare Server, Hyper-V, KVM, Xen [40, 21, 16, 3]).

Containers are lightweight but inflexible, forcing all middleboxes to run on the same operating system. This is a limitation even in the context of an operator wanting to run software middleboxes from different vendors.

Hypervisors provide the flexibility needed for multi-tenant middleboxes (i.e., different guest operating systems are able to run on the same platform), but this is at the cost of high performance, especially in networking. For high-performance networking with hypervisors, the typical approach today is to utilize device pass-through, whereby virtual machines are given direct access to a device (NIC). Pass-through has a few downsides: it complicates live migration, and it reduces scalability since the device is monopolized by a given virtual machine. The latter issue is mitigated by modern NICs supporting technologies such as hardware multi-queuing, VMDq and SR-IOV [14], however the number of VMs is still limited by the number of queues offered by the device. In this work we will show that it is possible to maintain performance scalability even without device pass-through.

Minimalistic OSes and VMs: Minimalistic OSes or micro kernels are attractive because, unlike traditional OSes, they aim provide just the required functionality for the job. While many minimalist OSes have been built [22, 23, 1, 42, 43], they typically lack driver support for a wide range of devices (especially NICs), and most do not run in virtualized environments. With respect to ClickOS, Mirage [19] is also a Xen VM built on top of MiniOS, but the focus is to create Ocaml, type-safe virtualized applications and, as such, its network performance is not fully optimized (e.g., 1.7 Gb/s for TCP traffic). Erlang on Xen, LuaJIT and HalVM also leverage MiniOS to provide Erlang, Lua, and Haskell programming environments; none target middlebox processing nor are optimized for network I/O.

Network I/O Optimization: Routebricks [7] looked into creating fast software routers by scaling out to a number of servers. PacketShader [11] took advantage of low cost GPUs to speed up certain types of network

processing. More recently, PFQ, PF_RING, Intel DPDK and netmap [25, 6, 13, 29] focused on accelerating networking by directly mapping NIC buffers into user-space memory; in this work we leverage the last of these to provide a more direct pipe between NIC and VMs.

Regarding virtualization, work in the literature has looked at improving the performance of Xen networking [28, 35], and we make use of some of the techniques suggested, such as grant re-use. The works in [47, 24] look into modifying scheduling in the hypervisor in order to improve I/O performance; however, the results reported are considerably lower than ClickOS. Finally, Hyper-Switch [15] proposes placing the software switch used to mux/demux packets between NICs and VMs inside the hypervisor. Unfortunately, the switch's data plane relies on open vSwitch code [26], resulting in sub-optimal performance. More recently, two separate efforts have looked into optimizing network I/O for KVM [4] [32]; neither of these has focused on virtualizing middlebox processing, and the rates reported are lower than those in this paper.

Software Middleboxes: Comb [36] introduces an architecture for middlebox deployments targeted at consolidation. However, it does not support multi-tenancy nor isolation, and the performance figures reported (about 4.5Gb/s for two CPU cores assuming maximum-sized packets) are lower than the line-rate results we present in Section 9. The work in [37] uses Vyatta software (see below) to run software middleboxes on Amazon EC2 instances. Finally, while a number of commercial offerings exist (Cisco [5], Vyatta [41]), there are no publicly-available detailed evaluations.

It is worth noting that a preliminary version of this paper has appeared as [20]. This version includes a detailed account of our solution and design decisions, extensive benchmarking as well as implementation and evaluation of a range of ClickOS middleboxes.

4 ClickOS Design

To achieve flexibility, isolation and multi-tenancy, we rely on hypervisor virtualization, which adds an extra software layer between the hardware and the middlebox software which could hurt throughput or increase delay. To minimize these effects, para-virtualization is preferable to full virtualization: para-virtualization makes minor changes to the guest OSes, greatly reducing the overheads inherent in full virtualization such as VM exits [2] or the need for instruction emulation [3].

Consequently, we base ClickOS on Xen [3] since its support for para-virtualized VMs provides the possibility to build a low-delay, high-throughput platform, though its potential is not fulfilled out of the box (Section 6).

Middlebox	Key Click Elements
Load balancer	`RatedSplitter, HashSwitch`
Firewall	`IPFilter`
NAT	`[IP|UDP|TCP]Rewriter`
DPI	`Classifier, IPClassifier`
Traffic shaper	`BandwidthShaper, DelayShaper`
Tunnel	`IPEncap, IPsecESPEncap`
Multicast	`IPMulticastEtherEncap, IGMP`
BRAS	`PPPControlProtocol, GREEncap`
Monitoring	`IPRateMonitor, TCPCollector`
DDoS prevention	`IPFilter`
IDS	`Classifier, IPClassifier`
IPS	`IPClassifier, IPFilter`
Congestion control	`RED, SetECN`
IPv6/IPv4 proxy	`ProtocolTranslator46`

Table 1: Key Click elements that allow developing a wide range of middleboxes.

KVM also supports driver para-virtualization through `virtio` [33], but yields lower performance (Section 6).

Programming Abstractions. Today's software middleboxes are written either as user-space applications on top of commodity OSes (e.g., Snort or Bro) or as kernel changes (e.g., iptables, etc). Either way, C is the de-facto programming language as it offers high performance.

Our platform aims to allow today's middleboxes to run efficiently in the context of virtualization. However, we believe that there are much better ways to develop fast middleboxes. C offers great flexibility but has high development and debugging costs, especially in the kernel. In addition, there is not much software one can reuse when programming a new type of middlebox.

Finding the best programming abstraction for middleboxes is an interesting research topic, but we do not set out to tackle it in this paper. Instead, we want to pragmatically choose the best tool out of the ones we have available today. As a result, we leverage the Click modular router software. Previous work [36] showed that a significant amount of functionality is common across a wide range of middleboxes; Click makes it easy to reuse such functionality, abstracting it into a set of re-usable elements. Click comes with over 300+ stock elements which make it possible to construct middleboxes with minimal effort (Table 1). Finally, Click is extensible, so we are not limited to the functionality provided by the stock elements. Click is of course no panacea: it does not cover all types of middlebox processing, for instance middleboxes that need a full-fledged TCP stack. In such cases it is better to use a standard Linux VM.

Running Click Efficiently: By default, Click runs on top of Linux either as a userland process (with poor performance, see [30]) or as a kernel module. To get domain isolation, we would have to run each Click middlebox inside a Linux virtual machine. This, however, violates our scalability requirement: even stripped down Linux VMs are memory-hungry (128MB or more) and take 5s to boot.

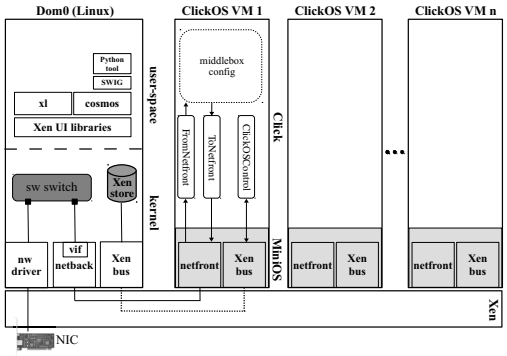


Figure 1: ClickOS architecture.

Instead, we take a step back and ask: *what support does Click need from the operating system to be able to enable a wide range of middlebox processing?* The answer is, surprisingly, not much:

- Driver support to be able to handle different types of network interfaces.
- Basic memory management to allocate different data structures, packets, etc.
- A simple scheduler that can switch between running Click element code and servicing interrupts (mostly from the NICs). Even a cooperative scheduler is enough - there is no need for pre-emptive scheduling, or multi-threading.

The first requirement seems problematic, given the large number of interface vendors and variety of models. Xen elegantly solves this issue through paravirtualization: the guest accesses all NIC types through a single, hardware-agnostic driver connected to the driver domain, and the driver domain (a full-blown Linux machine with the customary driver support) talks to the hardware itself.

Almost all operating systems meet the other two requirements, so there is no need to build one from scratch: we just need an OS that is minimalistic and is able to boot quickly. Xen comes with MiniOS, a tiny operating system that fits the bill and allows us to build efficient, virtualized middleboxes without all of the unnecessary functionality included in a conventional operating system. MiniOS is the basis for our ClickOS VMs.

In short, our ClickOS virtualized middlebox platform consists of (1) a number of optimizations to Xen's network I/O sub-system that allow fast networking for traditional VMs (Section 7); (2) tailor-made middlebox virtual machines based on Click; and (3) tools to build and manage the ClickOS VMs, including inserting, deleting, and inspecting middlebox state (Figure 1).

5 ClickOS Virtual Machines

Before describing what a ClickOS virtual machine is, it is useful to give a brief Xen background. Xen is split into a privileged virtual machine or *domain* called dom0 (typically running Linux), and a set of guest or user domains comprising the users' virtual machines (also known as domUs). In addition, Xen includes the notion of a *driver domain* VM which hosts the device drivers, though in most cases dom0 acts as the driver domain. Further, Xen has a split-driver model, where the back half of a driver runs in a driver domain, the front-end in the guest VM, and communications between the two happen using shared memory and a common, ring-based API. Xen networking follows this model, with dom0 containing a `netback` driver and the guest VM implementing a `netfront` one. Finally, *event channels* are essentially Xen inter-VM interrupts, and are used to notify VMs about the availability of packets.

MiniOS implements all of the basic functionality needed to run as a Xen VM. MiniOS has a single address space, so no kernel/user space separation, and a cooperative scheduler, reducing context switch costs. MiniOS does not have SMP support, though this could be added. We have not done so because a single core is sufficient to support 10 Gbps line-rate real middlebox processing, as we show later. Additionally, we scale up by running many tiny ClickOS VMs rather than a few large VMs using several CPU cores each.

Each ClickOS VM consists of the Click modular router software running on top of MiniOS, but building such a VM image is not trivial. MiniOS is intended to be built with standard GCC and as such we can in principle link any standard C library to it. However, Click is written in `c++`, and so it requires special precautions. The most important of these is that standard `g++` depends on (among others) `ctypes.h` (via `glibc`) which contains Linux specific dependencies that break the standard MiniOS `iostream` libraries. To resolve this we developed a new build tool which creates a Linux-independent `c++` cross-compiler using `newlibc` [38].

In addition, our build tool re-designs the standard MiniOS toolchain so that it is possible to quickly and easily build arbitrary, MiniOS-based VMs by simply linking an application's entry point so that it starts on VM boot; this is useful for supporting middleboxes that cannot be easily supported by Click. Regarding libraries, we have been conservative in the number of them we link, and have been driven by need rather than experimentation. In addition to the standard libraries provided with the out-of-the-box MiniOS build (`lwip, zlib, libpci`) we add support for `libpcre, libpcap and libssl`, libraries that certain Click elements depend on. The result is a ClickOS image with 216/282 Click elements, with many of the remaining ones requiring a filesystem to run, which we plan to add.

Once built, booting a ClickOS image start by creating the virtual machine itself, which involves reading its configuration, the image file, and writing a set of entries

to the Xen store, a `proc`-like database residing in dom0 that is used to share control information with the guest domains. Next, we attach the VM to the back-end switch, connecting it to physical NICs.

MiniOS boots, after which a special control thread is created. At this point, the control thread creates an *install* entry in the Xen store to allow users to install Click configurations in the ClickOS VM. Since Click is designed to run on conventional OSes such as Linux or FreeBSD which, among other things, provide a console through which configurations can be controlled and, given that MiniOS does not provide these facilities, we leverage the Xen store to emulate such functionality.

Once the install entry is created, the control thread sets up a watch on it that monitors changes to it. When written to, the thread launches a second MiniOS thread which runs a Click instance, allowing several Click configurations to run within a single ClickOS VM. To remove the config we write an empty string to the Xen store entry.

We also need to support Click *element handlers*, which are used to set and retrieve state in elements (e.g, the `AverageCounter` element has a read counter to get the current packet count and a write one to reset the count); to do so, we once again leverage the Xen store. For each VM, we create additional entries for each of the elements in a configuration and their handlers. We further develop a new Click element called `ClickOSControl` which gets transparently inserted into all configurations. This element takes care of interacting, on one end, with the read and write operations happening on the Xen store, and communicating those to the corresponding element handlers within Click.

In order to control these mechanisms which are not standard to all Xen VMs, ClickOS comes with its own dom0 CLI called Cosmos (as opposed to the standard, Xen-provided `xl` tool). Cosmos is built directly on top of the Xen UI libraries (Figure 1) and therefore does not incur any extraneous costs when processing requests. To simplify development and user interaction, Cosmos implements a SWIG [39] wrapper enabling users to automatically generate Cosmos bindings for any of the SWIG supported languages. For convenience, we have also implemented a Python-based ClickOS CLI.

Finally, it is worth mentioning that while MiniOS represents a low-level development environment, programming for ClickOS is relatively painless: development, building and testing can take place in user-space Click, and the resulting code/elements simply imported into the ClickOS build process when ready.

6 Xen Networking Analysis

In this section we investigate where the Xen networking bottlenecks are. Figure 1 illustrates the Xen network I/O sub-system: the network driver, software switch, virtual interface and netback driver in dom0 and the netfront driver (either the Linux or MiniOS one) in the guest domains, any of which could be bottlenecks.

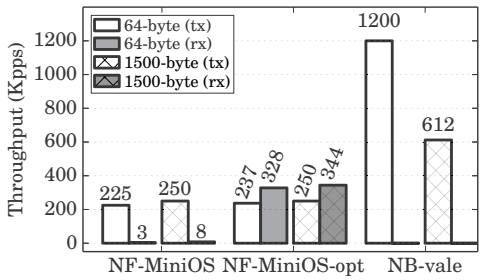


Figure 2: Xen performance bottlenecks using a different back-end switch and netfront (NF) and netback (NB) drivers ("opt" stands for optimized).

In order to get some baseline numbers, we begin by performing a simple throughput test. For this test we used a server with an Intel Xeon E3-1220 3.1GHz 4-core CPU, 16GB memory and an Intel x520-T2 dual Ethernet port 10Gb/s card (about $1,500 including the NIC). The server had Xen 4.2, Open vSwitch as its back-end switch and a single ClickOS virtual machine. The VM was assigned a single CPU core, the remainder given to dom0.

The first result (labeled "NF-MiniOS" in Figure 2) shows the performance of the MiniOS netfront driver when sending (Tx, in which case we measure rates at the netback driver in dom0) and receiving (Rx) packets. Out of the box, the MiniOS netfront driver yields poor rates, especially for Rx, where it can barely handle 8 Kp/s.

To improve this receive rate, we modified the netfront driver to re-use memory grants. Memory grants are Xen's mechanism to share memory between two virtual machines, in this case the packet buffers between dom0 and the ClickOS VM. By default, the driver requests a grant for *each* packet, requiring an expensive *hypercall* to the hypervisor (essentially the equivalent of a system call for an OS); we changed the driver so that it receives the grants for packet buffers at initialization time, and to re-use these buffers for all packets handled. The driver now also uses polling, further boosting performance.

The results are labeled "NF-MiniOS-opt" in Figure 2. We see important improvements in Rx rates, from 8 Kp/s to 344 Kp/s for maximum-sized packets. Still, this is far from the 10Gb/s line rate figure of 822 Kp/s, and quite far from the 14.8 Mp/s figure for minimum-sized packets, meaning that other significant bottlenecks remain.

Next, we took a look at the software switch. By default, Xen uses Open vSwitch, which previous work reports as capping out at 300 Kp/s [30]. As a result, we decided to replace it with the VALE switch [31]. Because VALE ports communicate using the netmap API,

description	function	ns
get vif	poll_net_schedule_list	119
handle frags if any	netbk_count_requests	53
alloc skb	alloc_skb reserve_skb	384
alloc page for packet data	xen_netbk_alloc_page	293
build grant op struct	fills *gnttab_copy*	96
extends the skb with the expected size	__skb_put	96
build grant op struct (for frags)	xen_netbk_get_requests	61
add the skb to the Tx queue	__skb_queue_tail	53
checks for packets received	check_rx_xenvif	206
packet grant copy	HYPERCALL	24708
dequeue packet from Tx queue	__skb_dequeue	94
copy pkt data to skb	memcpy	90
put a response in the ring	fills *xen_netif_tx_response* notify_via_remote_irq	52
copy frag data	xen_netbk_fill_frags	179
calc checksum	checksum_setup	78
forward pkt to bridge	xenvif_receive_skb	3446

Table 2: Per-function netback driver costs when sending a batch of 32 packets. Small or negligible costs are not listed for readability. Timings are in nanoseconds.

we modified the netback driver to implement that API, and removed the Xen virtual interface (*vif*) in the process. These changes ("NB-vale") gave a noticeable boost of up to 1.2 Mp/s for 64B packets, confirming that the switch was at least partly to blame [2].

Despite the improvement, the figures were still far from line rate speeds. Sub-optimal performance in the presence of a fast software switch, no *vif* and an optimized netfront driver seem to point to issues in the netback driver, or possibly in the communication between netback and netfront drivers. To dig in deeper, we carried out a per-function analysis of the netback driver to determine where the major costs were coming from.

The results in Table 2 report the main costs in the code path when transmitting a batch of 32 packets. We obtain timings via the `getnstimeofday()` function, and record them using the `trace_printk` function from the lightweight FTrace tracing utility.

The main cost, as expected, comes from the hypercall, essentially a system call between the VM and the hypervisor. Clearly this is required, though its cost can be significantly amortized by techniques such as batching. The next important overhead comes from transmitting packets from the netback driver through the *vif* and onto the switch. The *vif*, basically a tap device, is not fundamental to having a VM communicate with the netback driver

[2]We did not implement Rx on this modified netback driver as the objective was to see if the only remaining major bottleneck was the software switch.

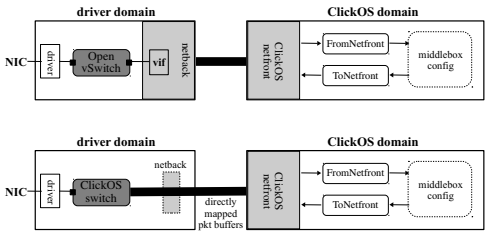


Figure 3: Standard Xen network I/O pipe (top) and our optimized, ClickOS one with packet buffers directly mapped into the VM's memory space.

and switch, but as shown adds non-negligible costs arising from extra queuing and packet copies. Other further penalties come from using the Xen ring API, which for instance requires responses to all packets transmitted in either direction. Finally, a number of overheads are due to `sk_buff` management, not essential to having a VM transmit packets to the network back-end – especially a non-Linux VM such as ClickOS.

In the next section we discuss how we revamped the Xen I/O network pipe in order to remove or alleviate most of these costs.

7 Network I/O Re-Design

The Xen network I/O pipe has a number of components and mechanisms that add overhead but that are not fundamental to the task of getting packets in and out of VMs. In order to optimize this, it would be ideal if we could have a more direct path between the back-end NIC and switch and the actual VMs. Conceptually, we would like to directly map ring packet buffers from the device driver or back-end switch all the way into the VMs' memory space, much like certain fast packet I/O frameworks do between kernel and user-space in non-virtualized environments [29, 25, 6].

To achieve this, and to boost overall performance, we take three main steps. First, we replace the standard but sub-optimal Open vSwitch back-end switch with a high-speed, ClickOS switch; this switch exposes per-port ring packet buffers which are able to we map into a VM's memory space. Second, we observe that since in our model the ClickOS switch and netfront driver transfer packets between one another directly, the netback driver becomes redundant. As a result, we remove it from the pipe, but keep it as a control plane driver to perform actions such as communicating ring buffer addresses (grants) to the netfront driver. Finally, we changed the VM netfront driver to map the ring buffers into its memory space.

These changes are illustrated in Figure 3, which contrasts the standard Xen network pipe (top diagram) with ours (bottom). We dedicate the rest of this section to providing a more detailed explanation of our optimized

switch, netback and netfront drivers (both MiniOS' and the Linux one) and finally a few modifications to Click.

ClickOS Switch. Given the throughput limitations of Xen's standard Open vSwitch back-end switch, we decided to replace it with the VALE high-speed switch [18], and to extend its functionality in a number of ways. First, VALE only supports virtual ports, so we add the ability to connect NICs directly to the switch. Second, we increase the maximum number of ports on the switch from 64 to 256 so as to accommodate a larger number of VMs.

In addition, we add support for each individual VM to configure the number of slots in the packet buffer ring, up to a maximum of 2048 slots. As we will see in the evaluation section, larger ring sizes can improve performance at the cost of larger memory requirements.

Finally, we modify the switch so that its switching logic is modular, and replace the standard learning bridge behavior with static MAC address-to-port mappings to boost performance (since in our environment we are in charge of assigning MAC addresses to the VMs this change does not in any way limit our platform's functionality). All of these changes have been now upstreamed into VALE's main code base.

Netback Driver. We redesign the netback driver to turn it (mostly) into a control-plane only driver. Our modified driver is in charge of allocating memory for the receive and transmit packet rings and their buffers and to set-up memory grants for these so that the VM's netfront driver can map them into its memory space. We use the Xen store to communicate the rings' memory grants to the VMs, and use the rings themselves to tell the VM about the ring buffers' grants; doing so ensures that the numerous grants do not overload the Xen store.

On the data plane side, the driver is only in charge of (1) setting up the `kthreads` that will handle packet transfers between switch and netfront driver; and (2) proxy event channel notifications between the netfront driver and switch to signal the availability of packets.

We also make a few other optimizations to the netback driver. Since the driver is no longer involved with actual packet transfer, we no longer use *vifs* nor OS-specific data structures such as `sk_buffs` for packet processing. Further, as suggested in [46], we adopt a 1:1 model for mapping kernel threads to CPU cores: this avoids unfairness issues. Finally, the standard netback uses a single event channel (a Xen interrupt) for notifying the availability of packets for both transmit and receive. Instead, we implement separate Tx and Rx event channels that can be serviced by different cores.

Netfront Driver. We modify MiniOS' netfront driver to be able to map the ring packet buffers exposed by the ClickOS switch into its memory space. Further, since the switch uses the netmap API [29], we implement a netmap module for MiniOS. This module uses the standard netmap data structures and provides the same abstractions as user-space netmap: `open`, `mmap`, `close` and finally `poll` to transmit/receive packets.

Beyond these mechanisms, our netfront driver includes a few other changes

- **Asynchronous Transmit:** In order to speed up transmit throughput, we modify the transmit function to run asynchronously.
- **Grant Re-Use:** Unlike the standard MiniOS netfront driver, we set-up grants once, and re-use them for the lifetime of the VM. This is a well-known technique for improving the performance of Xen's network drivers [35].
- **Linux Support:** While our modifications result in important performance increases, the departure from the standard Xen network I/O model means that we break support for other, non-MiniOS guests. To remedy this, we implemented a new Linux netfront driver suited to our optimized network pipe. Using this new netfront results in 10 Gb/s rates for most packet sizes (see Section 8) and allows us to run, at speed, any remaining middleboxes that cannot be easily implemented in Click or on top of MiniOS.

Click Modifications. Finally, we have made a few small changes to Click (version 2.0.1, less than 50 lines of code), including adding new elements to send and receive packets via the netfront driver, and optimizations to the `InfiniteSource` element to allow it to reach high packet rates.

ClickOS Prototype. The ClickOS prototype is open-source software. It includes changes to the XEN back-end (around 1000 LoC) and the frontend (1200 LoC). We are beginning to upstream these changes to Xen, but this process is lengthy; in the meantime, we plan to make the code available so that prospective users can just download our patches and recompile the netback and netfront modules (or recompile the dom0 kernel altogether).

8 Base Evaluation

Having presented the ClickOS architecture, its components and their optimization, we now provide a thorough base evaluation of the system. After this, in Section 9, we will describe the implementation of several middleboxes as well as performance results for them.

Experimental Set-up. The ClickOS tests in this section were conducted using either (1) a *low-end*, single-CPU Intel Xeon E3-1220 server with 4 cores at 3.1 GHz and 16 GB of DDR3-ECC RAM (most tests); or (2) a *mid-range*, single-CPU Intel Xeon E5-1650 server with 6 cores at 3.2 GHz and 16 GB of DDR3-ECC RAM (switch and scalability tests). In all cases we used Linux

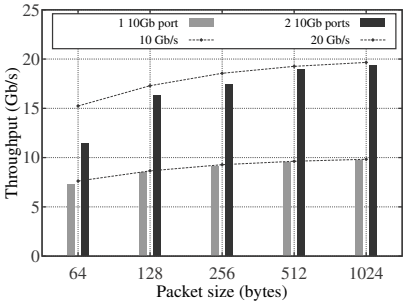


Figure 4: ClickOS switch performance using one and two 10 Gb/s NIC ports.

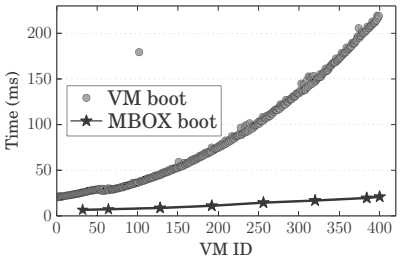


Figure 5: Time to create and boot 400 ClickOS virtual machines in sequence and to boot a Click configuration within each of them.

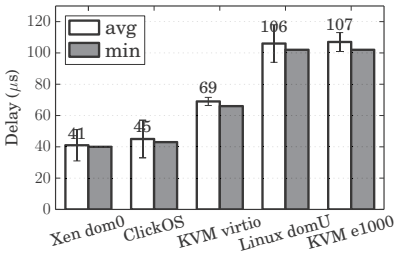


Figure 6: Idle VM ping delays for ClickOS, a Linux Xen VM, dom0, and KVM using the e1000 or virtio drivers.

3.6.10 for dom0 and domU, Xen 4.2.0, Click 2.0.1 and netmap's `pkt-gen` application for packet generation and rate measurements. All packet generation and rate measurements on an external box are conducted using one or more of the low-end servers, and all NICs are connected through direct cables. For reference, 10Gb/s equates to about 14.8 Mp/s for minimum-sized packets and 822 Kp/s for maximum-sized packets.

ClickOS Switch. The goal is to ensure that the switching capacity is high so that it does not become a bottleneck as more ClickOS VMs, cores and NICs are added to the system.

For this test we rely on a Linux (i.e., non-Xen) system. We use a user-space process running `pkt-gen` to generate packets towards the switch, and from there onto a single 10 Gb/s Ethernet port; a separate, low-end server then uses `pkt-gen` once again to receive the packets and to measure rates. We then add another `pkt-gen` user-process and 10Gb/s Ethernet port to test scalability. Each `pkt-gen`/port pair uses a single CPU core (so two in total for the 20Gb/s test).

For the single port pair case, the switch saturated the 10Gb/s pipe for all packet sizes (Figure 4). For the two port pairs case, the switch fills up the entire cumulative 20Gb/s pipe for all packet sizes except minimum-sized ones, for which it achieves 70% of line rate. Finally, we also conducted receive experiments (where packets are sent from an external box towards the system hosting the switch) which resulted in roughly similar rates.

Memory Footprint. As stated previously, the basic memory footprint of a ClickOS image is 5MB (including all the supported Click elements). In addition to this, a certain amount of memory is needed to allocate the netmap ring packet buffers. How much memory depends on the size of the rings (i.e., how many slots or packets the ring can hold at a time), which can be configured on a per-ClickOS VM basis.

To get an idea of how much memory might be required, Table 3 reports the memory requirements for different ring sizes, ranging from kilobytes for small rings all the way up to 8MB for a 2048-slot ring. As we will see later on in this section, this is a trade-off between the higher throughput that can be achieved with larger rings and the larger number of VMs that can be concurrently run when using small ring sizes. Ultimately, it might be unlikely that a single ClickOS VM will need to handle very large packet rates, so in practice a small ring size might suffice. It is also worth pointing out that larger rings require more memory grants; while there is a maximum number of grants per VM that a Xen system can have, this limit is configurable at boot time.

Ring size	Required memory (KB)	# of grants
64	264	65
128	516	129
256	1032	258
512	2064	516
1024	4128	1032
2048	8260	2065

Table 3: Memory requirements for different ring sizes.

What about the state that certain middleboxes might contain? To get a feel for this, we inserted 1,000 forwarding rules into an IP router, 1,000 rules into a firewall and 400 into an IDS (see Section 9 for a description of these middleboxes); the memory consumption from this was 20KB, 87KB and 30KB, respectively, rather small amounts. All in all, even if we use large ring sizes, a ClickOS VM requires approximately 15MB of memory.

Boot Times. In this set of tests we use the Cosmos tool to create ClickOS VMs and measure how long it takes for them to boot. A detailed breakdown of the ClickOS boot process may be found in [20]; for brevity, here we provide a summary. During boot up most of the time is spent issuing and carrying out the hypercall to create the VM (5.2 milliseconds), building the image (7.1 msecs) and creating the console (4.4 msecs), for a total of about 20.8 msecs. Adding roughly 1.4 msecs to attach the VM to the back-end switch and about 6.6 msecs to install a Click configuration brings the total to about 28.8 msecs

from when the command to create the ClickOS VM is issued until the middlebox is up and running.

Next we measured how booting large numbers of ClickOS VMs on the same system affects boot times. For this test we boot an increasing number of VMs in sequence and measure how long it takes for each of them to boot and install a Click configuration (Figure 5). Both the boot and startup times increase with the number of VMs, up to a maximum of 219 msecs boot and 20.0 msecs startup for the 400th VM. This increase is due to contention on the Xen store and could be improved upon.

Delay. Most middleboxes are meant to work transparently with respect to end users, and as such, should introduce little delay when processing packets. Virtualization technologies are infamous for introducing extra layers and with them additional delay, so we wanted to see how ClickOS' streamlined network I/O pipe would fare.

To set-up the experiment, we create a ClickOS VM running an ICMP responder configuration based on the `ICMPPingResponder` element. We use an external server to ping the ClickOS VM and measure RTT. Further, we run up to 11 other ClickOS VMs that are either idle, performing a CPU-intensive task (essentially an infinite loop) or a memory-intensive-one (repeatedly allocating and deallocating several MBs of memory).

The results show low delays of roughly 45 μsecs for the test with idle VMs, a number that stays fairly constant as more VMs are added. For the memory intensive task test the delay is only slightly worse, starting again at 45 μsecs and ramping up to 64 μsecs when running 12 VMs. Finally, the CPU intensive task test results in the largest delays (RTTs of up to 300 μsecs), though these are still small compared to Internet end-to-end delays.

Next, we compared ClickOS' idle delay to that of other systems such as KVM and other Xen domains (Figure 6). Unsurprisingly, dom0 has a small delay of 41 μsecs since it does not incur the overhead of going through the netback and netfront drivers. This overhead does exist when measuring delay for the standard, unoptimized netback/netfront drivers of a Xen Linux VM (106 μsecs). KVM, in comparison, clocks in at 69 μsecs when using its para-virtualized `virtio` drivers and 107 μsecs for its virtualized e1000 driver.

Throughput. In the next batch of tests we perform a number of baseline measurements to get an understanding of what packet rates ClickOS can handle. All of these tests are done on the low-end servers, with one CPU core dedicated to the VM and the remaining three to dom0.

Before testing a ClickOS VM we would like to benchmark the underlying network I/O pipe, from the NIC through to the back-end switch, netback driver and the netfront one. To do so, we employ our build tool to create a special VM consisting of only MiniOS and `pkt-gen` on top of it. After MiniOS boots, `pkt-gen` begins to immediately generate packets (for Tx tests) or measure rates (Rx). We conduct the experiment for different ring sizes (set using a `sysctl` command to the netmap kernel module) and for different packet sizes (for Tx tests this is set via Cosmos before the VM is created).

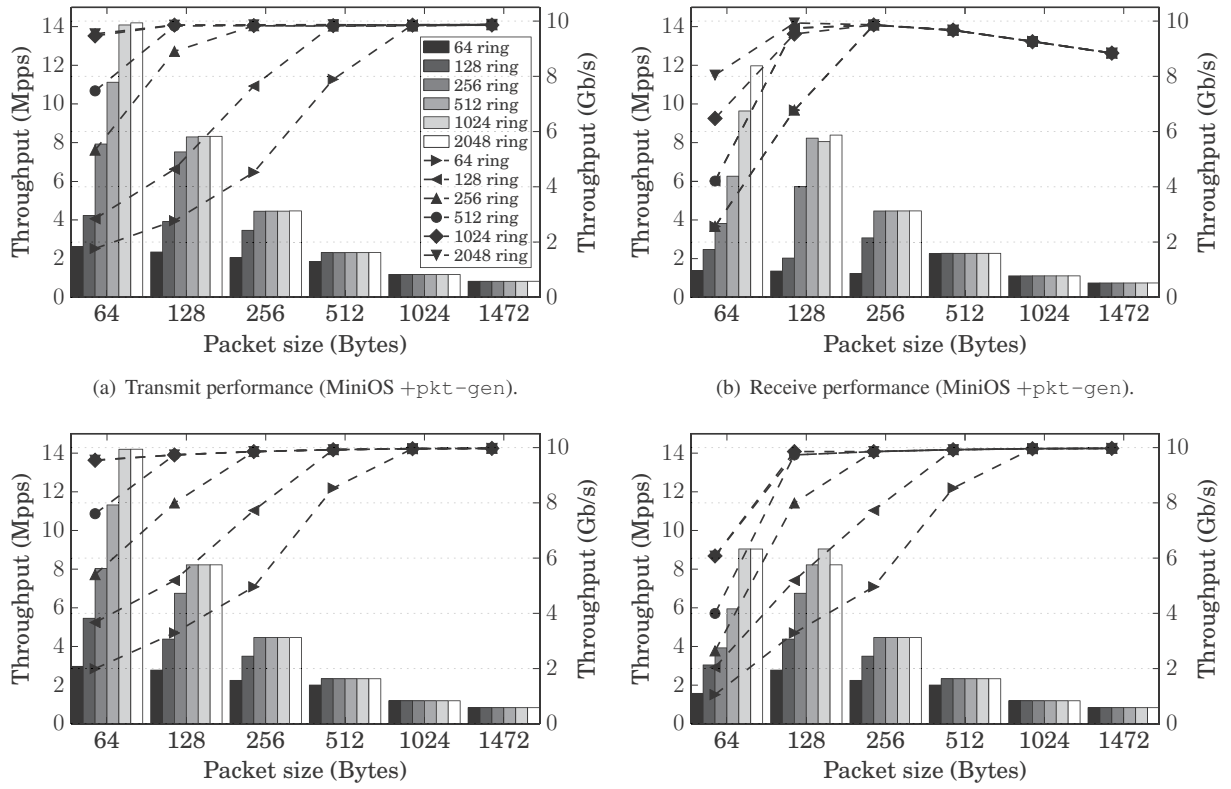
Figure 7 reports the results of the measurements. On transmit, the first thing to notice is that our optimized I/O pipe achieves close to line rate for minimum-sized packets (14.2 Mp/s using 2048-slot rings out of a max of 14.8 Mp/s) and line rate for all other sizes. Further, ring size matters, but mostly for minimum-sized packets. The receive performance is also high but somewhat lower due to extra queuing overheads at the netfront driver.

With these rates in mind, we proceed to deriving baseline numbers for ClickOS itself. In this case, we use a simple Click configuration based on the `AverageCounter` element to measure receive rates and another one based on our modified `InfiniteSource` to generate packets. Figure 7(c) shows ClickOS' transmit performance, which is comparable to that produced by the pkt-gen VM, meaning that at least for simple configurations ClickOS adds little overhead. The same is true for receive, except for minimum-sized packets, where the rate drops from about 12.0 Mp/s to 9.0 Mp/s.

For the last set of throughput tests we took a look at the performance of our optimized Linux domU netfront driver, comparing it to that of a standard netfront/Linux domU and KVM. For the latter, we used Linux version 3.6.10, the emulated e1000 driver, Vhost enabled, the standard Linux bridge, and `pkt-gen` once again to generate and measure rates. As seen in Figure 8 the Tx and Rx rates for KVM and the standard Linux domU are fairly similar, reaching only a fraction of line rate for small packet sizes and up to 7.88 Gb/s (KVM) and 6.46 Gb/s (Xen) for maximum-sized ones. The optimized netfront/Linux domU, on the other hand, hits 8.53 Mp/s for Tx and 7.26 Mp/s for Rx for 64-byte frames, and practically line rate for 256-byte packets and larger.

State Insertion. In order for our middlebox platform to be viable, it has to allow the middleboxes running on it to be quickly configured. For instance, this could involve inserting rules into a firewall or IDS, or adding extra external IP addresses to a carrier-grade NAT. In essence, we would like to test the performance of ClickOS element handlers and their use of the Xen store to communicate state changes. In this test we use Cosmos to perform a large number of reads and writes to a dummy ClickOS element with handlers, and measure how long these take for different transaction sizes (i.e., the number of bytes in question for each read and write operation).

Figure 9 reports read times of roughly 9.4 msecs and writes of about 0.1 msecs, numbers that fluctuate little



(a) Transmit performance (MiniOS `+pkt-gen`).

(b) Receive performance (MiniOS `+pkt-gen`).

(c) Transmit performance (ClickOS).

(d) Receive performance (ClickOS).

Figure 7: Performance of a single VM `pkt-gen` running on top of MiniOS/ ClickOS on a single CPU core, when varying the number of ring slots. The line graphs correspond to the right-hand y-axis.

across different transaction sizes. Note that read takes longer since it basically involves doing a write, waiting for the result, and then reading it. However, the more critical operation for middleboxes should be write, since it allows state insertion and deletion. For completeness, we also include measurements when using the XEN python API; in this case, the read and write operations jump to 10.1 and 0.3 msecs, respectively.

Chaining. Is it quite common for middleboxes to be chained one after the other in operator networks (e.g., a firewall followed by an IDS). Given that ClickOS has the potential to host large numbers of middleboxes on the same server, we wanted to measure the system's performance when chaining different numbers of middleboxes back-to-back. In greater detail, we instantiate one ClickOS VM to generate packets as fast as possible, another one to measure them, and an increasing number of intermediate ClickOS VMs to simply forward them. As with other tests, we use a single CPU core to handle the VMs and assign the rest to dom0.

As expected, longer chains result in lower rates, from 21.7 Gb/s for a chain of length 2 (just a generator VM and the VM measuring the rate) all the way down to 3.1 Gb/s for a chain with 9 VMs (Figure 10). Most of the decrease is due to the single CPU running the VMs being overloaded, but also because of the extra copy operations in the back-end switch and the load on dom0. The former could be alleviated with additional CPU cores; the latter by having multiple switch instances (which our switch supports) or driver domains (which Xen does).

Scaling Out. In the final part of our platform's base evaluation we use our mid-range server to test how well ClickOS scales out with additional VMs, CPU cores and 10 Gb/s NICs. For the first of these, we instantiate an increasing number of ClickOS VMs, up to 100 of them. All of them run on a single CPU core and generate packets as fast as possible towards an outside box which measures the cumulative throughput. In addition, we measure the *individual* contribution of each VM towards the cumulative rate in order to ensure that the platform is fairly scheduling the VMs: all of VMs contribute equally to the rate and that none are starved.

Figure 11 plots the results. Regardless of the number of VMs, we get a cumulative throughput equivalent to line rate for 512-byte packets and larger and a rate of 4.85 Mp/s for minimum-sized ones. The values on top of the bars represent the standard deviation for all the individual

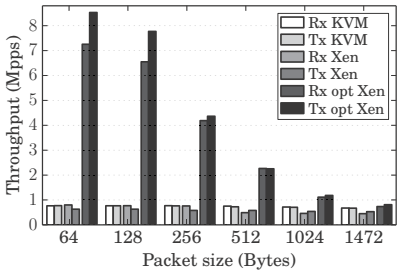


Figure 8: Linux domU performance with an optimized (opt) netmap-based netfront driver versus the performance of out-of-the-box Xen and KVM Linux virtual machines.

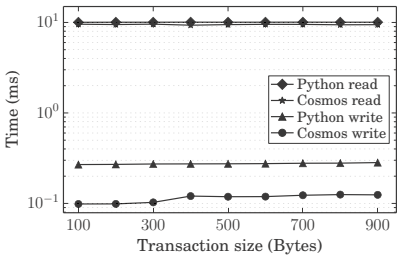


Figure 9: ClickOS middlebox state insertion (write) and retrieval (read) for different transaction sizes (log scale).

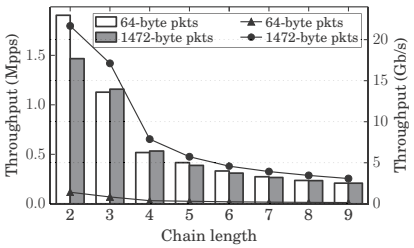


Figure 10: Performance when chaining ClickOS VMs back-to-back. The first VM generates packets, the ones in the middle forward them and the last one measures rates. Ring size is set to 64 slots.

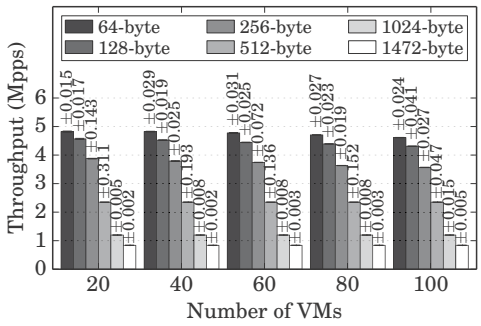


Figure 11: Running many ClickOS packet generator VMs on one core and a 10 Gb/s port. Fairness is shown by the low standard deviations above the the bars.

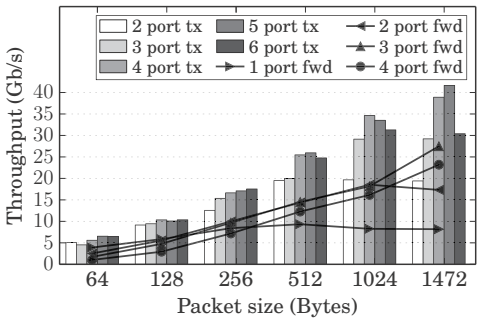


Figure 12: Cumulative throughput when using multiple 10 Gb/s ports and one ClickOS VM per port to (1) send out traffic (tx) or (2) forward traffic (fwd).

rates contributed by each VM; the fact that these values are rather low confirms fairness among the VMs.

Next, we test ClickOS' scalability with respect to additional CPU cores and 10 Gb/s ports. We use one packet generator ClickOS VM per port, up to a maximum of six ports. In addition, we assign two cores to dom0 and the remaining four to the ClickOS VMs in a round-robin fashion. Each pair of ports is connected via direct cables to one of our low-end servers and we calculate the cumulative rate measured at them; ring size is 1024.

For maximum-sized packets we see a steady, line-rate increase as we add ports, VMs and CPU cores, up to 4 ports (Figure 12). After this point, VMs start sharing cores (our system has six of them, with four of them assigned to the VMs) and the performance no longer scales linearly. For the final experiment we change the configuration that the ClickOS VMs are running from a packet generator to one that bounces packets back onto the same interface that they came on (line graphs in Figure 12). In this configuration, ClickOS rates go up to 27.5 Gb/s.

Scaling these experiments further requires a CPU with more cores than in our system, or adding NUMA support to ClickOS so that performance scales linearly with additional CPU packages; the latter is our future work.

9 Middlebox Implementations

Having evaluated the baseline performance of ClickOS, we now turn our attention to evaluating its performance when running actual middleboxes. Clearly, since the term middleboxes covers a wide range of processing, exhaustively testing them all is impossible. We therefore evaluate the performance of ClickOS on a set candidate middleboxes which vary in the type of workload they generate.

For these set of tests we use two of our low-end servers connected via two direct cables, one per pair of Ethernet ports. One of the servers generates packets towards the other server, which runs them through a ClickOS middlebox and forwards them back towards the first server where their rate is measured. The ClickOS VM is assigned a single CPU core, with the remaining three given to dom0. We test each of the following middleboxes:

Wire (WR): A simple "middlebox" which sends packets from its input to its output interface. This configuration serves to give a performance baseline.

EtherMirror (EM): Like wire, but also swap the Ethernet source and destination fields.

IP Router (IR): A standards-compliant IPv4 router configured with a single rule.

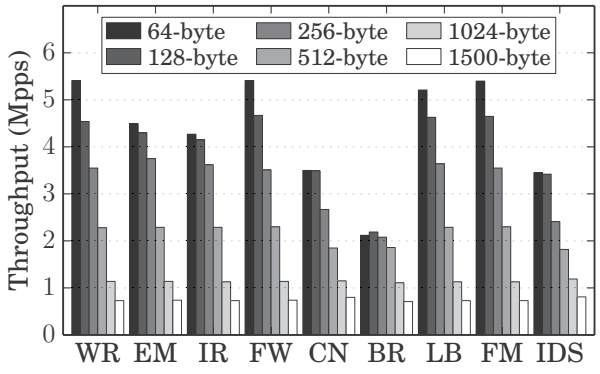


Figure 13: Performance for different ClickOS middleboxes and packet sizes using a single CPU core.

Firewall (FW): Based on the `IPFilter` element and configured with ten rules, none matching any packets.
Carrier Grade NAT (CN): An almost standards-compliant carrier-grade NAT. To stress the NAT, each packet has a different set of source and destination port numbers. Using a single flow/set of ports results in a higher rate of 5.1 Mp/s for minimum-sized packets.
Software BRAS (BR): An implementation of a Broadband Remote Access Server (BRAS), including PPPoE session handling. The data plane checks session numbers and PPPoE/PPP message types, strips tunnel headers, and performs IP lookup and MAC header re-writing.
Intrusion Detection System (IDS): A simple Intrusion Detection System based on regular expression matching. The reported results are for a single rule that matches the incoming packets.
Load Balancer (LB): This re-writes packet source MAC addresses in a round-robin fashion based on the IP src/dst, port src/dst and type 5-tuple in order to split packets to different physical ports.
Flow Monitor (FM) retains per flow (5-tuple) statistics.

Figure 13 reports throughput results for the various middleboxes. Overall, ClickOS performs well, achieving almost line rate for all configurations for 512-byte and larger packets (the BRAS and CG-NAT middleboxes have rates slightly below the 2.3 Mp/s line rate figure). For smaller packet sizes the percentage of line rate drops, but ClickOS is still able to process packets in the millions/second.

To get and idea of how this relates to a real-world traffic matrix, compare this to an average packet size of 744 bytes reported by a recent study done on a tier-1 OC192 (about 10Gb/s) backbone link [34]: if we take our target to be packets of around this size, all middleboxes shown can sustain line rate.

Naturally, some of these middleboxes fall short of being fully functional, and different configurations (e.g., a large number of firewall rules) would cause their performance to drop from what we present here. Still, we believe these figures to be high enough to provide a sound basis upon which to build production middleboxes. The carrier-grade NAT, for instance, is proof of this: it is fully functional, and in stress tests it is still able to handle packets in the millions/second.

10 Conclusions

This paper has presented ClickOS, a Xen-based virtualized platform optimized for middlebox processing. ClickOS can turn Network Function Virtualization into reality: it runs hundreds of middleboxes on commodity hardware, offers millions of packets per second processing speeds and yields low packet delays. Our experiments have shown that a low-end server can forward packets at around 30Gb/s.

ClickOS is proof that software solutions alone are enough to significantly speed up virtual machine processing, to the point where the remaining overheads are dwarfed by the ability to safely consolidate heterogeneous middlebox processing onto the same hardware. ClickOS speeds up networking for all Xen virtual machines by applying well known optimizations including reducing the number of hypercalls, use of batching, and removing unnecessary software layers and data paths.

The major contribution of ClickOS is adopting Click as the main programming abstraction for middleboxes and creating **a tailor-made guest operating system to run Click configurations**. Such specialization allows us to optimize the runtime of middleboxes to the point where they boot in milliseconds, while allowing us to support a wide range of functionality. Our implementations of a software BRAS and a Carrier-Grade NAT show that ClickOS delivers production-level performance when running real middlebox functionality.

In the end, we believe that ClickOS goes beyond replacing hardware middleboxes with the software equivalent. Small, quick-to-boot VMs make it possible to offer personalized processing (e.g., firewalls) to a large number of users with comparatively little hardware. Boot times in the order of milliseconds allow fast scaling of processing dynamically (e.g., in response to a flash crowd) as well as migration with negligible down-time. Finally, ClickOS could help with testing and deployment of new features by directing subsets of flows to VMs running experimental code; issues with the features would then only affect a small part of the traffic, and even VMs crashing would not represent a major problem since they could be re-instantiated in milliseconds.

Acknowledgments

The authors are in debt to Adam Greenhalgh for initial ideas and work towards ClickOS. The research leading to these results was partly funded by the EU FP7 CHANGE project (257322).

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *USENIX Conference*, pages 93–112, 1986.

[2] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. ACM SOSP, 2003*, New York, NY, USA, 2003. ACM.

[4] A. Cardigliano, L. Deri, J. Gasparakis, and F. Fusco. vpf_ring: towards wire-speed network monitoring using virtual machines. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 533–548, New York, NY, USA, 2011. ACM.

[5] Cisco. Cisco Cloud Services Router 1000v Data Sheet. `http://www.cisco.com/en/US/prod/collateral/routers/ps12558/ps12559/data_sheet_c78-705395.html`, July 2012.

[6] L. Deri. Direct NIC Access. `http://www.ntop.org/products/pf_ring/dna/`, December 2011.

[7] 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, pages 15–28, New York, NY, USA, 2009. ACM.

[8] ETSI. Leading operators create ETSI standards group for network functions virtualization. `http://www.etsi.org/index.php/news-events/news/644-2013-01-isg-nfv-created`, September 2013.

[9] ETSI Portal. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`, October 2012.

[10] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM.

[11] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[12] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *Proc. ACM IMC*, 2011.

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

[14] Intel. Intel Virtualization Technology for Connectivity. `http://www.intel.com/content/www/us/en/network-adapters/virtualization.html`, September 2013.

[15] M. C. Kaushik Kumar Ram, Alan L. Cox and S. Rixner. Hyper-switch: A scalable software virtual switching architecture. In *Proc. of USENIX Annual Technical Conference*, 2013.

[16] A. Kivity, Y. Kamay, K. Laor, U. Lublin, and A. Liguori. Kvm: The linux virtual machine monitor. In *Proc. of the Linux Symposium*, 2007.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems, August 2000*, 2000.

[18] Luigi Rizzo. VALE, a Virtual Local Ethernet. `http://info.iet.unipi.it/~luigi/vale/`, July 2012.

[19] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, Mar. 2013.

[20] J. Martins, M. Ahmed, C. Raiciu, and F. Huici. Enabling fast, dynamic network processing with clickos. In *HotSDN*, pages 67–72, 2013.

[21] Microsoft Corporation. Microsoft Hyper-V Server 2012. `http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx`, September 2013.

[22] Minix3. Minix3. http://www.minix3.org/, July 2012.

[23] MIT Parallel and Distributed Operating Systems Group. MIT Exokernel Operating System. http://pdos.csail.mit.edu/exo.html, March 2013.

[24] Nadav HarEl and Abel Gordon and Alex Landau and Muli Ben-Yehuda and Avishay Traeger and Razya Ladelsky. Efcient and Scalable Paravirtual I/O System. In *Proc. of USENIX Annual Technical Conference*, 2013.

[25] N.Bonelli, A. D. Pietro, S. Giordano, and G. Procissi. On multi–gigabit packet capturing with multi–core commodity hardware. In *Passive and Active Measurement conference (PAM)*, 2012.

[26] Open vSwitch. Production Quality, Multilayer Open Virtual Switch. http://openvswitch.org/, March 2013.

[27] Openvz.org. OpenVZ Linux Containers. http://openvz.org/Main_Page, September 2013.

[28] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 61–70, New York, NY, USA, 2009. ACM.

[29] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. USENIX Annual Technical Conference*, 2012.

[30] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In A. G. Greenberg and K. Sohraby, editors, *INFOCOM*, pages 2471–2479. IEEE, 2012.

[31] L. Rizzo and G. Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[32] L. Rizzo, G. Lettieri, and V. Maffione. Speeding up packet i/o in virtual machines. In *Proc. ACM/IEEE ANCS*, 2013.

[33] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[34] P. M. Santiago del Rio, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil. Wire-speed statistical classification of network traffic on commodity hardware. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, IMC '12, pages 65–72, New York, NY, USA, 2012. ACM.

[35] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[36] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[37] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratsanamy, and V. Sekarl. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM*, 2012.

[38] Sourceware.org. The Newlib Homepage. http://sourceware.org/newlib/, September 2013.

[39] Swig.org. Simplified Wrapper and Interface Generator. http://swig.org, September 2013.

[40] VMware. VMware Virtualization Software for Desktops, Servers and Virtual Machines for Public and Private Cloud Solutions. http://www.vmware.com, July 2012.

[41] Vyatta. The Open Source Networking Community. http://www.vyatta.org/, July 2012.

[42] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, Dec. 2002.

[43] Wikipedia. L4 microkernel family. http://en.wikipedia.org/wiki/L4_microkernel_family, July 2012.

[44] Wikipedia. FreeBSD Jail. http://en.wikipedia.org/wiki/FreeBSD_jail, September 2013.

[45] Wikipedia. Solaris Containers. http://en.wikipedia.org/wiki/Solaris_Containers, September 2013.

[46] Xen Blog. Xen Network: The Future Plan. http://blog.xen.org/index.php/2013/06/28/xen-network-the-future-plan/, September 2013.

[47] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *Proc. of USENIX Annual Technical Conference*, 2013.

SENIC: Scalable NIC for End-Host Rate Limiting

Sivasankar Radhakrishnan*, Yilong Geng+, Vimalkumar Jeyakumar+,
Abdul Kabbani†, George Porter*, Amin Vahdat†*

* University of California, San Diego
{sivasankar, gmporter, vahdat}@cs.ucsd.edu

+ Stanford University
{gengyl08, jvimal}@stanford.edu

† Google Inc.
akabbani@google.com

Abstract

Rate limiting is an important primitive for managing server network resources. Unfortunately, software-based rate limiting suffers from limited accuracy and high CPU overhead, and modern NICs only support a handful of rate limiters. We present SENIC, a NIC design that can natively support 10s of thousands of rate limiters—100x to 1000x the number available in NICs today. The key idea is that the host CPU only classifies packets, enqueues them in per-class queues in host memory, and specifies rate limits for each traffic class. On the NIC, SENIC maintains class metadata, computes the transmit schedule, and only pulls packets from host memory when they are ready to be transmitted (on a real time basis). We implemented SENIC on NetFPGA, with 1000 rate limiters requiring just 30KB SRAM, and it was able to accurately pace packets. Further, in a memcached benchmark against software rate limiters, SENIC is able to sustain up to 250% higher load, while simultaneously keeping tail latency under 4ms at 90% network utilization.

1 Introduction

Today's trend towards consolidating servers in dense data centers necessitates careful resource management. It is hence unsurprising that there have been several recent proposals to manage and allocate network bandwidth to different services, tenants and traffic flows in data centers. This can be a challenge given the bursty and unpredictable nature of data center traffic, which has necessitated new designs for congestion control [29].

Many of these recent proposals can be realized on top of a simple substrate of *programmable rate limiters*. For example, Seawall [38], Oktopus [4], EyeQ [12] and Gatekeeper [35] use rate limiters between pairs of communicating virtual machines to provide tenant rate guarantees. QCN [1] and D^3 [40] use explicit network feedback to rate limit traffic sources. Such systems need to support thousands of rate limited flows or traffic classes, especially in virtual machine deployments.

Unfortunately these new ideas have been hamstrung by the inability of current NIC hardware to support more than a handful of rate limiters (e.g., 8–128) [11, 18]. This has resulted in delegating packet scheduling functionality to software, which is unable to keep up with line rates, while diverting CPU resources away from application processing. As networks get faster, this problem will only get worse since the capabilities of individual cores will likely not increase. We are left with a compromise between precise hardware rate limiters that are few in number [14, 38] and software rate limiters that support more flows but suffer from high CPU overhead and burstiness (see Table 1). Software rate limiters also preclude VMs from bypassing the hypervisor for better performance [20, 24].

Property	Hardware	Software
Scales to many classes	×	✓
Works at high link speeds	✓	×
Low CPU overhead	✓	×
Precise rate enforcement	✓	×
Supports hypervisor bypass	✓	×

Table 1: Pros and cons of current hardware and software based approaches to rate limiting.

The NIC is an ideal place to offload common case or repetitive network functions. Features such as segmentation offload (TSO), and checksum offload are widely used to improve CPU performance as we scale communication rates. However, a key missing functionality is scalable rate limiting.

In this work, we present SENIC, a NIC architecture that combines the scalability of software rate limiters with the precision and low overhead of hardware rate limiters. Specifically, in hardware, SENIC supports 10s of thousands of rate limiters, 100–1000x the number available in today's NICs. The key insight in SENIC is to invert the current duties of the host and the NIC: the OS stores packet queues in host memory, and classifies packets into them. The NIC handles packet scheduling and proactively pulls packets via host memory DMA for transmission. This late-binding enables SENIC to maintain transmit queues for many classes in host memory, while the NIC enforces precise rate limits in real-time.

This paper's contributions are: (1) identifying the limitations of current operating system and NIC capabilities, (2) the SENIC design that provides scalable rate limiting with low CPU overhead, and supports hypervisor bypass, (3) a unified scheduling algorithm that enforces strict rate limits and gracefully falls back to weighted sharing if the link is oversubscribed, and (4) evaluating SENIC through implementation of a software prototype and a hardware 10G-NetFPGA prototype. Our evalua-

tion shows that SENIC can pull packets on-demand and achieve (nearly) perfect packet pacing. SENIC sustains 43–250% higher memcached load than current software rate limiters, and achieves low tail latency under 4ms even at high loads. SENIC isolates memcached from bandwidth intensive tenants, and sustains the configured rate limits for all tenants even at high loads (9Gb/s), unlike current approaches.

2 Motivation

We motivate SENIC by describing two capabilities which rely on scalable rate limiting, then describe the limitations of current NICs which prevent these capabilities from being realized.[1]

2.1 The Need For Scalable Rate Limiting

Scalable rate limiting is required for network virtualization as well as new approaches for data center congestion control, as we now describe.

Network Virtualization: Sharing network bandwidth often relies on hierarchical rate limiting and weighted bandwidth sharing. For example, Gatekeeper [35], and EyeQ [12] both rate limit traffic between every communicating source-destination VM pair, as well as use weighted sharing across source VMs on a single machine. With greater server consolidation and increasing number of cores per server, the number of rate limiters needed is only expected to increase.

To quantify the number of rate limiters required for network virtualization, we observe that Moshref et al. [22] cite the need for 10s of thousands of flow rules per server to support VM-to-VM rules in a cluster with 10s of thousands of servers. Extending these to support rate limits would thus necessitate an equal number of rate limiters. For example, if there are 50 VMs/server, each communicating with a modest 50 other VMs, we need 2500 rate limiters to provide bandwidth isolation. Furthermore, supporting native hardware rate limiting is necessary, since VMs with latency sensitive applications may want to bypass the hypervisor entirely [20, 24].

Data Center Congestion Control: Congestion control has typically been an end-host responsibility, as exemplified by TCP. Bursty correlated traffic at high link speeds, coupled with small buffers in commodity switches can result in poor application performance [29]. This has led to the development of QCN [1], DCTCP [2], HULL [3], and D^3 [40] to demonstrate how explicit network feedback can be used to pace or rate limit traffic sources and reduce congestion. In the limit, each flow (potentially thousands [7]) needs its own rate limiter.

[1]The motivation for this work appeared in an earlier workshop paper [30].

2.2 Limitations of Current Systems

Today, rate limiting is performed either (1) in hardware in the NIC, or (2) in software in the OS or VM hypervisor. We consider these alternatives in detail.

2.2.1 Hardware Rate Limiting

Modern NICs support a few hardware transmit queues (8-128) that can be rate limited. When the OS transmits a packet, it sends a doorbell request[2] to the NIC notifying it of the packet and the NIC Tx ring buffer to use. The NIC DMA's the packet descriptor from host RAM to its internal SRAM memory. The NIC uses an arbiter to compute the order in which to fetch packets from different Tx ring buffers. It looks up the physical address of the packet in the descriptor, and initiates a DMA transfer of the packet contents to its internal packet buffer. Eventually a scheduler decides when different packets are transmitted.

A straightforward approach of storing per-class packet queues on the NIC does not scale well. For instance, even storing 15KB packet data per queue for 10,000 queues requires around 150MB of SRAM, which is too expensive for commodity NICs. Likewise, storing large packet descriptor ring buffers for each queue is also expensive.

2.2.2 Software Rate Limiting

Operating systems and VM hypervisors support rate limiting and per-class prioritization; for example, Linux offers a configurable queueing discipline (QDisc) layer for enforcing packet transmission policies. The QDisc can be configured with traffic classes from which packets are transmitted by the operating system.

In general, handling individual packets in software imposes high CPU overhead due to lock contention and frequent interrupts for computing and enforcing the schedule. To reduce CPU load, the OS transfers packets to the NIC in batches, leveraging features like TSO. Once these batches of packets are in the NIC, the operating system loses control over packet schedules; packets may end up being transmitted at unpredictable times on the wire, frequently in large bursts (e.g., 64KB with 10Gb/s NICs) of back-to-back MTU-sized packets transmitted at the full line rate.

Quantifying Software Overheads: Accurate rate limiting is challenging at 10Gb/s and higher. For instance, at 40Gb/s, accurately pacing 1500B packets means sending a packet approximately every 300ns. Such accuracy is difficult to achieve even with Linux's high resolution timers, as servicing an interrupt can easily cost thousands of nanoseconds. To quantify the overhead of software rate limiting, we benchmarked Linux's

[2]A doorbell request is a mechanism whereby the network driver notifies the NIC that packet(s) are ready to be transmitted.

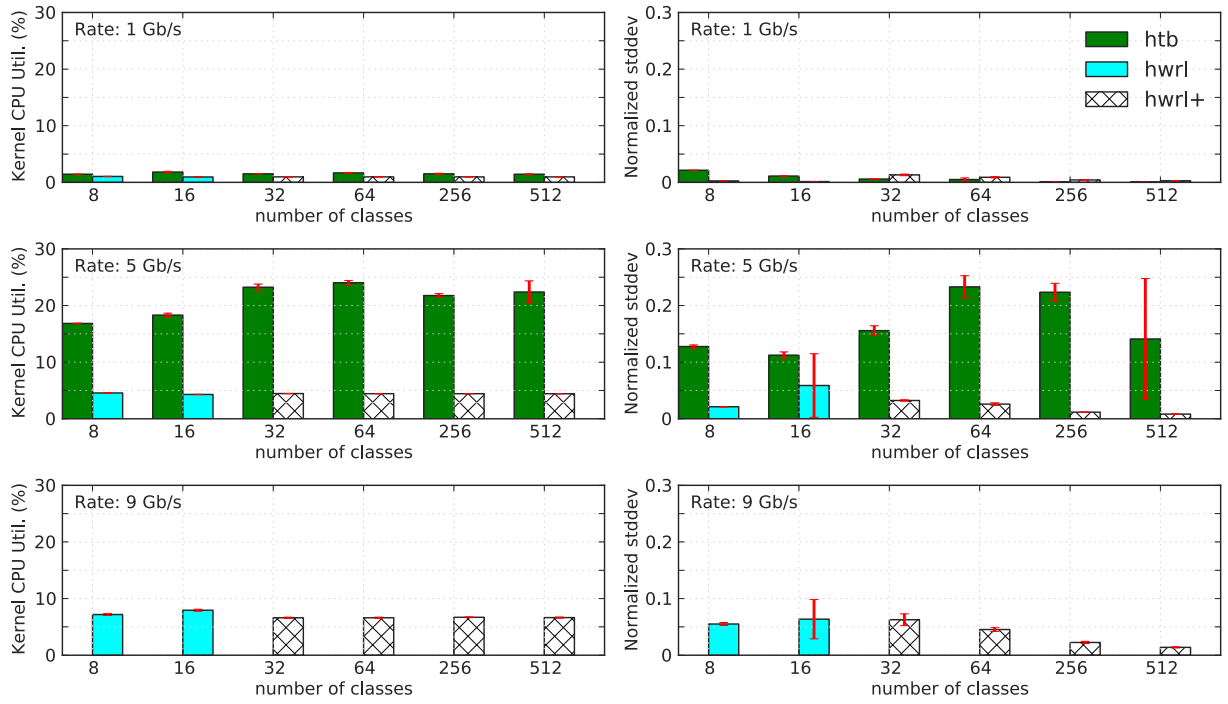


Figure 1: Comparison of CPU overhead and accuracy of software (Linux htb) and hardware (hwrl, hwrl+) rate limiting. At high rates (5Gb/s and 9Gb/s), hwrl ensures low CPU overhead and high accuracy, while htb is unable to drive more than 6.5Gb/s of aggregate throughput. Accuracy is measured as the ratio between the standard deviation of successive packet departure time differences, to the ideal. For instance, at 0.5Gb/s, 1500B packets should depart at times roughly 24us apart, but a "normalized stddev" of 0.2 means the observed deviation from 24us was as much as ∼4.8us.

Hierarchical Token Bucket (htb), and compared it to the hardware rate limiter (hwrl) on an Intel 82599 NIC. The tests were conducted on a dual 4-core, 2-way hyperthreaded Intel Xeon E5520 2.27GHz server running Linux 3.6.6.

We use userspace UDP traffic generators to send 1500B packets, and compare htb and hwrl on two metrics—OS overhead and accuracy—for varying number of classes. Each class is allocated an equal rate (total rate is 1Gb/s, 5Gb/s, or 9Gb/s). When the number of classes exceeds the available hardware rate limiters (16 in our setup), we assign classes to them in a round robin fashion (shown as hwrl+). OS overhead is the total fraction of CPU time spent in the kernel across all cores, and includes overheads in the network stack, packet scheduling, and servicing interrupts. To measure how well traffic is paced, we use a hardware packet sniffer at the receiver, which records timestamps with a 500ns precision. These metrics are plotted in Figure 1; the shaded bars indicate that many classes are mapped to one hardware rate limiter (hwrl+).

These experiments show that implementations of rate limiting in hardware are promising and deliver accurate rate limiting at low CPU overheads. However, they only offer few rate limiters, in part due to limited buffering on the NIC. Figure 1 shows that htb, while scalable in terms of the number of queues supported, is unable to pace packets at 9Gb/s, resulting in inaccurate rates.

3 Design

In the previous section, we described limitations of today's software and hardware approaches to rate limiting. The primary limitation in hardware today is scalability on the transmit path; we do not modify the receive path. In light of this, we now describe the design of the basic features in SENIC, and defer more advanced NIC features to §5. We begin with the service model abstraction.

3.1 Service Model

SENIC has a simple service model. The NIC exposes multiple transmit queues (classes), each with an associated rate limit. When the sum of rate limits of active classes does not exceed link capacity, each class is restricted to its rate limit. When it exceeds link capacity (i.e., the link is oversubscribed), SENIC gracefully shares the capacity in the ratio of class rate limits.

Entry	Bytes	Description
Queue management		
`ring_buffer`	4	Aligned address of the head of the ring buffer
`buffer_size`	2	Size of ring buffer (entries)
`head_index`	2	Index of first packet
`tail_index`	2	Index of last packet
Head packet descriptor		
`head_paddr`	8	Address of the first packet
`head_plen`	2	Length of the first packet (B)
`pkt_offset`	2	Next segment offset into the packet (for TSO)
Scheduler state (say for token bucket scheduler)		
`rate_mbps`	2	Rate limit for the queue
`tokens_bytes`	2	Number of bytes that can be sent from the queue without violating rate limit
`timestamp`	4	Last timestamp at which tokens were refreshed

Table 2: Per-class metadata in NIC SRAM. Total size=30B

3.2 CPU and NIC Responsibilities

To enforce a service model, we need a packet scheduler, and must store state for all classes. The state and functionality are spread across the CPU/host and the NIC.

State: Memory on the NIC (typically SRAM) is expensive, and we therefore use it to only store metadata about the classes. To store packet queues, SENIC leverages the large amount of host memory. Table 2 shows an example class metadata structure; the total size for storing 10,000 classes is about 300kB of SRAM. Note that the Myricom 10Gb/s NIC has 2MB SRAM [23].

Functionality: At a high level, the CPU classifies and enqueues packets in transmit queues, while the NIC computes a schedule that obeys the rate limits, pulls packets from queues in host memory using DMA, and transmits them on to the wire. The NIC handles all real time per-packet operations and transmit scheduling of packets from different classes based on their rate limits. This frees up the CPU to batch network processing, which reduces overall CPU utilization. This architecture is illustrated in Figure 2, which we now describe in detail.

3.2.1 CPU Functionality

As in current systems, the OS manages the NIC and initializes the device, creates/deletes classes, and configures rate limits. The OS is also in charge of classifying and enqueueing packets in appropriate queues. In both cases, the OS communicates with the NIC through memory-mapped IO. For instance, when the OS enqueues a packet (or a burst of packets) into a queue, it notifies the NIC through a special doorbell request that it writes to a device-specific memory address.

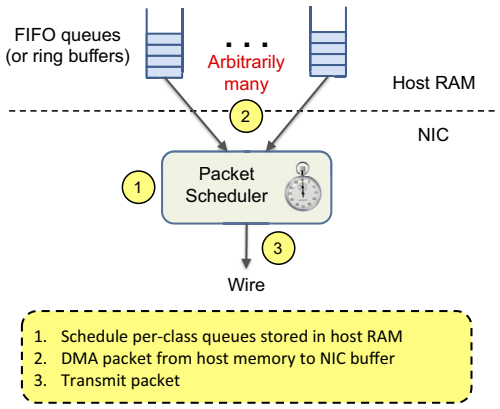


Figure 2: SENIC — "Schedule and Pull" model.

3.2.2 NIC Functionality

The NIC is responsible for all per-packet real time operations on transmit queues. Since it has limited hardware buffer resources, the NIC first computes the transmit schedule based on the rate limits. It then chooses the next packet that should be transmitted, and DMAs the packet from the per-class queue in host memory to a small internal NIC buffer for transmitting on to the wire. Figure 3 shows a schematic of the SENIC hardware and related interfaces from software.

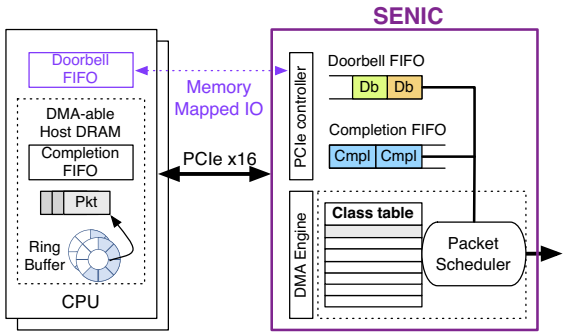


Figure 3: SENIC hardware design. Once the NIC DMAs a packet from host memory, there is further processing (e.g. checksum offloads) before the packet is transmitted on the wire. This paper focuses on the scheduler and the NIC interaction with the software stack.

Metadata: The NIC maintains state about traffic classes to enforce rate limits. In the case of a token bucket scheduler, each class maintains metadata on the number of tokens, and the global state is a list of active classes with enough tokens to transmit the next packet. The memory footprint is small, easily supporting 10,000 or more traffic classes with a few 100kB of metadata.

Scheduling: The NIC schedules and pulls packets from host memory on demand at link speed. Packets are not pulled faster, even though PCIe bandwidth between the NIC and CPU is much higher. This late bind-

ing reduces the size of NIC hardware buffers required for storing packets. It also avoids head-of-line blocking, and allows the NIC to quickly schedule newly active classes or use updated rate limits. This offloading of scheduling and real time work to the NIC is what enables SENIC to accurately enforce rate limits even at high link speeds.

Other Functionality: The NIC does more tasks than just state management and rate limiting. After the packet is DMA'd onto NIC memory, there is a standard pipeline of operations that we leave unmodified. For instance, NICs support TCP and IP checksum offloading, VLAN encapsulation, and send completions to notify the CPU when it can reclaim packet memory.

4 Packet Scheduling in SENIC

SENIC employs an internal scheduler to rate limit traffic classes. The task of packet scheduling can be realized using a number of algorithms such as Deficit Round Robin (DRR) [39], Weighted Fair Queueing (WFQ) [9], Worst-case Fair weighted Fair Queueing (WF^{2}Q) [5], or simple token buckets. The choice of algorithm impacts the sharing model, and packet delay bounds. For instance, token buckets support rate limits, but DRR is work-conserving; simply arbitrating across token buckets in a DRR-like fashion can result in bursty transmissions [6].

In this section, we start with our main requirements to pick the appropriate scheduling algorithm. We desire hierarchical rate limits, so the above work-conserving algorithms (DRR, WFQ, etc.) do not directly suit our needs. We now describe a unified scheduling algorithm that supports hierarchies and rate limits.

4.1 SENIC Packet Scheduling Algorithm

Recall that the service model exposed by SENIC is rate limits on classes, with fallback to weighted sharing proportional to the class rates. We begin by describing a scheduling algorithm which can enforce this service model. We leverage a virtual time based weighted sharing algorithm, WF^{2}Q+ [6], and modify its *system virtual time (V)* computation to support strict rate limiting with a fallback to weighted sharing. The algorithm computes a *start (S)* and *finish (F)* time for every packet based on the class rate w_i. Packets with $S \leq V$ are considered *eligible*, and the algorithm transmits eligible packets in increasing order of their finish times.

Computing Start and Finish Time: Since each class is a FIFO, the start and finish times are maintained only for the packets at the head of each transmit queue. The start time S_i of a class C_i is only updated when a packet is dequeued from that class or a packet is enqueued into a previously empty class. The finish time F_i is updated whenever S_i is updated. S_i and F_i for each flow C_i are computed in the same way as in WF^{2}Q+, as follows:

$$S_i = \begin{cases} \max(F_i, V_{enq}) & \text{on enqueue into empty queue} \\ F_i & \text{on dequeue} \end{cases}$$

$$F_i = S_i + \frac{L}{w_i}$$

where V_{enq} is the *system time V* (described below) when the packet is enqueued, and L is the head packet's length.

System Time Computation: WF^{2}Q+ computes a work-conserving schedule where at least one class is always eligible to transmit data. To enforce strict rate limits, SENIC incorporates the notions of real time and the link drain rate (R) to compute the transmit schedule. The system time is increased by 1 unit (*bytetime*), in the time it takes to transmit 1B of data at link speed, and thus incorporates the link's known drain rate R (e.g. 10Gb/s).

SENIC supports graceful fallback to weighted sharing when the link is oversubscribed. When the link is oversubscribed, we slow the system time V down to reflect the marginal rate at which the active flows are serviced. Without loss of generality, let the rate limits of flows C_i be represented as fractions w_i of the link speed R. We define the *rate oversubscription factor* ϕ to be the sum of rate limits (weights) of currently backlogged classes or flows in the system; $\phi = 0$ when no flows are active. The scheduler modifies the system time V to slow down by the rate oversubscription factor and proceed at most as fast as the link speed. V is computed as:

$$V(0) = 0$$
$$V(t+\tau) = V(t) + R\tau \times \max(1, \phi)$$

where τ is a single packet transmission period, or contiguous link idle period, or the period between successive updates to ϕ. Given the system time, the start and finish times of all classes, we schedule packets in the same order as WF^{2}Q+, i.e. in order of increasing finish times among all eligible classes at the time of dequeueing.

Example: We now look at an example transmit schedule computed using these time functions. Assume a 10Gb/s link with two continuously backlogged classes C_1 and C_2 (with rate limits 4Gb/s and 2Gb/s respectively). The transmit schedule is shown in Figure 4. The values of S_i and F_i are computed using rate limits as a fraction of link speed (so w_1 = 0.4 and w_2 = 0.2). All packets are 1500B in length.

If we consider a single iteration (7500 bytetimes), C_1 transmits 3000B, C_2 transmits 1500B, and the link is idle for (750 + 2250 = 3000 bytetimes). Thus C_1 achieves 3000 / 7500 = 0.4 of link capacity and C_2 achieves 1500 / 7500 = 0.2 of link capacity. The link remains idle for 40% of the time in each iteration, thereby enforcing strict rate limits. Notice also that the packets are appropriately interleaved and accurately paced.

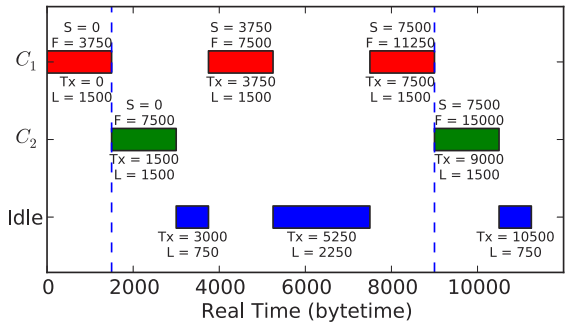


Figure 4: Transmit schedule example. Link is not oversubscribed. The interval between the vertical dashed lines indicates repeating sequence in the transmit schedule (only 1 repetition shown for clarity). S, F, L as defined in the text. T_x is the time when a particular packet transmission or idle period starts.

Delay Guarantees: The advantage of using virtual time based scheduling algorithms is that they offer strong per-packet delay guarantees. Specifically, WF^{2}Q+ guarantees that the finish time of a packet in the discretized system is no more than a bounded delay from an ideal fluid model system. SENIC's unified scheduling algorithm offers similar strong guarantees. Algorithms such as DRR do not have such strong guarantees [6].

4.2 Hierarchical bandwidth sharing

So far we discussed a flat rate limiting scheme. In practice, it may be desirable to group classes and enforce another rate limit on the group. For example, an approach useful in multi-tenant environments is a two level hierarchy where the first level implements strict rate limits for each VM on the server, and the second level provides weighted sharing between the flows originating from each VM.

It is possible to enforce any hierarchical allocation by modulating the rate limits of hardware traffic classes. Control logic in the hypervisor can measure demands and hardware counters, and adjust the rates based on preconfigured limits. We instead now describe an extension to the virtual time based scheduler described above to support a simple two level hierarchy.

Sharing Model: We define an L_1 (level 1) class as one which is directly attached to the root of the hierarchy. An L_2 (level 2) class is attached to an L_1 class. Each class is configured with a rate limit. The L_1 classes only support strict rate limits, i.e. sum of rate limits of active L_1 classes should not exceed link capacity. L_2 classes support strict rate limiting, but fallback to weighted sharing in the ratio of their rate limits when the active L_2 classes within an L_1 class oversubscribe the rate limit of that L_1 class. An L_1 class might be a leaf or an internal class while L_2 classes can only be leaves.

Start and Finish Time Computation: SENIC only computes time variables for leaf classes as packets are "enqueued" and "dequeued" only at the leaves. For L_1 leaf classes, the scheduler computes start and finish times as usual, using the rate limits of the respective classes. For each L_1 class, it maintains a rate oversubscription factor ϕ_{L_1}, of active L_2 classes within the L_1 class. For L_2 classes, to compute finish time, the scheduler scales the rate limits and uses the minimum of (1) the configured rate limit w_i of the L_2 class, and (2) the scaled rate limit of the parent L_1 class based on L_2's share, given as:

$$w_{i_{scaled}} = \min\left(w_i, w_{L_1} \times \frac{w_i}{\phi_{L_1}}\right)$$

System Time Computation: System time is purely based on real time and link drain rate R, as the L_1 classes are configured such that they never oversubscribe the link. This condition can be easily met even if weighted sharing is required at level 1 of the hierarchy, by simply having the host driver periodically measure demand and adjust the rate limits of the L_1 classes.

Summary: Driven by requirements to support rate limits, we described a scheduling algorithm incorporating both weighted sharing and rate limiting into one coherent algorithm. We also extended the algorithm to support two-level rate limits across classes and groups of classes. We realized the unified scheduling algorithm on top of QFQ [8], which in turn implements WF^{2}Q+ efficiently. The metadata structure for this QFQ based scheduler is around 40B per class, and it needs only 10kB of global state, thereby scaling easily to 10,000 classes.

5 Advanced NIC features

This section touches upon advanced features in today's NICs that are impacted by SENIC's design, and how we achieve similar functionality with SENIC.

5.1 OS and Hypervisor Bypass

Many applications benefit from bypassing the OS network stack to meet their stringent latency and performance requirements [13, 25]. Further, high-performance virtualized workloads benefit from bypassing the hypervisor entirely, and directly access the NIC [20, 24]. To support such requirements, modern NICs expose queues directly to user-space, and include features that virtualize the device state (ring buffers, etc.) through technologies like Single-Root IO Virtualization (SR-IOV [28]). We now describe how SENIC provides these features.

Configurable SR-IOV Slices or VNICs: SENIC leverages SR-IOV to expose multiple VNICs. Each

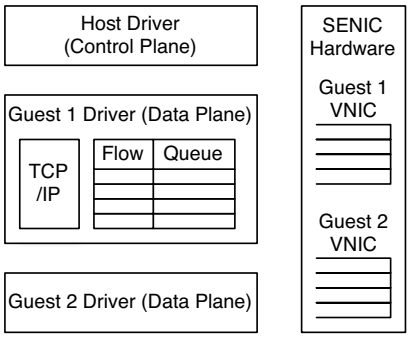


Figure 5: The SENIC architecture, with each guest given a virtualized slice of the NIC (VNIC) using SR-IOV.

VNIC is allocated a configurable number of queues, and guest VMs directly transmit and receive packets through the VNICs, as shown in Figure 5. Guest VMs are only aware of queues for their respective VNICs (which is standard SR-IOV functionality), thereby ensuring isolation between transmit queues of different guest VMs. A simple lookup table on the NIC translates VNIC queue IDs to actual queue IDs. A host SENIC driver provides the interface for the hypervisor to configure VNICs, allocate queues, and configure rate limits. A guest driver running in the VM provides a standard interface to enqueue packets into different queues on the VNIC.

Classifying Packets: SENIC relies on the operating system to classify and enqueue packets in the right traffic classes or queues. The host driver residing in the hypervisor maintains the packet classification table. It exports an OpenFlow [26] like API to configure traffic classes and rate limits. When SR-IOV is enabled, the hypervisor is bypassed in the datapath. SENIC therefore relies on the guest VM to perform packet classification.

The guest driver maintains a cached copy of the packet classification table. When the guest driver receives a packet from the network stack for transmission, it looks up its guest packet classification table for a match. If no match is found, it makes a hypercall to the hypervisor for a lookup and caches the matching rule. The actual mapping to the appropriate queue is also cached in the socket data structure to avoid repeated lookups for each packet of a flow. The hypervisor can also proactively setup rules in guest classification tables. Once the rules are cached in the guest, the hypervisor is completely bypassed during packet transmission.

Untrusted Guests: It may be unwise to trust guests to classify packets correctly. However, we argue this is not an issue. Even though SR-IOV ensures that a VM can only place packets in queues for its own VNIC, the guest may ignore the hypervisor-specified classification among its queues. We adopt a *trust-but-verify* approach to ensure that guest VMs do not cheat by directing packets to queues with higher rate limits. The key idea is that the hypervisor need not look at every packet to ensure rate limits are not violated, but instead only look at a sampled subset of packets. Since classification is used to provide QoS, sampling packet headers and verifying their classification is sufficient to identify violations. The administrator can be alerted to misbehaving guests, or they can be halted, or forced to give up SR-IOV, and rely on the hypervisor for future packet transmissions.

5.2 Other features

Below we describe few other features that are affected by SENIC's design.

Segmentation Offload: TCP Segmentation Offload (TSO) is a widely available NIC feature to reduce CPU load by transferring large (upto 64KB) TCP segments to the NIC, which are then divided into MTU sized segments and transmitted with appropriately updated checksums and sequence numbers. SENIC only pulls MTU sized portions of the packet on demand from host memory queues before transmission. This avoids long bursts from a single class, and enables better interleaving and pacing. SENIC augments per-queue metadata with a *TSO-offset* field that indicates which portion of the packet at the head of the queue remains to be transmitted. When interleaving packets, SENIC does not cache packet headers for each class on the NIC, thereby keeping NIC SRAM requirements low. When transmitting TSO packets, SENIC issues two DMA requests: one for the packet header, and another for the MTU sized payload based on TSO-offset.

Scatter-Gather: A related optimization is scatter-gather, where the NIC can fetch packet data spread across multiple memory regions, e.g., the header separately from the payload. In such cases, SENIC stores the location of the next segment to be transmitted for each queue and fetches descriptors and data on demand.

Handling Concurrency: The design assumed each transmit queue corresponds to one traffic class. To allow multiple CPU cores to concurrently enqueue packets to a class, the SENIC design is extended to support some number of queues (say 8) for each class. Round robin ordering is used among queues within a class, whenever the class gets its turn to transmit. This is easily accomplished by separately storing head and tail indices for each queue in the class metadata table, an active queue bitmap and round robin counter for each class.

Priority Scheduling: SENIC can easily also support strict priority scheduling between transmit queues of a class instead of round-robin scheduling. In this case, a priority encoder picks the highest priority active class. One use case is for applications to prioritize their traffic within a given rate limit.

6 Implementation

We have implemented two SENIC prototypes:

1. A software prototype using a dedicated CPU core to perform custom NIC processing. This implements the unified QFQ-based rate limiting and weighted sharing scheduler described in §4.1.
2. A NetFPGA-based hardware prototype designed to run microbenchmarks and evaluate the feasibility of pulling packets on demand from host memory for transmission. For engineering expediency, this prototype relies on a simpler, token bucket scheduler (without hierarchies).

We now describe both prototypes in detail. Both prototypes are available for download at `http://sivasankar.me/senic/`.

6.1 Software Prototype

The software prototype is implemented as a Linux kernel module with modest changes to the kernel. The scheduler is implemented in a new Linux queueing discipline (QDisc) kernel module. We also modified the Linux `tc` utility to enable us to configure the new QDisc module. As described in §4, SENIC's packet scheduling algorithm is implemented on top of the Quick Fair Queueing (QFQ) scheduler available in Linux.

Transmit Queues and Rate Limits: The SENIC QDisc maintains per-class FIFO transmit queues in host memory as linked lists. We configure classification rules via `tc`, and also set a rate limit for each class.

Enqueueing Packets: In Linux, when the transport layer wants to transmit a packet, it hands it down to the IP layer, which in turn hands it to the QDisc layer. When the QDisc receives a packet from IP, it first classifies the packet, then enqueues it in the corresponding queue, marking the class as active.

Dedicated CPU Core for Packet Scheduling: In today's kernel, the dequeue operation starts right after enqueue. However, to mimic NIC functionality, we modified the kernel so the enqueue call immediately returns to the caller, and dedicate a CPU core to perform all NIC scheduling (i.e. dequeueing). The dedicated CPU core runs a kernel thread that computes the schedule based on configured rate limits, and pulls packets from the active transmit queues when they should be transmitted. Packets are transferred to the physical NIC using the standard NIC driver. We disabled TSO to control the transmit schedule at a fine granularity and avoid traffic bursts.

6.2 NetFPGA Prototype

We now describe our SENIC hardware implementation on a NetFPGA [16]. The primary hardware components of SENIC are (a) the packet scheduler with the class table, (b) doorbell FIFOs to process notifications from the host, and (c) completion FIFOs to send notifications to the host. Each component maintains its own independent state machine and executes in parallel. Figure 6 below zooms into the operation of the packet scheduler. We now describe each component in detail.

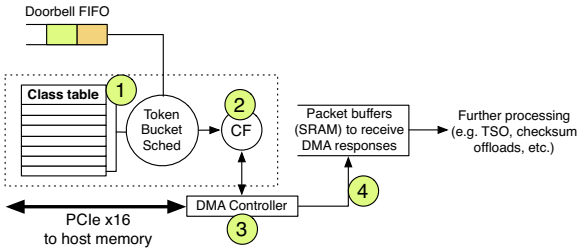


Figure 6: The 4 stages of scheduling a packet: (1) pick a class for dequeueing, (2) submit work-request to the class-fetch (CF) module, (3) DMA descriptors and packet payload from the class, (4) handoff packet payload for further processing.

6.2.1 Packet Scheduler

The scheduler operates on the class metadata table (SRAM block), and performs the following operations:

- It cycles through all active classes (i.e., classes with at least one enqueued packet), and determines if a class has enough tokens to transmit a packet (i.e., whether it is *eligible*). If not, the scheduler refreshes the class's tokens and continues with other classes.
- If the class is eligible, the scheduler submits a work-request to a 'class-fetch' (CF) module and disables the class. Each CF module has a small FIFO to accept requests from the scheduler.
- If the CF module's FIFO is full, the scheduler stalls and waits for feedback from the CF module.
- In parallel, the scheduler processes any pending doorbell requests that modify the class metadata table. For instance, if the doorbell request is an enqueue operation, the scheduler parses the class ID in the request and updates the class table.

6.2.2 Class-Fetch Module

The class-fetch (CF) module is given a class entry, and its task is to dequeue as many packets as possible until limited by (a) the tokens available for the class, or (b) the burst size of the class. The class entry only stores the descriptor for the first packet. Therefore the CF dispatches DMA requests to (a) fetch the descriptor of the next packet in the ring buffer, and (b) fetch the packet payload of the first descriptor stored in the class entry. The module then synchronously waits for the first DMA

to complete, and repeats the process until it exhausts the class tokens, or burst size. Finally, it issues (a) feedback to the scheduler with the new class entry state (updated tokens, tail pointer, and the first packet descriptor), and (b) a completion notification for the class.

The latency to make a scheduling decision, and the DMA fetch latency determine the maximum achievable throughput. We evaluate this in detail in §7.1.3.

6.2.3 Host Notifications

SENIC uses standard notification mechanisms to synchronize state between the NIC and the host: doorbell requests and completions. Doorbells update class state on the NIC (e.g., new packets and new rates), and completions notify the host about transmitted packets and processed doorbells. Doorbells and completions are stored in FIFO ring buffers, on the NIC and host respectively.

Doorbells: The doorbell is a 16B message written by the host to the memory mapped doorbell FIFO on the NIC. The FIFO is a circular buffer—the host enqueues at the tail while the NIC dequeues at the head. The host synchronizes the head index when it receives completions from the NIC, thereby freeing FIFO entries.

Completions: The NIC issues completions by DMA'ing an entry into the completion FIFO in host memory and interrupting the CPU. Each entry indicates (1) the class and number of packets transmitted from the class, or (2) the number of doorbell requests processed. This information is used by the host to reclaim packet memory, and doorbell FIFO entries. These event notifications are similar to BSD's kqueue mechanism [15].

Avoiding Write Conflicts: Note that the CF module's feedback, and host notifications both modify the class entry state. However, the feedback only modifies tokens, the first packet's length and address; the host notification only modifies the tail index. If the class's rate changes while it is being serviced, the new rate takes effect only the next iteration when the scheduler refreshes tokens.

7 Evaluation

This section dissects SENIC to answer the following aspects of the system:

- How scalable and accurate are the hardware rate limiters? We synthesized our hardware prototype with 1000 rate limiters. At 1Gb/s, we found the mean inter-packet timing was within 10ns of ideal, and the standard deviation was 191ns (less than 1.6% of the mean).
- How many packets should be pipelined for achieving line rate at various link speeds? This value depends on the scheduling and DMA latency, and the dominant factor is the DMA latency across the PCIe bus.
- How effective is SENIC at supporting high loads and delivering low latency compared to state of the art software rate limiters? We compare SENIC against Linux HTB and a Parallel Token Bucket (PTB) implementation in software (used in EyeQ [12]). We found that at very low load, all approaches have comparable latencies. But SENIC sustains 55% higher load compared to PTB, and 250% higher than HTB while keeping memcached 99.9th percentile latency under 3ms.
- How effectively can SENIC isolate different tenants—memcached latency sensitive tenants and a background bandwidth intensive UDP tenant? We found that SENIC could comfortably sustain the configured 3Gb/s of UDP traffic and nearly 6Gb/s of memcached traffic with tail latency under 4ms. However, HTB and PTB had trouble sustaining more than 1.4Gb/s of UDP traffic. SENIC sustains 233% higher memcached load compared to HTB and 43% higher than PTB.

7.1 Hardware Microbenchmarks

7.1.1 Scalability and Accuracy

Due to limitations on the number of outstanding DMA requests[3], and pipeline datawidth, we were unable to sustain more than 3Gb/s packet transmission rate, and we restrict our tests to rates less than 3Gb/s.

N	Rate	$\mu \pm \sigma$	Rel. error in μ
500	1Mb/s	12ms $\pm$ 7.1us	3.1×10^{-6}
1	10Mb/s	1.2ms $\pm$ 233ns	1.5×10^{-6}
10		1.2ms $\pm$ 240ns	1.5×10^{-6}
100		1.2ms $\pm$ 1.3μs	2.3×10^{-5}
1	100Mb/s	120μs $\pm$ 87ns	1.7×10^{-7}
10		120μs $\pm$ 173ns	1.6×10^{-6}
1	1Gb/s	11.25μs† $\pm$ 161ns	3.5×10^{-4}
3		11.25μs† $\pm$ 191ns	3.8×10^{-4}

Table 3: Rate limit accuracy as we vary the number of rate limiters N, and the rate per class. We see that SENIC is within 10^{-2}% of ideal even as we approach the maximum throughput we could push through the NetFPGA (3Gb/s).

Table 3 shows the rate limiting accuracy of one of the classes, as we vary the number of eligible classes on the NIC. We measure accuracy by timestamping every packet with a clock resolution of 10ns, and retrieving the inter-packet timestamp difference for packets of that one class. We compute the mean (μ) and standard deviation (σ), and also the relative error in μ as $|\mu_{\text{empirical}} - \mu_{\text{ideal}}|/\mu_{\text{ideal}}$. We see that SENIC very accurately enforces the configured rate even with 500 classes each operating at 1Mb/s.

[3] Our NetFPGA stalls the processor if it has more than 2 outstanding DMA requests. Others have reported a similar issue with the Virtex5 FPGA [41].

†**Note**: NetFPGA supports rates that are of form 12.8Gb/s/K, where K is an integer. Therefore, though we set the rate limit to 1Gb/s, the output will 12.8/12 Gb/s (1.067Gb/s), for which the inter-packet time is 11.25μs.

7.1.2 Scheduler Latency

We dig deeper into how long it takes for a scheduling operation in hardware. On the NetFPGA, the SRAM has a datawidth of 512 bits (64B), an access latency of 1 cycle, and enough bandwidth to support one operation (either a read or a write) every cycle. In the *worst case*, each scheduler iteration takes at most 5 cycles:

- 1 for reading the class metadata from SRAM.
- 1 for refreshing the tokens and CF-enqueue.
- 1 SRAM write for processing CF-feedback.
- 2 for processing a doorbell: 1 for reading the class metadata from SRAM, and 1 for updating class metadata and writing it back.

We synthesized our NetFPGA prototype at 100MHz (10ns per clock cycle), and therefore, it takes no more than 50ns to make a scheduling decision. We expect a production-quality NIC to have a higher clock rate, and thus a faster scheduler. For instance, the ASIC in Myricom 10Gb/s Ethernet NIC runs at a clock rate of 364.6MHz [23]. The QFQ based scheduler takes about twice as many cycles as simple token buckets [8], so with a higher clock rate, it can still complete in 50ns.

7.1.3 Maximum Per-Class Throughput

In this experiment, we first analyze the DMA latency which affects the achievable throughput per-class. We measure the time interval between sending a DMA request from the CF-module to fetch 16B from host memory, and receiving the response. We find that the average latency is $L = 1.25\mu$s ($\sigma = 40$ns) with the NetFPGA platform (using a second generation PCIe x8 bus). However, the number is often better with a production-quality NIC. For instance, the DMA latency on an Intel NIC was found to be close to 200ns [31].

Recall that the CF-module processes each class by issuing a DMA request for the class's second packet descriptor, followed by the request for the class's first packet payload. With a burst size of 1 packet per class, the maximum achievable throughput per class depends on the sum of DMA latency and scheduler latency. For instance, if the scheduler takes 50ns to dispatch a class to the CF module, the DMA latency to fetch a packet descriptor is 1250ns, and burst size is 1 packet, the maximum achievable throughput per-class is about 1500B (MTU) every 1300ns. Therefore, to achieve line rate

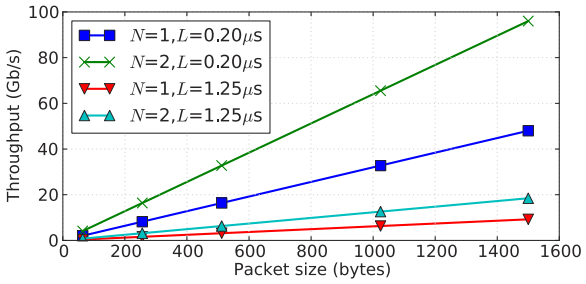


Figure 7: Maximum throughput per class as a function of the packet size, and the number N of CF modules operating in parallel, and the DMA latency L. We see that the achievable throughput on the NetFPGA ($L = 1.25\mu$s) with $N = 1$ is 9.23Gb/s with 1500B packets (if not for the DMA request constraints described earlier).

we can instantiate multiple CF modules, and the scheduler dispatches classes to them in parallel. Further, using TSO, or multiple queues per class enables higher throughput per class. Figure 7 shows the trend.

7.2 Software Macrobenchmarks

We ran experiments with our software based SENIC prototype to evaluate the application level performance when SENIC is used for rate limiting traffic.

7.2.1 Memcached

We conducted an experiment with several memcached tenants sharing a cluster—10 tenants on each machine in an 8 node cluster. Each node is a dual 4-core, 2-way hyperthreaded Intel Xeon E5520 2.27GHz server, with 24GB of RAM, a 10Gb/s NIC (Intel or Myricom), and running Linux 3.9.0. Each tenant was allocated 1 CPU hyperthread on each machine, and 2GB of RAM. One machine (M_{srv}) had 10 memcached server instances—1 for each tenant. We pre-populated them with 12B-key, 2KB-value pairs. Each of the other 7 machines (M_{cli}) ran 10 memcached client processes that sent `GET` requests to the respective tenant's memcached server instance.

Rate limits were configured for each memcached client-server pair. The total rate limit was 9.5Gb/s on M_{srv}, and 6Gb/s on M_{cli} machines. Each tenant got an equal share of the total rate, divided equally among its own destinations. These limits were chosen to be large enough that memcached would not be bandwidth limited. We ran experiments using HTB, PTB, and the SENIC software prototype.

We define the unit *rpstc*, requests per second per tenant per client, to denote the load on the system. For instance, 2,000 rpstc means each of the 7 client instances of each tenant generates a load of 2,000 req/s, resulting in a total load on M_{srv} of 140,000 req/s.

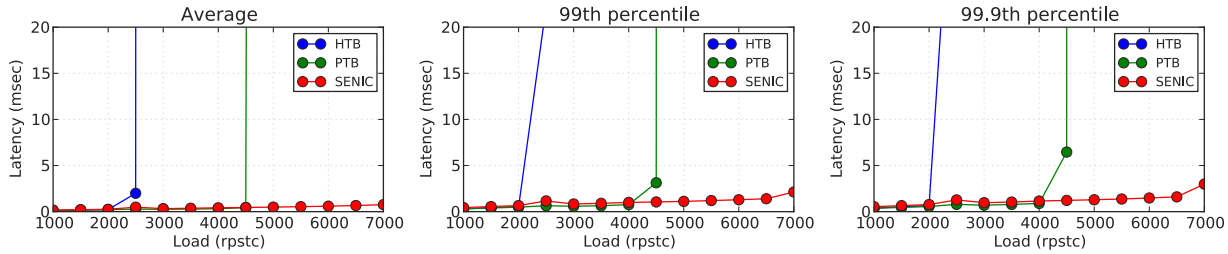


Figure 9: Memcached response latency at different loads. We see that SENIC easily sustains 7,000 rpstc (which was also the maximum load the cluster sustained without any rate limiting). However HTB and PTB latencies spike up at much lower loads.

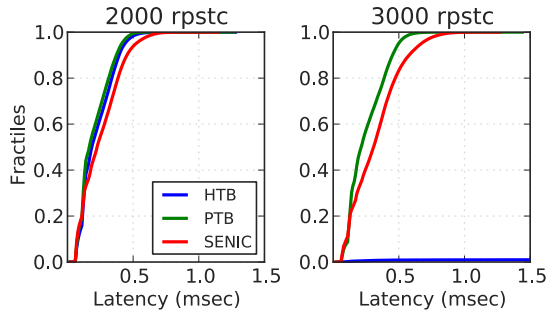


Figure 8: CDF of memcached response latency at different loads. SENIC, HTB and PTB have similar latency at 2,000 rpstc, but HTB latency shoots up at 3,000 rpstc.

Latency: We varied the client load (2000, 3000 rpstc) and observed the latency distribution of memcached responses (Figure 8). The total egress bandwidth utilization on M_{srv} is quite low at 2.3Gb/s and 2.9Gb/s respectively at the two loads. At 2,000 rpstc, we observed that HTB, PTB and SENIC perform similarly. But at 3,000 rpstc, HTB's latency suffers a drastic hit, whereas PTB and SENIC are able to keep up. With HTB, requests keep getting backlogged as the scheduler is the bottleneck and is unable to push packets out of the server fast enough. At the fairly low load of 3,000 rpstc, PTB has marginally lower latency than the SENIC software prototype due to the cache misses incurred for pulling and transmitting all packets from a single CPU core. A hardware SENIC implementation would not have this penalty.

Throughput: We varied the memcached load and measured the average, 99th, and 99.9th percentile latency in each case. Figure 9 shows that SENIC could comfortably handle 7,000 rpstc, sustaining 55% higher load compared to PTB, and 250% higher than HTB. We stopped at 7,000 rpstc as that was the maximum load the cluster could sustain even without any rate limiters (with the default Linux multi queue QDisc).

While the SENIC software prototype is much better than HTB and PTB, a hardware SENIC implementation would perform even better as there would not be cache misses for each transmit operation. Further, if hypervisor bypass is used by VMs to communicate directly with SENIC hardware, the relative latency and throughput benefits of the hardware solution would be even more.

7.2.2 Memcached and UDP Tenant Isolation

To evaluate how effectively SENIC can isolate different tenants, we repeated the above experiments with 1 co-located UDP tenant on each machine, that generates all-to-all UDP traffic as fast as it can. The total rate limit was set at 3Gb/s for UDP traffic, and 6Gb/s for memcached on each machine—divided equally among respective tenants and destinations. The maximum memcached bandwidth utilization we tested was around 5.75Gb/s on M_{srv}, so memcached was again not bandwidth limited.

Memcached Latency and Throughput: As shown in Figure 10, SENIC was able to sustain 5,000 rpstc memcached throughput (5.75Gb/s) with 99.9th percentile latency around 4ms while simultaneously delivering very close to the configured 3Gb/s of total UDP tenant traffic on the memcached server machine. On the other hand, HTB was only able to sustain 1,500 rpstc, while PTB sustained 3,500 rpstc.

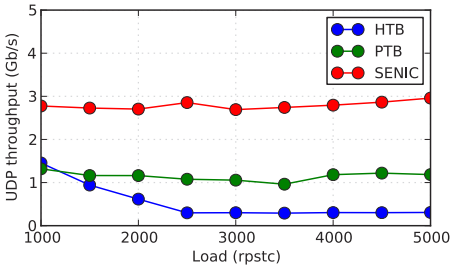


Figure 11: Throughput achieved by UDP background traffic. The configured rate limit was 3Gb/s. We found that SENIC could sustain very close to the configured 3Gb/s throughput, but HTB and PTB had trouble delivering more than 1.3Gb/s.

UDP Tenant Throughput: We measured the total throughput the UDP tenant achieved on M_{srv} as it was the primary machine under heavy overall load. Figure 11 shows that while SENIC sustained the configured 3Gb/s of throughput for the bandwidth intensive UDP tenant,

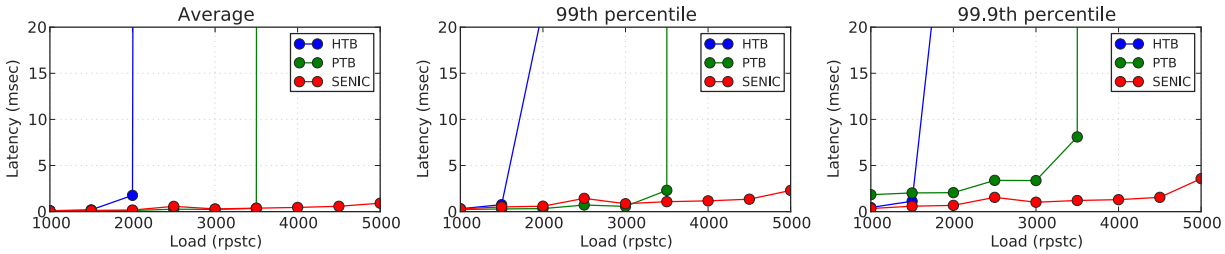


Figure 10: Memcached latency at different loads, with configured background all-to-all UDP traffic of 3Gb/s from each server. We see that SENIC could sustain 5,000 rpstc (network throughput was roughly equal to the configured limit of 6Gb/s). HTB and PTB on the other hand, fell over at lower loads.

HTB and PTB had difficulty keeping up. Even at lower memcached loads, HTB and PTB had trouble delivering more than 1.3Gb/s UDP throughput. Measurements showed that the CPU cores allocated to the UDP tenant were highly loaded, indicating that current software approaches suffer when CPU load increases and the tenants with high CPU load might notice degraded performance as the rate limiter is unable to keep up.

8 Practical Considerations

SENIC's design goals expose a tension in its implementation. Its on-board packet scheduler must be able to transfer sequences of individual packets from a potentially large number of traffic classes for fine-grained rate control. Yet, to drive high line rates, it must support a high overall DMA transfer rate to transfer packets from host memory to the wire. Thus, the performance of SENIC is upper-bounded by the performance of the host's underlying DMA subsystem.

Today's NICs rely on a number of optimizations to drive high link rates, while lowering their impact on the DMA subsystem. For example, when TSO is enabled, they can transfer the packet header just once from memory and cache it on the NIC. The NIC can then pull in the rest of the payload (issuing the appropriate DMA operations), combine it with the cached header and transmit MTU-sized segments. SENIC's design supports interleaving MTU-sized segments from different traffic classes, depending on their configured rates and burst sizes. Because the number of such classes can be quite large, SENIC does not cache packet headers on the NIC for each class. Thus, SENIC's impact on the underlying DMA subsystem is going to be greater than a traditional NIC with TSO. We now briefly examine this impact.

In the absence of TSO, SENIC requires the same number of DMA transfers from host memory as current NICs—one for each packet, in addition to the packet descriptors. However when TSO is active, SENIC issues a DMA operation for the header in addition to one for the payload, for each MTU-sized segment. Note that NICs today are capable of processing many more DMA transfers per second than required for handling MTU-sized frames at line rate. This headroom allows SENIC to drive high line rates even when TSO is enabled, despite the larger number of DMA transfers it requires.

To ground this claim experimentally, we examined the DMA subsystem performance of both 10Gb/s and 40Gb/s commercial NICs. Using a Myricom 10Gb/s NIC, we were able to sustain 13–14 million 64 byte packets per second (pps). Since packets were randomly spread across host memory, each packet required at least one DMA transfer, and thus the NIC can sustain roughly the same number of DMA transfers per second.

For 40Gb/s, we used a Mellanox Connect-X3 NIC [18] to transmit 64 byte packets. We observed that it could only support about 13.1 Mpps, which is less than the rate required to sustain 40Gb/s with 64 byte packets. However, using MTU-sized frames, and TSO disabled, it was able to drive 3.25 Mpps, which was sufficient to sustain 40Gb/s.

The above reference points allow us to gauge the performance of SENIC at both 10Gb/s and 40Gb/s. For instance, at 40Gb/s, SENIC would require $3.25 \times 2 = 6.5$ million DMA transfers per second (to DMA both payloads and headers) to achieve line rate. This is well under the 13.1 million transfers per second we were able to sustain on the same NIC. Hence, we believe that SENIC should be able to support line rate performance with TSO enabled for MTU-sized segments. Since SENIC does not introduce additional DMA requests for non-TSO packets, it should perform comparably to today's commercial NICs.

9 Related Work

We classify related work into two parts: (1) hardware improvements, and (2) software improvements, some of which try to work around limited hardware capabilities. The NIC hardware datapath has only recently received attention from the research community in light of the requirements listed in §2.

Hardware Efforts: Commercial NICs support transport offloading to support millions of connection endpoints, such as 'queue pairs' in InfiniBand [32], or TCP sockets in case of TCP offload engines [19]. The SENIC design is simpler as we only offload rate limiting, and leave the task of reliable delivery to software.

Recent work [20, 36] calls for changes in the NIC architecture in light of low-latency applications (e.g. RAMCloud [10]), and virtualized environments (e.g. public clouds). Such efforts are complementary to SENIC, which focuses only on scaling transmit scheduling. ServerSwitch [17] presented a programmable NIC to support packet classification and configurable congestion management. ServerSwitch can directly benefit from the large number of rate limiters in SENIC.

A number of efforts have focused on scalable packet schedulers in switches [21, 33]. A NIC is conceptually no different from a switch; however, switch schedulers have to deal with additional complexity due to limited on-chip SRAM, and the fact that they cannot control the exogenous traffic arrival rate. Thus, commercial switches often resort to simpler approaches like AFD [27] which can scale to 1000s of policers, but can only *drop* packets (instead of accurate pacing). On the other hand, the NIC being the first hop is in a unique position—its design can be made considerably simpler by leveraging host DRAM to store all packets. This approach enables SENIC to simultaneously scale to, and accurately pace, a large number of traffic classes.

Software Efforts: An alternate approach to deal with limited NIC rate limiters is to share them in some fashion, which has been explored by approaches like vShaper [14] and FasTrak [24]. SENIC eases the burden on such approaches, as we believe the NIC is particularly amenable to large-scale rate limiting by taking advantage of host DRAM. However, if unforeseen applications require more rate limiters than SENIC can offer, such techniques come in handy.

IsoStack [37] proposed offloading the entire TCP/IP network stack to dedicated cores. Our SENIC software prototype mimics this approach (offloading only the scheduler to a dedicated core), which explains the performance benefits in our evaluation. Architectures for fast packet IO such as Netmap [34] are orthogonal to SENIC, and they only stand to benefit from scalable rate limiting in the NIC.

10 Conclusion

Historically, the NIC has been an ideal place to offload common network tasks such as packet segmentation, VLAN encapsulation, checksumming, and rate limiting is no exception. Today's NICs offer only a handful of rate limiters, however new requirements such as performance isolation and OS-bypass for low-latency transport demand more rate limiters. We argued why it makes sense to pursue a hardware offload approach to rate limiting: at data center scale, a custom ASIC is cheaper than dedicating CPU resources for a task that requires real time packet processing. We implemented a proof-of-concept NIC on the NetFPGA to demonstrate the feasibility of scaling hardware rate limiters to thousands of queues. We believe the NIC hardware is *the* cost-effective place to implement rate limiting, especially as we scale the bandwidth per-server to 40Gb/s and beyond.

Acknowledgments

This research was supported in part by the NSF through grants CNS-1314921 and CNS-1040190. Additional funding was provided by a Google Focused Research Award. We would like to thank our shepherd Saikat Guha and the anonymous NSDI reviewers.

References

[1] ALIZADEH, M., ATIKOGLU, B., KABBANI, A., LAKSHMIKANTHA, A., PAN, R., PRABHAKAR, B., AND SEAMAN, M. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *46th Annual Allerton Conference on Communication, Control, and Computing* (2008).

[2] 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).

[3] ALIZADEH, M., KABBANI, A., EDSALL, T., PRABHAKAR, B., VAHDAT, A., AND YASUDA, M. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI* (2012).

[4] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards Predictable Datacenter Networks. In *SIGCOMM* (2011).

[5] BENNETT, J. C., AND ZHANG, H. WF^2Q : Worst-case Fair Weighted Fair Queueing. In *INFOCOM* (1996).

[6] BENNETT, J. C. R., AND ZHANG, H. Hierarchical Packet Fair Queueing Algorithms. In *SIGCOMM* (1996).

[7] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *IMC* (2010).

[8] CHECCONI, F., RIZZO, L., AND VALENTE, P. QFQ: Efficient Packet Scheduling With Tight Guarantees. In *IEEE/ACM Transactions on Networking* (June 2013).

[9] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM* (1989).

[10] FLAJSLIK, M., AND ROSENBLUM, M. Network Interface Design for Low Latency Request-Response Protocols. In *USENIX ATC* (2013).

[11] Intel 82599 10GbE Controller. `http://www.intel.com/content/dam/doc/datasheet/82599-10-gbe-controller-datasheet.pdf`.

[12] JEYAKUMAR, V., ALIZADEH, M., MAZIÈRES, D., PRABHAKAR, B., KIM, C., AND GREENBERG, A. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI* (2013).

[13] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: Predictable Low Latency for Data Center Applications. In *SOCC* (2012).

[14] KUMAR, G., KANDULA, S., BODIK, P., AND MENACHE, I. Virtualizing Traffic Shapers for Practical Resource Allocation. In *HotCloud* (2013).

[15] LEMON, J. Kqueue - A Generic and Scalable Event Notification Facility. In *USENIX ATC* (2001).

[16] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA - An Open Platform for Gigabit-rate Network Switching and Routing. In *IEEE International Conference on Microelectronic Systems Education* (2007).

[17] LU, G., GUO, C., LI, Y., ZHOU, Z., YUAN, T., WU, H., XIONG, Y., GAO, R., AND ZHANG, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *NSDI* (2011).

[18] Mellanox Connect-X3. `http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN_Card.pdf`.

[19] MOGUL, J. C. TCP Offload Is a Dumb Idea Whose Time Has Come. In *HotOS* (2003).

[20] MOGUL, J. C., MUDIGONDA, J., SANTOS, J. R., AND TURNER, Y. The NIC Is the Hypervisor: Bare-Metal Guests in IaaS Clouds. In *HotOS* (2013).

[21] MOON, S., REXFORD, J., AND SHIN, K. G. Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches. *IEEE Transactions on Computers* (Nov. 2000).

[22] MOSHREF, M., YU, M., SHARMA, A., AND GOVINDAN, R. Scalable Rule Management for Data Centers. In *NSDI* (2013).

[23] Myri-10G PCI Express Network Adapter. `https://www.myricom.com/products/network-adapters/10g-pcie-8b-2s.html`, Retrieved 25 September 2013.

[24] MYSORE, R. N., PORTER, G., AND VAHDAT, A. FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers. In *CoNEXT* (2013).

[25] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast Crash Recovery in RAMCloud. In *SOSP* (2011).

[26] OpenFlow Consortium. `http://www.openflow.org`.

[27] PAN, R., BRESLAU, L., PRABHAKAR, B., AND SHENKER, S. Approximate Fairness Through Differential Dropping. *SIGCOMM CCR* (Apr. 2003).

[28] PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. `http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html`, Retrieved 25 September 2013.

[29] PHANISHAYEE, A., KREVAT, E., VASUDEVAN, V., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND SESHAN, S. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *USENIX FAST* (2008).

[30] RADHAKRISHNAN, S., JEYAKUMAR, V., KABBANI, A., PORTER, G., AND VAHDAT, A. NicPic: Scalable and Accurate End-Host Rate Limiting. In *HotCloud* (2013).

[31] RAMCloud RPC Performance Numbers. `https://ramcloud.stanford.edu/wiki/display/ramcloud/RPC+Performance+Numbers`, Retrieved 25 September 2013.

[32] RDMA Aware Networks Programming User Manual. `http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf`, Retrieved 25 September 2013.

[33] REXFORD, J., BONOMI, F., GREENBERG, A., AND WONG, A. A Scalable Architecture for Fair Leaky-Bucket Shaping. In *INFOCOM* (1997).

[34] RIZZO, L. netmap: a novel framework for fast packet I/O. In *USENIX ATC* (2012).

[35] RODRIGUES, H., SANTOS, J. R., TURNER, Y., SOARES, P., AND GUEDES, D. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In *WIOV* (2011).

[36] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's Time for Low Latency. In *HotOS* (2011).

[37] SHALEV, L., SATRAN, J., BOROVIK, E., AND BEN-YEHUDA, M. IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *USENIX ATC* (2010).

[38] SHIEH, A., KANDULA, S., GREENBERG, A., AND KIM, C. Seawall: Performance Isolation for Cloud Datacenter Networks. In *HotCloud* (2010).

[39] SHREEDHAR, M., AND VARGHESE, G. Efficient Fair Queueing Using Deficit Round Robin. In *SIGCOMM* (1995).

[40] WILSON, C., BALLANI, H., KARAGIANNIS, T., AND ROWTRON, A. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM* (2011).

[41] Xilinx User Community Forums: ML506 board: Why my DMA IP hangs OS? `http://forums.xilinx.com/t5/PCI-Express/ML506-board-Why-my-DMA-IP-hangs-OS/td-p/94298`, Retrieved 25 September 2013.

mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems

EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong
Sunghwan Ihm*, Dongsu Han, and KyoungSoo Park

KAIST *Princeton University

Abstract

Scaling the performance of short TCP connections on multicore systems is fundamentally challenging. Although many proposals have attempted to address various shortcomings, inefficiency of the kernel implementation still persists. For example, even state-of-the-art designs spend 70% to 80% of CPU cycles in handling TCP connections in the kernel, leaving only small room for innovation in the user-level program.

This work presents mTCP, a high-performance user-level TCP stack for multicore systems. mTCP addresses the inefficiencies from the ground up—from packet I/O and TCP connection management to the application interface. In addition to adopting well-known techniques, our design (1) translates multiple expensive system calls into a single shared memory reference, (2) allows efficient flow-level event aggregation, and (3) performs batched packet I/O for high I/O efficiency. Our evaluations on an 8-core machine showed that mTCP improves the performance of small message transactions by a factor of 25 compared to the latest Linux TCP stack and a factor of 3 compared to the best-performing research system known so far. It also improves the performance of various popular applications by 33% to 320% compared to those on the Linux stack.

1 Introduction

Short TCP connections are becoming widespread. While large content transfers (*e.g.*, high-resolution videos) consume the most bandwidth, short "transactions" [1] dominate the number of TCP flows. In a large cellular network, for example, over 90% of TCP flows are smaller than 32 KB and more than half are less than 4 KB [45].

Scaling the processing speed of these short connections is important not only for popular user-facing online services [1, 2, 18] that process small messages. It is also critical for backend systems (*e.g.*, memcached clusters [36]) and middleboxes (*e.g.*, SSL proxies [32] and redundancy elimination [31]) that must process TCP connections at high speed. Despite recent advances in software packet processing [4, 7, 21, 27, 39], supporting high TCP transaction rates remains very challenging. For example, Linux TCP transaction rates peak at about 0.3 million transactions per second (shown in Section 5), whereas packet I/O can scale up to tens of millions packets per second [4, 27, 39].

Prior studies attribute the inefficiency to either the high system call overhead of the operating system [28, 40, 43] or inefficient implementations that cause resource contention on multicore systems [37]. The former approach drastically changes the I/O abstraction (*e.g.*, socket API) to amortize the cost of system calls. The practical limitation of such an approach, however, is that it requires significant modifications within the kernel and forces existing applications to be re-written. The latter one typically makes incremental changes in existing implementations and, thus, falls short in fully addressing the inefficiencies.

In this paper, we explore an alternative approach that delivers high performance without requiring drastic changes to the existing code base. In particular, we take a clean-slate approach to assess the performance of an untethered design that divorces the limitation of the kernel implementation. To this end, we build a user-level TCP stack from the ground up by leveraging high-performance packet I/O libraries that allow applications to directly access the packets. Our user-level stack, mTCP, is designed for three explicit goals:

1. Multicore scalability of the TCP stack.
2. Ease of use (i.e., application portability to mTCP).
3. Ease of deployment (i.e., no kernel modifications).

Implementing TCP in the user level provides many opportunities. In particular, it can eliminate the expensive system call overhead by translating syscalls into inter-process communication (IPC). However, it also in-

[1] We refer to a request-response pair as a transaction. These transactions are typically small in size.

	Accept queue	Conn. Locality	Socket API	Event Handling	Packet I/O	Application Modification	Kernel Modification
PSIO [12], DPDK [4], PF_RING [7], netmap [21]	No TCP stack				**Batched**	No interface for transport layer	**No (NIC driver)**
Linux-2.6	Shared	None	BSD socket	Syscalls	Per packet	**Transparent**	**No**
Linux-3.9	**Per-core**	None	BSD socket	Syscalls	Per packet	Add option SO_REUSEPORT	**No**
Affinity-Accept [37]	**Per-core**	**Yes**	BSD socket	Syscalls	Per packet	**Transparent**	Yes
MegaPipe [28]	**Per-core**	**Yes**	lwsocket	**Batched syscalls**	Per packet	Event model to completion I/O	Yes
FlexSC [40], VOS [43]	Shared	None	BSD socket	**Batched syscalls**	Per packet	Change to use new API	Yes
mTCP	**Per-core**	**Yes**	**User-level socket**	**Batched function calls**	**Batched**	Socket API to mTCP API	**No (NIC driver)**

Table 1: Comparison of the benefits of previous work and mTCP.

troduces fundamental challenges that must be addressed—processing IPC messages, including shared memory messages, involve context-switches that are typically much more expensive than the system calls themselves [3, 29].

Our key approach is to amortize the context-switch overhead over a batch of packet-level and socket-level events. While packet-level batching [27] and system-call batching [28, 40, 43] (including socket-level events) have been explored individually, integrating the two requires a careful design of the networking stack that translates packet-level events to socket-level events and vice-versa.

This paper makes two key contributions:

First, we demonstrate that significant performance gain can be obtained by integrating packet- and socket-level batching. In addition, we incorporate all known optimizations, such as per-core listen sockets and load balancing of concurrent flows on multicore CPUs with receive-side scaling (RSS). The resulting TCP stack outperforms Linux and MegaPipe [28] by up to 25x (w/o `SO_REUSEPORT`) and 3x, respectively, in handling TCP transactions. This directly translates to application performance; mTCP increases existing applications' performance by 33% (SSLShader) to 320% (lighttpd).

Second, unlike other designs [23,30], we show that such integration can be done purely at the user level in a way that ensures ease of porting without requiring significant modifications to the kernel. mTCP provides BSD-like socket and epoll-like event-driven interfaces. Migrating existing event-driven applications is easy since one simply needs to replace the socket calls to their counterparts in mTCP (*e.g.*, `accept()` becomes `mtcp_accept()`) and use the per-core listen socket.

2 Background and Motivation

We first review the major inefficiencies in existing TCP implementations and proposed solutions. We then discuss our motivation towards a user-level TCP stack.

2.1 Limitations of the Kernel's TCP Stack

Recent studies proposed various solutions to address four major inefficiencies in the Linux TCP stack: lack of connection locality, shared file descriptor space, inefficient packet processing, and heavy system call overhead [28].

Lack of connection locality: Many applications are multi-threaded to scale their performance on multicore systems. However, they typically share a listen socket that accepts incoming connections on a well-known port. As a result, multiple threads contend for a lock to access the socket's accept queue, resulting in a significant performance degradation. Also, the core that executes the kernel code for handling a TCP connection may be different from the one that runs the application code that actually sends and receives data. Such lack of connection locality introduces additional overhead due to increased CPU cache misses and cache-line sharing [37].

Affinity-Accept [37] and MegaPipe [28] address this issue by providing a local accept queue in each CPU core and ensuring flow-level core affinity across the kernel and application thread. Recent Linux kernel (3.9.4) also partly addresses this by introducing the `SO_REUSEPORT` [14] option, which allows multiple threads/processes to bind to the same port number.

Shared file descriptor space: In POSIX-compliant operating systems, the file descriptor (fd) space is shared within a process. For example, Linux searches for the minimum available fd number when allocating a new socket. In a busy server that handles a large number of concurrent connections, this incurs significant overhead due to lock contention between multiple threads [20]. The use of file descriptors for sockets, in turn, creates extra overhead of going through the Linux Virtual File System (VFS), a pseudo-filesystem layer for supporting common file operations. MegaPipe eliminates this layer for sockets by explicitly partitioning the fd space for sockets and regular files [28].

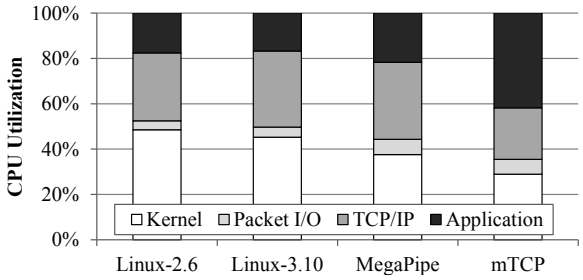


Figure 1: CPU usage breakdown when running `lighttpd` serving a 64B file per connection.

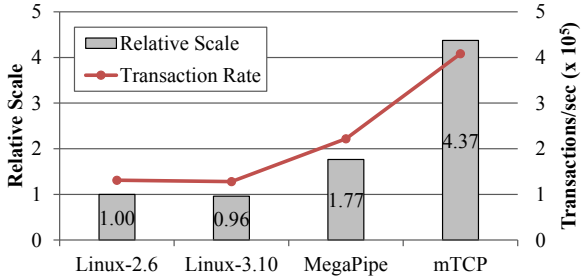


Figure 2: Relative scale of # transactions processed per CPU cycle in the kernel (including TCP/IP and I/O) across four `lighttpd` versions.

Inefficient per-packet processing: Previous studies indicate per-packet memory (de)allocation and DMA overhead, NUMA-unaware memory access, and heavy data structures (*e.g.*, `sk_buff`) as the main bottlenecks in processing small packets [27, 39]. To reduce the per-packet overhead, it is essential to batch process multiple packets. While many recent user-level packet I/O libraries [4, 7, 27, 39] address these problems, these libraries do not provide a full-fledged TCP stack, and not all optimizations are incorporated into the kernel.

System call overhead: The BSD socket API requires frequent user/kernel mode switching when there are many short-lived concurrent connections. As shown in FlexSC [40] and VOS [43], frequent system calls can result in processor state (*e.g.*, top-level caches, branch prediction table, etc.) pollution that causes performance penalties. Previous solutions propose system call batching [28, 43] or efficient system call scheduling [40] to amortize the cost. However, it is difficult to readily apply either approach to existing applications since they often require user and/or kernel code modification due to the changes to the system call interface and/or its semantics.

Table 1 summarizes the benefits provided by previous work compared to a vanilla Linux kernel. Note that there is not a single system that provides all of the benefits.

2.2 Why User-level TCP?

While many previous designs have tried to scale the performance of TCP in multicore systems, few of them truly overcame the aforementioned inefficiencies of the kernel. This is evidenced by the fact that even the best-performing system, MegaPipe, spends a dominant portion of CPU cycles (∼80%) inside the kernel. Even more alarming is the fact that these CPU cycles are not utilized efficiently; according to our own measurements, Linux spends more than 4x the cycles (in the kernel and the TCP stack combined) than mTCP does while handling the same number of TCP transactions.

To reveal the significance of this problem, we profile the server's CPU usage when it is handling a large number of concurrent TCP transactions (8K to 48K concurrent TCP connections). For this experiment, we use a simple web server (`lighttpd` v1.4.32 [8]) running on an 8-core Intel Xeon CPU (2.90 GHz, E5-2690) with 32 GB of memory and a 10 Gbps NIC (Intel 82599 chipsets). Our clients use `ab` v2.3 [15] to repeatedly download a 64B file per connection. Multiple clients are used in our experiment to saturate the CPU utilization of the server. Figure 1 shows the breakdown of CPU usage comparing four versions of the lighttpd server: a multithreaded version that harnesses all 8 CPU cores on Linux 2.6.32 and 3.10.12 [2] (Linux), a version ported to MegaPipe [3] (MegaPipe), and a version using mTCP, our user-level TCP stack, on Linux 2.6.32 (mTCP). Note that MegaPipe adopts all recent optimizations such as per-core accept queues and file descriptor space, as well as user-level system call batching, but reuses the existing kernel for packet I/O and TCP/IP processing.

Our results indicate that Linux and MegaPipe spend 80% to 83% of CPU cycles in the kernel which leaves only a small portion of the CPU to user-level applications. Upon further investigation, we find that lock contention for shared in-kernel data structures, buffer management, and frequent mode switch are the main culprits. This implies that the kernel, including its stack, is the major bottleneck. Furthermore, the results in Figure 2 show that the CPU cycles are not spent efficiently in Linux and MegaPipe. The bars indicate the relative number of transactions processed per each CPU cycle inside the kernel and the TCP stack (*e.g.*, outside the application), normalized by the performance of Linux 2.6.32. We find that mTCP uses the CPU cycles 4.3 times more effectively than Linux. As a result, mTCP achieves 3.1x and 1.8x the performance of Linux 2.6 and MegaPipe, respectively, while using fewer CPU cycles in the kernel and the TCP stack.

Now, the motivation of our work is clear. Can we design a user-level TCP stack that incorporates all existing optimizations into a single system and achieve all benefits that individual systems have provided in the past? How much of a performance improvement can we get if we build such a system? Can we bring the performance of existing packet I/O libraries to the TCP stack?

[2] This is the latest Linux kernel version as of this writing.

[3] We use Linux 3.1.3 for MegaPipe due to its patch availability.

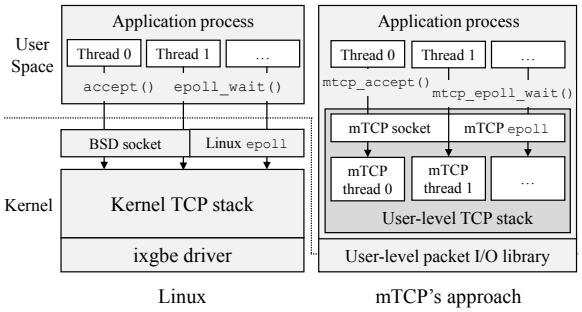


Figure 3: mTCP Design Overview.

To answer these questions, we build a TCP stack in the user level. User-level TCP is attractive for many reasons. First, it allows us to easily depart from the kernel's complexity. In particular, due to shared data structures and various semantics that the kernel has to support (*e.g.*, POSIX and VFS), it is often difficult to separate the TCP stack from the rest of the kernel. Furthermore, it allows us to directly take advantage of the existing optimizations in the high-performance packet I/O library, such as netmap [39] and Intel DPDK [4]. Second, it allows us to apply batch processing as the first principle, harnessing the ideas in FlexSC [40] and VOS [43] without extensive kernel modifications. In addition to performing batched packet I/O, the user-level TCP naturally collects multiple flow-level events to and from the user application (*e.g.*, `connect()`/`accept()` and `read()`/`write()` for different connections) without the overhead of frequent mode switching in system calls. Finally, it allows us to easily preserve the existing application programming interface. Our TCP stack is backward-compatible in that we provide a BSD-like socket interface.

3 Design

The goal of mTCP is to achieve high scalability on multicore systems while maintaining backward compatibility to existing multi-threaded, event-driven applications. Figure 3 presents an overview of our system. At the highest level, applications link to the mTCP library, which provides a socket API and an event-driven programming interface for backward compatibility. The two underlying components, user-level TCP stack and packet I/O library, are responsible for achieving high scalability. Our user-level TCP implementation runs as a thread on each CPU core within the same application process. The mTCP thread directly transmits and receives packets to and from the NIC using our custom packet I/O library. Existing user-level packet libraries only allow one application to access an NIC port. Thus, mTCP can only support one application per NIC port. However, we believe this can be addressed in the future using virtualized network interfaces (more details in Section 3.3). Applications can still choose to work with the existing TCP stack, provided that they only use NICs that are not used by mTCP.

In this section, we first present the design of mTCP's highly scalable lower-level components in Sections 3.1 and 3.2. We then discuss the API and semantics that mTCP provides to support applications in Section 3.3.

3.1 User-level Packet I/O Library

Several packet I/O systems allow high-speed packet I/O ($\sim$100M packets/sec) from a user-level application [4, 7, 12]. However, they are not suitable for implementing a transport layer since their interface is mainly based on polling. Polling can significantly waste precious CPU cycles that can potentially benefit the applications. Furthermore, our system requires efficient multiplexing between TX and RX queues from multiple NICs. For example, we do not want to block a TX queue while sending a data packet when a control packet is waiting to be received. This is because if we block the TX queue, important control packets, such as SYN or ACK, may be dropped, resulting in a significant performance degradation due to retransmissions.

To address these challenges, mTCP extends the PacketShader I/O engine (PSIO) [27] to support an efficient *event-driven* packet I/O interface. PSIO offers high-speed packet I/O by utilizing RSS that distributes incoming packets from multiple RX queues by their flows, and provides flow-level core affinity to minimize the contention among the CPU cores. On top of PSIO's high-speed packet I/O, the new event-driven interface allows an mTCP thread to efficiently wait for events from RX and TX queues from multiple NIC ports at a time.

The new event-driven interface, `ps_select()`, works similarly to `select()` except that it operates on TX/RX queues of interested NIC ports for packet I/O. For example, mTCP specifies the interested NIC interfaces for RX and/or TX events with a timeout in microseconds, and `ps_select()` returns immediately if any event of interest is available. If such an event is not detected, it enables the interrupts for the RX and/or TX queues and yields the thread context. Eventually, the interrupt handler in the driver wakes up the thread if an I/O event becomes available or the timeout expires. `ps_select()` is also similar to the `select()`/`poll()` interface supported by netmap [39]. However, unlike netmap, we do not integrate this with the general-purpose event system in Linux to avoid its overhead.

The use of PSIO brings the opportunity to amortize the overhead of system calls and context switches throughout the entire system, in addition to eliminating the per-packet memory allocation and DMA overhead. In PSIO, packets are received and transmitted in batches [27], amortizing the cost of expensive PCIe operations, such as DMA address mapping and IOMMU lookups.

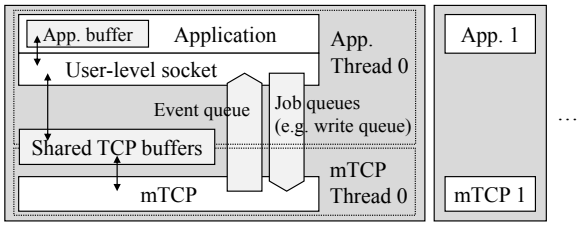


Figure 4: Thread model of mTCP.

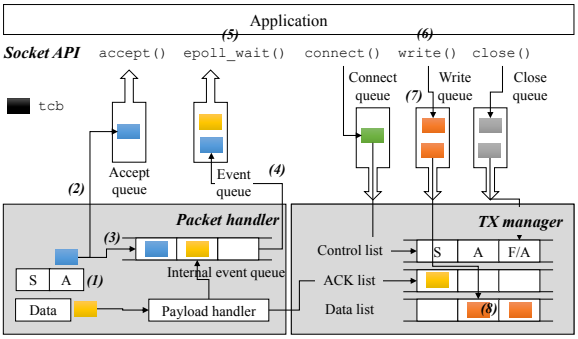


Figure 5: An example of TCP processing in mTCP.

3.2 User-level TCP Stack

A user-level TCP stack naturally eliminates many system calls (*e.g.*, socket I/O), which can potentially reduce a significant part of the Linux TCP overhead. One approach to a user-level TCP stack is to implement it completely as a library that runs as part of the application's main thread. This "zero-thread TCP" could potentially provide the best performance since this translates costly system calls into light-weight user-level function calls. However, the fundamental limitation of this approach is that the correctness of internal TCP processing depends on the timely invocation of TCP functions from the application.

In mTCP, we choose to create a separate TCP thread to avoid such an issue and to minimize the porting effort for existing applications. Figure 4 shows how mTCP interacts with the application thread. The application uses mTCP library functions that communicate with the mTCP thread via shared buffers. The access to the shared buffers is granted only through the library functions, which allows safe sharing of the internal TCP data. When a library function needs to modify the shared data, it simply places a request (*e.g.*, `write()` request) to a job queue. This way, multiple requests from different flows can be piled to the job queue at each loop, which are processed in batch when the mTCP thread regains the CPU. Flow events from the mTCP thread (*e.g.*, new the CPU core. Flow events from the mTCP thread (*e.g.*, new connections, new data arrival, etc.) are delivered in a similar way

This, however, requires additional overhead of managing concurrent data structures and context switch between the application and the mTCP thread. Such cost is unfortunately not negligible, typically much larger than the system call overhead [29]. One measurement on a recent Intel CPU shows that a thread context switch takes 19 times the duration of a null system call [3].

In this section, we describe how mTCP addresses these challenges and achieves high scalability with the user-level TCP stack. We first start from how mTCP processes TCP packets in Section 3.2.1, then present a set of key optimizations we employ to enhance its performance in Sections 3.2.2, 3.2.3, and 3.2.4.

3.2.1 Basic TCP Processing

When the mTCP thread reads a batch of packets from the NIC's RX queue, mTCP passes them to the TCP packet processing logic which follows the standard TCP specification. For each packet, mTCP first searches (or creates) a TCP control block (`tcb`) of the corresponding flow in the flow hash table. As in Figure 5, if a server side receives an ACK for its SYN/ACK packet (1), the `tcb` for the new connection will be enqueued to an accept queue (2), and a read event is generated for the listening socket (3). If a new data packet arrives, mTCP copies the payload to the socket's read buffer and enqueues a read event to an internal event queue. mTCP also generates an ACK packet and keeps it in the ACK list of a TX manager until it is written to a local TX queue.

After processing a batch of received packets, mTCP flushes the queued events to the application event queue (4) and wakes up the application by signaling it. When the application wakes up, it processes multiple events in a single event loop (5), and writes responses from multiple flows without a context switch. Each socket's `write()` call writes data to its send buffer (6), and enqueues its `tcb` to the write queue (7). Later, mTCP collects the `tcb`s that have data to send, and puts them into a send list (8). Finally, a batch of outgoing packets from the list will be sent by a packet I/O system call, transmitting them to the NIC's TX queue.

3.2.2 Lock-free, Per-core Data Structures

To minimize inter-core contention between the mTCP threads, we localize all resources (*e.g.*, flow pool, socket buffers, etc.) in each core, in addition to using RSS for flow-level core affinity. Moreover, we completely eliminate locks by using lock-free data structures between the application and mTCP. On top of that, we also devise an efficient way of managing TCP timer operations.

Thread mapping and flow-level core affinity: We preserve flow-level core affinity in two stages. First, the packet I/O layer ensures to evenly distribute TCP connection workloads across available CPU cores with RSS. This essentially reduces the TCP scalability problem to each core. Second, mTCP spawns one TCP thread for each application thread and co-locates them in the same physical CPU core. This preserves the core affinity of

packet and flow processing, while allowing them to use the same CPU cache without cache-line sharing.

Multi-core and cache-friendly data structures: We keep most data structures, such as the flow hash table, socket id manager, and the pool of `tcb` and socket buffers, local to each TCP thread. This significantly reduces any sharing across threads and CPU cores, and achieves high parallelism. When a data structure must be shared across threads (*e.g.*, between mTCP and the application thread), we keep all data structures local to each core and use lock-free data structures by using a single-producer and single-consumer queue. We maintain write, connect, and close queues, whose requests go from the application to mTCP, and an accept queue where new connections are delivered from mTCP to the application.

In addition, we keep the size of frequently accessed data structures small to maximize the benefit of the CPU cache, and make them aligned with the size of a CPU cache line to prevent any false sharing. For example, we divide `tcb` into two parts where the first-level structure holds 64 bytes of the most frequently-accessed fields and two pointers to next-level structures that have 128 and 192 bytes of receive/send-related variables, respectively.

Lastly, to minimize the overhead of frequent memory allocation/deallocation, we allocate a per-core memory pool for `tcb`s and socket buffers. We also utilize huge pages to reduce the TLB misses when accessing the `tcb`s. Because their access pattern is essentially random, it often causes a large number of TLB misses. Putting the memory pool of `tcb`s and a hash table that indexes them into huge pages reduces the number of TLB misses.

Efficient TCP timer management: TCP requires timer operations for retransmission timeouts, connections in the TIME_WAIT state, and connection keep-alive checks. mTCP provides two types of timers: one managed by a sorted list and another built with a hash table. For coarse-grained timers, such as managing connections in the TIME_WAIT state and connection keep-alive check, we keep a list of `tcb`s sorted by their timeout values. Every second, we check the list and handle any `tcb`s whose timers have expired. Note that keeping the list sorted is trivial since a newly-added entry should have a strictly larger timeout than any of those that are already in the list. For fine-grained retransmission timers, we use the remaining time (in milliseconds) as the hash table index, and process all `tcb`s in the same bucket when a timeout expires for the bucket. Since retransmission timers are used by virtually all `tcb`s whenever a data (or SYN/FIN) packet is sent, keeping a sorted list would consume a significant amount of CPU cycles. Such fine-grained event batch processing with millisecond granularity greatly reduces the overhead.

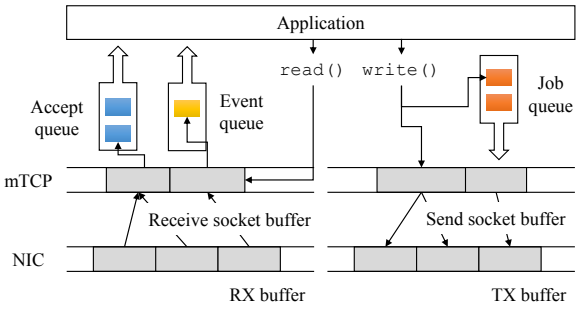


Figure 6: Batch processing of events and jobs.

3.2.3 Batched Event Handling

mTCP transparently enables batch processing of multiple flow events, which effectively amortizes the context switch cost over multiple events. After receiving packets in batch, mTCP processes them to generate a batch of flow-level events. These events are then passed up to the application, as illustrated in Figure 6. The TX direction works similarly, as the mTCP library transparently batches the write events into a write queue. While the idea of amortizing the system call overhead using batches is not new [28, 43], we demonstrate that benefits similar to that of batched syscalls can be effectively achieved in user-level TCP.

In our experiments with 8 RX/TX queues per 10 Gbps port, the average number of events that an mTCP thread generates in a single scheduling period is about 2,170 for both TX and RX directions (see Section 5.1). This ensures that the cost of a context switch is amortized over a large number of events. Note the fact that the use of multiple queues does not decrease the number of the events processed in a batch.

3.2.4 Optimizing for Short-lived Connections

We employ two optimizations for supporting many short-lived concurrent connections.

Priority-based packet queueing: For short TCP connections, the control packets (*e.g.*, SYN and FIN) have a critical impact on the performance. Since the control packets are mostly small-sized, they can often be delayed for a while when they contend for an output port with a large number of data packets. We prioritize control packets by keeping them in a separate list. We maintain three kinds of lists for TX as shown in Figure 5. First, a control list contains the packets that are directly related to the state of a connection such as SYN, SYN/ACK, and ACK, or FIN and FIN/ACK. We then manage ACKs for incoming data packets in an ACK list. Finally, we keep a data list to send data in the socket buffers of TCP flows. When we put actual packets in a TX queue, we first fill the packets from a control list and an ACK list, and later queue the data packets. By doing this, we prioritize important packets

to prevent short connections from being delayed by other long connections. [4]

Lightweight connection setup: In addition, we find that a large portion of connection setup cost is from allocating memory space for TCP control blocks and socket buffers. When many threads concurrently call `malloc()` or `free()`, the memory manager in the kernel can be easily contended. To avoid this problem, we pre-allocate large memory pools and manage them at user level to satisfy memory (de)allocation requests locally in the same thread.

3.3 Application Programming Interface

One of our primary design goals is to minimize the porting effort of existing applications so that they can easily benefit from our user-level TCP stack. Therefore, our programming interface must preserve the most commonly used semantics and application interfaces as much as possible. To this end, mTCP provides a socket API and an event-driven programming interface.

User-level socket API: We provide a BSD-like socket interface; for each BSD socket function, we have a corresponding function call (*e.g.*, `accept()` becomes `mtcp_accept()`). In addition, we provide functionalities that are frequently used with sockets, e.g., fcntl and ioctl, for setting the socket as nonblocking or getting/setting the socket buffer size. To support various applications that require inter-process communication using `pipe()`, we also provide `mtcp_pipe()`.

The socket descriptor space in mTCP (including the fds of `pipe()` and `epoll()`) is local to each mTCP thread; each mTCP socket is associated with a thread context. This allows parallel socket creation from multiple threads by removing lock contention on the socket descriptor space. We also relax the semantics of `socket()` such that it returns any available socket descriptor instead of the minimum available fd. This reduces the overhead of finding the minimum available fd.

User-level event system: We provide an `epoll()`-like event system. While our event system aggregates the events from multiple flows for batching effects, we do not require any modification in the event handling logic. Applications can fetch the events through `mtcp_epoll_wait()` and register events through `mtcp_epoll_ctl()`, which correspond to `epoll_wait()` and `epoll_ctl()` in Linux. Our current `mtcp_epoll()` implementation supports events from mTCP sockets (including listening sockets) and pipes. We plan to integrate other types of events (*e.g.*, timers) in the future.

[4] This optimization can potentially make the system more vulnerable to attacks, such as SYN flooding. However, existing solutions, such as SYN cookies, can be used to mitigate the problem.

Applications: mTCP integrates all techniques known at the time of this writing without requiring substantial kernel modification while preserving the application interface. Thus, it allows applications to easily scale their performance without modifying their logic. We have ported many applications, including `lighttpd`, `ab`, and `SSLShader` to use mTCP. For most applications we ported, the number of lines changed were less than 100 (more details in Section 4). We also demonstrate in Section 5 that a variety of applications can directly enjoy the performance benefit by using mTCP.

However, this comes with a few trade-offs that applications must consider. First, the use of shared memory space offers limited protection between the TCP stack and the application. While the application cannot directly access the shared buffers, bugs in the application can corrupt the TCP stack, which may result in an incorrect behavior. Although this may make debugging more difficult, we believe this form of fate-sharing is acceptable since users face a similar issue in using other shared libraries such as dynamic memory allocation/deallocation. Second, applications that rely on the existing socket fd semantics must change their logic. However, most applications rarely depend on the minimum available fd at socket(), and even if so, porting them will not require significant code change. Third, moving the TCP stack will also bypass all existing kernel services, such as the firewall and packet scheduling. However, these services can also be moved into the user-level and provided as application modules. Finally, our prototype currently only supports a single application due to the limitation of the user-level packet I/O system. We believe, however, that this is not a fundamental limitation of our approach; hardware-based isolation techniques such as VMDq [5] and SR-IOV [13] support multiple virtual guest stacks inside the same host using multiple RX/TX queues and hardware-based packet classification. We believe such techniques can be leveraged to support multiple applications that share a NIC port.

4 Implementation

We implement 11,473 lines of C code (LoC), including packet I/O, TCP flow management, user-level socket API and event system, and 552 lines of code to patch the PSIO library.[5] For threading and thread synchronization, we use `pthread`, the standard POSIX thread library [11].

Our TCP implementation follows RFC793 [17]. It supports basic TCP features such as connection management, reliable data transfer, flow control, and congestion control. For reliable transfer, it implements cumulative acknowledgment, retransmission timeout, and fast retransmission. mTCP also implements popular options such as timestamp, Maximum Segment Size (MSS), and window scaling. For

[5] The number is counted by SLOCCount 2.26.

congestion control, mTCP implements NewReno [10], but it can easily support other mechanisms like TCP CUBIC [26]. For correctness, we have extensively tested our mTCP stack against various versions of Linux TCP stack, and have it pass stress tests, including cases where a large number of packets are lost or reordered.

4.1 mTCP Socket API

Our BSD-like socket API takes on per-thread semantics. Each mTCP socket function is required to have a context, `mctx_t`, which identifies the corresponding mTCP thread. Our event notification function, `mtcp_epoll`, also enables easy migration of existing event-driven applications. Listing 1 shows an example mTCP application.

```
mctx_t mctx = mtcp_create_context();
ep_id = mtcp_epoll_create(mctx, N);
mtcp_listen(mctx, listen_id, 4096);
while (1) {
  n=mtcp_epoll_wait(mctx,ep_id,events,N,-1);
  for (i = 0; i < n; i++) {
    sockid = events[i].data.sockid;
    if (sockid == listen_id) {
      c = mtcp_accept(mctx, listen_id, NULL);
      mtcp_setsock_nonblock(mctx, c);
      ev.events = EPOLLIN | EPOLLOUT;
      ev.data.sockid = c;
      mtcp_epoll_ctl(mctx, ep_id,
          EPOLL_CTL_ADD, c, &ev);
    } else if (events[i].events == EPOLLIN) {
      r = mtcp_read(mctx, sockid, buf, LEN);
      if (r == 0)
        mtcp_close(mctx, sockid);
    } else if (events[i].events == EPOLLOUT){
      mtcp_write(mctx, sockid, buf, len);
    }
  }
}
```

Listing 1: Sample mTCP application.

mTCP supports `mtcp_getsockopt()` and `mtcp_setsockopt()` for socket options, and `mtcp_readv()` and `mtcp_writev()` for scatter-gather I/O as well.

4.2 Porting Existing Applications

We ported four different applications to mTCP.

Web server (lighttpd-1.4.32): Lighttpd is an open-sourced single-threaded web server that uses event-driven I/O for servicing client requests. We enabled multi-threading to support a per-core listen socket and ported it to mTCP. We changed only ∼65 LoC to use mTCP-specific event and socket function calls. For multi-threading, a total of ∼800 lines[6] were modified out of lighttpd's ∼40,000 LoC.

We also ported lighttpd to MegaPipe for comparison. Because its API is based on the I/O completion model, the porting required more effort as it involved revamping lighttpd's event-based `fdevent` backend library; an additional 126 LoC were required to enable MegaPipe I/O from the multi-threaded version.

Apache benchmarking tool (ab-2.3): ab is a performance benchmarking tool that generates HTTP requests. It acts as a client to measure the performance of a Web server. Scaling its performance is important because saturating a 10 Gbps port with small transactions requires multiple machines that run ab. However, with mTCP we can reduce the number of machines by more than a factor of 4 (see Section 5.3).

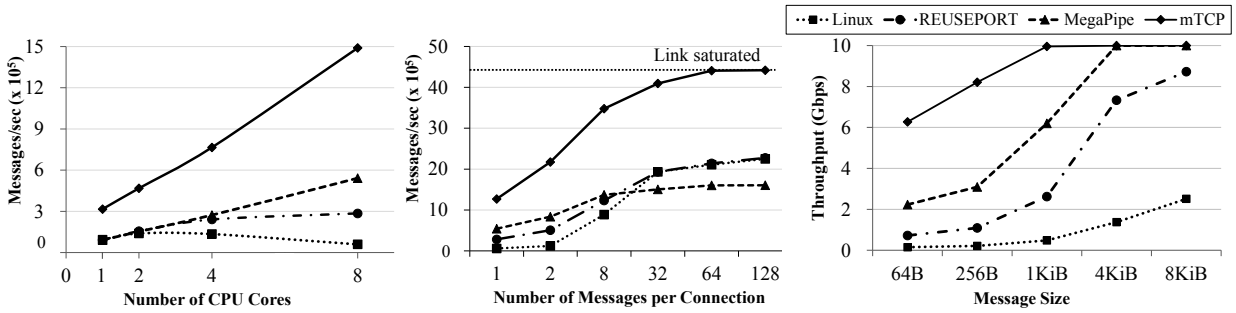
Porting ab was similar to porting lighttpd since ab is also single-threaded. However, ab uses the Apache Portable Runtime (APR) library [16] that encapsulates socket function calls, so we ported the APR library (version 1.4.6) to use mTCP. We modified 29 lines of the APR library (out of 66,493 LoC), and 503 lines out of 2,319 LoC of the ab code for making it multi-threaded.

SSL reverse proxy (SSLShader): SSLShader is a high-performance SSL reverse proxy that offloads crypto operations to GPUs [32]. For small-file workloads, SSLShader reports the performance bottleneck in TCP, spending over 60% CPU cycles in the TCP stack, under-utilizing the GPU. Porting SSLShader to mTCP was straightforward since SSLShader was already multi-threaded and uses `epoll()` for event notification. Besides porting socket function calls, we also replace `pipe()` with `mtcp_pipe()`, which is used to notify the completion of crypto operations by GPU threads. Out of 6,618 lines of C++ code, only 43 lines were modified to use mTCP. It took less than a day to port to mTCP and to finish basic testing and debugging.

Realistic HTTP replay client/server (WebReplay): WebReplay is a pair of client and server programs that reproduces realistic HTTP traffic based on the traffic log collected at a 10 Gbps backhaul link in a large cellular network [45]. Each line in the log has a request URL, a response size, start and end timestamps, and a list of SHA1 hashes of the 4KB content chunks of the original response. The client generates HTTP requests on start timestamps. Using the content hashes, the server dynamically generates a response that preserves the redundancy in the original traffic; the purpose of the system is to reproduce Web traffic with a similar amount of redundancy as the original. Using this, one can test the correctness and performance of network redundancy elimination (NRE) systems that sit between the server and the client. To simulate the traffic at a high speed, however, the WebReplay server must handle 100Ks of concurrent short connections, which requires high TCP performance.

WebReplay is multi-threaded and uses the `libevent` library [6] which in turn calls `epoll()` for event notification. Porting it to mTCP was mostly straightforward in

[6] Some global variables had to be localized to avoid race conditions.



(a) Different number of cores (b) Different number of messages (c) Different sizes of messages

Figure 7: Performance of short TCP connections with 64B messages. (a) and (c) use one message per connection.

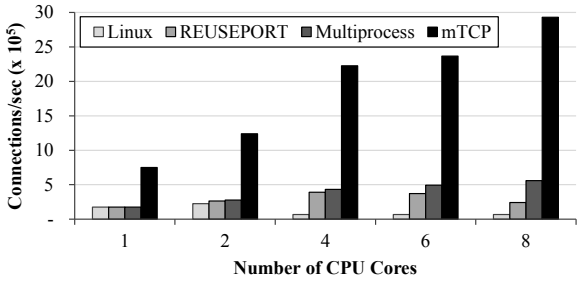


Figure 8: Comparison of connection accept throughputs.

that it only required replacing the socket and libevent calls with the corresponding mTCP API. We modified 44/37 LoC out of 1,703/1,663 lines of server and client code, respectively.

5 Evaluation

We answer three questions in this section:

1. *Handling short TCP transactions*: Does mTCP provide high-performance in handling short transactions? In Section 5.1, we show that mTCP outperforms MegaPipe and Linux (w/o `SO_REUSEPORT`) by 3x and 25x, respectively; mTCP connection establishment alone is 13x and 5x faster than Linux and MegaPipe, respectively.
2. *Correctness:* Does mTCP provide correctness without introducing undesirable side-effects? Section 5.2 shows that mTCP provide fairness and does not introduce long latency.
3. *Application performance*: Does mTCP benefit real applications under realistic workloads? In Section 5.3, we show that mTCP increases the performance of various applications running realistic workload by 33% to 320%.

Experiment Setup: We compare mTCP on Linux 2.6.32 with the TCP stack on the latest Linux kernel (version 3.10.12, with and without `SO_REUSEPORT`) as well as MegaPipe on Linux 3.1.3. We use a machine with one 8-core CPU (Intel Xeon E5-2690 @ 2.90 GHz), 32 GB RAM, and an Intel 10 GbE NIC as a server, and use up to 5 clients of the same type to saturate the server. While mTCP itself does not depend on the kernel version, the underlying PSIO library currently works on Linux 2.6.32. For Linux, we use ixgbe-3.17.3 as the NIC driver.

5.1 Handling Short TCP Transactions

Message benchmark: We first show mTCP's scalability with a benchmark for a server sending a short message as a response. All servers are multi-threaded with a single listening port. Our workload generates a 64 byte message per connection, unless otherwise specified. The performance result is averaged over a one minute period in each experiment. Figure 7 shows the performance as a function of the number of CPU cores, the number of messages per connection (MPC), and message size.

Figure 7(a) shows that mTCP scales almost linearly with the number of CPU cores. Linux without `SO_REUSEPORT` ('Linux') shows poor scaling due to the shared accept queue, and Linux with `SO_REUSEPORT` ('REUSEPORT') scales but not linearly with the number of cores. At 8 cores, mTCP shows 25x, 5x, 3x higher performance over Linux, REUSEPORT, and MegaPipe, respectively.

Figure 7(b) shows that the mTCP's benefit still holds even when persistent connections are used. mTCP scales well as the number of messages per connection (MPC) increases, and it nearly saturates the 10G link from 64 MPC. However, the performance of the other systems almost flattens out well below the link capacity. Even at 32 MPC, mTCP outperforms all others by a significant margin (up to 2.7x), demonstrating mTCP's effectiveness in handling small packets.

Finally, Figure 7(c) shows the throughput by varying the message size. mTCP's performance improvement is more noticeable with small messages, due to its fast processing of small packets. However, both Linux servers fail to saturate the 10 Gbps link for any message size. MegaPipe saturates the link from 4KiB, and mTCP can saturate the link from 1KiB messages.

Connection accept throughput: Figure 8 compares connection throughputs of mTCP and Linux servers. The

		Min	Mean	Max	Stdev
Connect	Linux	0	36	63,164	511.6
	mTCP	0	1	500	1.1
Processing	Linux	0	87	127,323	3,217
	mTCP	1	13	2,323	9.7
Total	Linux	0	124	127,323	3,258
	mTCP	9	14	2,348	9.8

Table 2: Distribution of response times (ms) for 64B HTTP messages for 10 million requests (8K concurrency).

server is in a tight loop that simply accepts and closes new connections. We close the connection by sending a reset (`RST`) to prevent the connection from lingering in the TIME_WAIT state. To remove the bottleneck from the shared fd space, we add 'Multiprocess' which is a multi-process version of the REUSEPORT server. mTCP shows 13x, 7.5x, 5x performance improvement over Linux, REUSEPORT, and Multiprocess, respectively. Among the Linux servers, the multi-process version scales the best while other versions show a sudden performance drop at multiple cores. This is due to the contention on the shared accept queue as well as shared fd space. However, Multiprocess shows limited scaling, due to the lack of batch processing and other inefficiencies in the kernel.

5.2 Fairness and Latency

Fairness: To verify the throughput fairness among mTCP connections, we use `ab` to generate 8K concurrent connections, each downloading a 10 MiB file to saturate a 10 Gbps link. On the server side, we run `lighttpd` with mTCP and Linux TCP. We calculate Jain's Fairness Index with the (average) transfer rate of each connection. As the value gets closer to 1.0, it shows better fairness. We find that Linux and mTCP show 0.973 and 0.999, respectively. mTCP effectively removes the long tail in the response time distribution, whereas Linux often drops `SYN` packets and enters a long timeout.

Latency: Since mTCP relies heavily on batching, one might think it may introduce undesirably long latency. Table 2 shows the latency breakdown when we run ab with 8K concurrent connections against the 64B message server. We generate 10 million requests in total. Linux and mTCP versions respectively achieve 45K and 428K transactions per second on average. As shown in the table, mTCP slightly increases the minimum (9 ms vs. 0 ms) and the median (13 ms vs. 3 ms) response times. However, the mean and maximum response times are 8.8x and 54.2x smaller than those of Linux, while handling 9.5x more transactions/sec. In addition, the standard deviation of the response times in mTCP is much smaller, implying that mTCP produces more predictable response times, which is becoming increasingly important for modern datacenter applications [33].

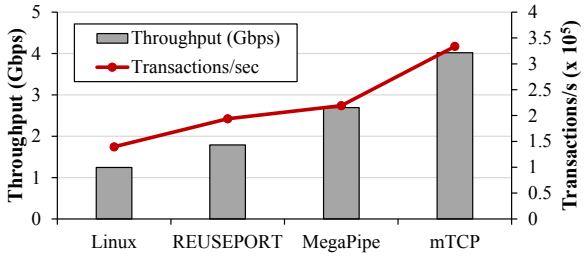


Figure 9: Performance of four versions of `lighttpd` for static file workload from SpecWeb2009.

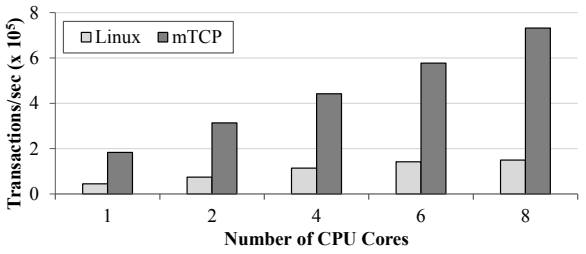


Figure 10: Performance of ab as a function of the number of cores. The file size is 64B and 8K concurrent connections are used.

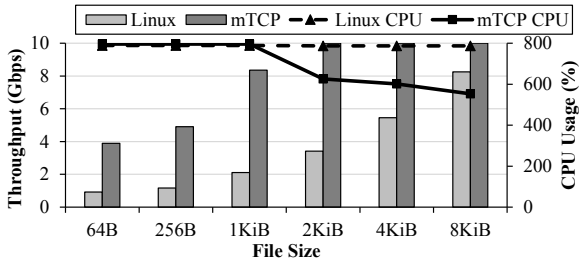


Figure 11: Performance of ab as a function of a file size. The number of cores is set to 8 with 8K concurrent connections.

5.3 Application Performance

We now demonstrate the performance improvement for existing applications under realistic workloads.

lighttpd and ab: To measure the performance of lighttpd in a realistic setting, we use the static file workload extracted from SpecWeb2009 and compare the performance of different lighttpd versions ported to use mTCP, MegaPipe, and Linux with and without `SO_REUSEPORT`. Figure 9 shows that mTCP improves the throughput of lighttpd by 3.2x, 2.2x, 1.5x over Linux, REUSEPORT, and MegaPipe, respectively. Even though the workload fits into the memory, we find that heavy system calls for VFS operations limit the performance.

We now show the performance of ab. Figure 10 shows the performance of Linux-based and mTCP-based ab when varying the number of CPU cores when fetching a 64 byte file over HTTP. The scalability of Linux is limited, since it shares the fd space across multiple threads.

Figure 10 shows the performance of ab and the corresponding CPU utilization when varying the file size from 64 bytes to 8 KiB. From 2 KiB, mTCP saturates the link.

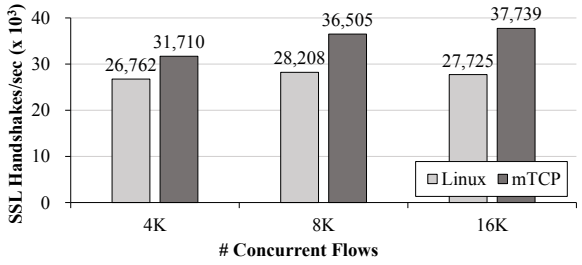


Figure 12: SSL handshake throughputs of SSLShader with a different levels of concurrency.

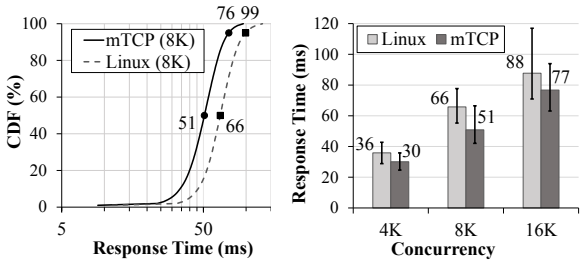


Figure 13: HTTPS response time distributions of SSLShader on Linux and mTCP stacks. We use 8K concurrent connections in the left graph, and mark median and 95th-percentile numbers.

# of copies	1	2	3	4	5	6	7
Linux (ms)	27.8	29.0	45.8	**1175.1**	-	-	-
mTCP (ms)	0.5	0.9	2.6	8.1	17.5	37.1	79.8

Table 3: Averages of extra delays (in ms) from the original response times when replaying *n* copies of the log concurrently.

	# of concurrent connections	# of new connections per second	Bandwidth (Gbps)
Mean	23,869	14,425	2.28
Min	20,608	12,755	1.79
Max	25,734	15,440	3.48

Table 4: Log statistics for WebReplay.

At the same time, mTCP's event-driven system saves CPU cycles.

When testing mTCP with long-lived connections (not shown in the figure), we find that it consumes more CPU cycles than Linux. mTCP shows a CPU utilization of 294% compared to 80% for Linux-3.10.12 when serving 8,000 concurrent connections, each transferring a 100 MiB file. This is because we did not fully utilize modern NIC features, such as TCP checksum offload, large segmentation offload (LSO), and large receive offload (LRO). However, we believe that mTCP can easily incorporate these features in the future.

SSLShader: We benchmark the performance of the SSLShader with one NVIDIA GPU (Geforce GTX 580) on our server. We use mTCP-based lighttpd as a server and ab as a client. On a separate machine, we run SSLShader as a reverse proxy to handle HTTPS transactions. SSLShader receives an HTTPS request from ab and decrypts the request. It then fetches the content from lighttpd in plaintext, encrypts the response using SSL, and sends it back to the client. We use 1024-bit RSA, 128bit-AES, and HMAC-SHA1 as the cipher suite, which is widely used in practice. To measure the performance of SSL handshakes, we have ab to fetch 1-byte objects through SSLShader while varying the number of concurrent connections.

Figure 12 shows that mTCP improves the performance over the Linux version by 18% to 33%. As the concurrency increases, the benefit of mTCP grows, since mTCP scales better with a large number of concurrent connections. Figure 13 indicates that mTCP also reduces the response times compared to the Linux version. Especially, mTCP reduces the tail in the response time distribution over large concurrent connections with a smaller variance, as is also shown in Section 5.2.

WebReplay: We demonstrate that mTCP improves the performance of a real HTTP traffic replayer. We focus on the server's performance improvement because it performs more interesting work than the client. To fully utilize the server, we use four 10 Gbps ports and connect each port to a client. The workload (HTTP requests) generated by the clients is determined by the log captured at a cellular backhaul link [45]. We replay the log for three minutes at a peak time (at 11 pm on July 7, 2012) during the measurement period. The total number of requests within the timeframe is 2.8 million with the median and average content size as 1.7 KB and 40 KB. Table 4 summarizes the workload that we replay. Unfortunately, we note that the trace we replay does not simulate the original traffic perfectly since a longer log is required to effectively simulate idle connections. Actually, the original traffic had as much as 270K concurrent connections with more than 1 million TCP connections created per minute. To simulate such a load, we run multiple copies of the same log concurrently for this experiment.

Table 3 compares the averages of extra delays from the original response times when we replay *n* copies of the log concurrently with Linux and mTCP-based WebReplayer. We find that the Linux server works fine up to the concurrent run of three copies, but the average extra delay goes up beyond 1 second at four copies. In contrast, mTCP server finishes up to seven copies while keeping the average extra delay under 100 ms. The main cause for the delay inflation in the Linux version is the increased number of concurrent TCP transactions, which draws the bottleneck in the TCP stack.

6 Related Work

We briefly discuss previous work related to mTCP.

System call and I/O batching: Frequent system calls are often the performance bottleneck in busy servers.

FlexSC [40] identifies that CPU cache pollution can waste more CPU cycles than the user/kernel mode switch itself. They batch the system calls by having user and kernel space share the syscall pages, which allows significant performance improvement for event-driven servers [41]. MegaPipe employs socket system call batching in a similar way, but it uses a standard system call interface to communicate with the kernel [28].

Batching also has been applied to packet I/O to reduce the per-packet processing overhead. PacketShader I/O engine [27] reads and writes packets in batches and greatly improves the packet I/O performance, especially for small packets. Packet I/O batching reduces the interrupt, DMA, IOMMU lookup, and dynamic memory management overheads. Similar approaches are found in other high-performance packet I/O libraries [4, 7, 39].

In contrast, mTCP eliminates socket system calls by running the TCP stack in the user level. Also, it enforces batching from packet I/O and TCP processing up to user applications. Unlike FlexSC and MegaPipe, batching in mTCP is completely transparent without requiring kernel or user code modification. Moreover, it performs batching in both directions (e.g., packet TX and RX, application to TCP and TCP to application).

Connection locality on multicore systems: TCP performance can be further optimized on multiprocessors by providing connection locality on the CPU cores [37]. By handling all operations of same connection on the same core, it can avoid inter-core contention and unnecessary cache pollution. mTCP adopts the same idea, but applies it to both flow- and packet-level processing.

User-level TCP stacks: There have been several attempts to move the entire networking stack from the kernel to the user level [22, 24, 25, 42]. These are mainly (1) to ease the customizing and debugging of new network protocols or (2) to accelerate the performance of existing protocols by tweaking some internal variables, such as the TCP congestion control parameters. They focus mostly on providing a flexible environment for user-level protocol development or for exposing some in-kernel variables safely to the user level. In contrast, our focus is on building a user-level TCP stack that provides high scalability on multicore systems.

Light-weight networking stacks: Some applications avoid using TCP entirely for performance reasons. High performance key-value systems, such as memcached [9], Pilaf [35], and MICA [34], either use RDMA or UDP-based protocols to avoid the overhead of TCP. However, these solutions typically only apply to applications running inside a datacenter. Most user-facing applications must still rely on TCP.

Multikernel: Many research efforts enhance operating system scalability for multicore systems [19, 20, 44]. Barrelfish [19] and fos [44] separate the kernel resources for each core by building an independent system that manages per-core resources. For efficient inter-core communication, they use asynchronous message passing. Corey [20] attempts to address the resource sharing problem on multicore systems by having the application explicitly declare shared and local resources across multiple cores. It enforces the default policy of having private resources for a specific core to minimize unnecessary contention. mTCP borrows the concept of per-core resource management from Barrelfish, but allows efficient sharing between application and mTCP threads with lock-free data structures.

Microkernels: The microkernel approach bears similarity with mTCP in that the operating system's services run within the user level [23, 30, 38]. Exokernel [23], for example, provides a minimal kernel and low-level interfaces for accessing hardware while providing protection. It exposes low-level hardware access directly to the user level so that applications perform their own optimizations. This is conceptually similar to mTCP's packet I/O library that directly accesses the NIC. mTCP, however, integrates flow-level and packet-level event batch processing to amortize the context switch overhead, which is often a critical bottleneck for microkernels.

7 Conclusion

mTCP is a high-performance user-level TCP stack designed for multicore systems. We find that the Linux kernel still does not efficiently use the CPU cycles in processing small packets despite recent improvements, and this severely limits the scalability of handling short TCP connections. mTCP unleashes the TCP stack from the kernel and directly delivers the benefit of high-performance packet I/O to the transport and application layer. The key enabler is transparent and bi-directional batching of packet- and flow-level events, which amortizes the context switch overhead over a batch of events. In addition, the use of lock-free data structures, cache-aware thread placement, and efficient per-core resource management contributes to mTCP's performance. Finally, our evaluation demonstrates that porting existing applications to mTCP is trivial and mTCP improves the performance of existing applications by up to 320%.

Acknowledgement

We would like to thank our shepherd George Porter and anonymous reviewers from NSDI 2014 for their valuable comments. We also thank Sangjin Han for providing the MegaPipe source code, and Sunil Pedapudi and Jaeheung Surh for proofreading the final version. This research is supported by the National Research Foundation of Korea (NRF) grant #2012R1A1A1015222 and #2013R1A1A1076024.

References

[1] Facebook. https://www.facebook.com/.

[2] Google. https://www.google.com/.

[3] How long does it take to make a context switch? http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html.

[4] Intel DPDK: Data Plane Development Kit. http://dpdk.org/.

[5] Intel VMDq Technology. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/vmdq-technology-paper.pdf.

[6] Libevent. http://libevent.org/.

[7] Libzero for DNA: Zero-copy flexible packet processing on top of DNA. http://www.ntop.org/products/pf_ring/libzero-for-dna/.

[8] Lighttpd. http://www.lighttpd.net/.

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

[10] The NewReno modification to TCP's fast recovery algorithm. http://www.ietf.org/rfc/rfc2582.txt.

[11] The open group base specifications issue 6, IEEE Std 1003.1. http://pubs.opengroup.org/onlinepubs/007904975/basedefs/pthread.h.html.

[12] Packet I/O Engine. http://shader.kaist.edu/packetshader/io_engine/.

[13] PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. http://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf.

[14] The SO_REUSEPORT socket option. https://lwn.net/Articles/542629/.

[15] The Apache HTTP Server Project. http://httpd.apache.org/.

[16] The Apache Portable Runtime Project. http://apr.apache.org/.

[17] Transmission control protocol. http://www.ietf.org/rfc/rfc793.txt.

[18] Twitter. https://twitter.com/.

[19] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS symposium on Operating systems principles (SOSP)*, 2009.

[20] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[21] 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 Symposium on Operating Systems Principles (SOSP)*, 2009.

[22] D. Ely, S. Savage, and D. Wetherall. Alpine: a user-level infrastructure for network protocol development. In *Proceedings of the conference on USENIX Symposium on Internet Technologies and Systems (USIT)*, 2001.

[23] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the ACM symposium on Operating systems principles (SOSP)*, 1995.

[24] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems (TOCS)*, 20(1):49–83, Feb. 2002.

[25] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deploying safe user-level network services with ictcp. In *Proceedings of the conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, 2004.

[26] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[27] S. Han, K. Jang, K. Park, and S. B. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2010.

[28] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: a new programming interface for scalable network I/O. In *Proceedings of the USENIX*

conference on Operating Systems Design and Implementation (OSDI), 2012.

[29] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[30] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. *SIGOPS Oper. Syst. Rev.*, 31(5):66–77, Oct. 1997.

[31] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *Proceedings of the ACM SIGCOMM conference on Internet measurement conference (IMC)*, 2011.

[32] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *Proceedings of the USENIX conference on Networked systems design and implementation (NSDI)*, 2011.

[33] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, 2012.

[34] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[35] 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 (ATC)*, 2013.

[36] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at Facebook. In *Proceedings of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[37] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the ACM european conference on Computer Systems (EuroSys)*, 2012.

[38] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A system software kernel. In *Proceedings of the Computer Society International Conference (COMPCON)*, 1989.

[39] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the USENIX conference on Annual Technical Conference (ATC)*, 2012.

[40] L. Soares and M. Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[41] L. Soares and M. Stumm. Exception-less system calls for event-driven servers. In *Proceedings of the USENIX conference on USENIX annual technical conference (ATC)*, 2011.

[42] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing network protocols at user level. *IEEE/ACM Transactions on Networking (TON)*, 1(5):554–565, Oct. 1993.

[43] V. Vasudevan, D. G. Andersen, and M. Kaminsky. The case for VOS: the vector operating system. In *Proceedings of the USENIX conference on Hot topics in operating systems (HotOS)*, 2011.

[44] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, Apr. 2009.

[45] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park. Comparison of caching strategies in modern cellular backhaul networks. In *Proceeding of the annual international conference on Mobile systems, applications, and services (MobiSys)*, 2013.

Warranties for Faster Strong Consistency

Jed Liu Tom Magrino Owen Arden Michael D. George Andrew C. Myers

Cornell University Department of Computer Science

{liujed,tmagrino,owen,mdgeorge,andru}@cs.cornell.edu

Abstract

We present a new mechanism, warranties, to enable building distributed systems with linearizable transactions. A warranty is a time-limited assertion about one or more distributed objects. These assertions generalize optimistic concurrency control, improving throughput because clients holding warranties need not communicate to verify the warranty's assertion. Updates that might cause an active warranty to become false are delayed until the warranty expires, trading write latency for read latency. For workloads biased toward reads, warranties improve scalability and system throughput. Warranties can be expressed using language-level computations, and they integrate harmoniously into the programming model as a form of memoization. Experiments with some nontrivial programs demonstrate that warranties enable high performance despite the simple programming model.

1 Introduction

Although the trend for many systems has been to weaken consistency in order to achieve greater scalability, strong consistency is critical when lives or money are at stake. Examples include systems for medical information, banking, payment processing, and the military.

Users of weakly consistent systems may be confused by applications that appear buggy. Moreover, weak consistency can significantly complicate the job of developers who try to detect and repair inconsistencies at the application layer. Consistency failures at the bottom of a software stack can percolate up through the stack and affect higher layers in unpredictable ways, requiring defensive programming.

The need for strong consistency and a simple programming model has kept databases with ACID transactions in business. However, transactions are often considered to have poor performance, especially in a distributed setting. In this work, we introduce *warranties*, a new mechanism that improves the performance of transactions, enabling them to scale better both with the number of application clients and with the number of persistent storage nodes. Warranties help avoid the unfortunate choice between consistency and performance.

A warranty is a limited guarantee that some (potentially complex) assertion remains true regarding the state of a distributed system. The guarantee is limited in that it eventually expires. But during the term of the warranty, the application can safely use it to perform computation locally without communicating with the server that issued the warranty. Warranties are like leases [21] in that they have a duration, but differ in that they make a logical assertion rather than conferring the right to use objects.

Warranties support implementing linearizable transactions by generalizing optimistic concurrency control (OCC) [16, 32]. OCC permits aggressive caching of objects read by transactions, but requires communicating with storage servers to ensure objects are up to date. Since warranties can express guarantees that objects are up to date, communication can be reduced. Warranties are particularly effective in the common case of high read contention, where many clients want to share the same popular—yet mutable—data.

More generally, warranties can contain an assertion that the results of a language-level computation has not changed. These *computation warranties* offer a form of distributed memoization, allowing clients to share cached computation in the manner often currently done using distributed caches such as `memcached`—but with strong consistency guarantees that are currently lacking.

Overall, warranties offer a new way to ameliorate the tension between consistency and scalability in distributed applications.

The remainder of this paper is structured as follows. Section 2 discusses our system model and relevant background material. Section 3 presents the warranty abstraction in more detail, and discusses its connection to leases. Section 4 explains in more detail how optimistic transactions are implemented using warranties. The mechanisms needed for computation warranties are explored in Section 5. Our implementation using the Fabric distributed object system is described in Section 6. The evaluation in Section 7 shows that warranties significantly improve the performance of both representative benchmarks and a substantial real-world program. Related work is discussed more broadly in Section 8, and we conclude in Section 9.

2 Background and system model

We assume a distributed system in which each node serves one of two main roles: *client nodes* perform computations locally using persistent data from elsewhere, and *persistent storage nodes* (*stores*) store the persistent data. Client nodes obtain copies of persistent data from stores, perform computations, and send updates to the persistent data back to the stores. For example, the lower two tiers of the traditional three-tier web application match this description: application servers are the clients and database servers are the stores.

Our goal is a simple programming model for application programmers, offering strong consistency so they do not need to reason about inconsistent or out-of-date state. In particular, we want linearizability [25], so each committed transaction acts as though it executes atomically and in logical isolation from the rest of the system. Linearizability strengthens serializability [42, 8] to offer external consistency.

A partially successful attempt at such a programming model is the Java Persistence API (JPA) [12], which provides an object–relational mapping (ORM) that translates accesses to language-level objects into accesses to underlying database rows. JPA implementations such as Hibernate [27] and EclipseLink [15] are widely used to build web applications. However, we want to improve on both the consistency and performance of JPA.

We assume that the working set of both clients and stores fits in the node's memory. This assumption is reasonable for many applications, though not for large-scale data analytics applications, which we do not target.

In a distributed transaction system using OCC (e.g., Thor [37]) clients fetch and then cache persistent objects across transactions. Optimistic caching allows client transactions to largely avoid talking to stores until commit time, unlike with pessimistic locking. The system is faster because persistent data is replicated at the memories of potentially many client nodes. However, care must be taken to avoid inconsistency among the cached copies.

Because of its performance advantages, optimism has become increasingly popular for JPA applications, where the best performance is usually achieved through an "optimistic locking" mode that appears to provide strong consistency in some but not all implementations of JPA.[1]

To provide strong consistency, OCC logs reads and writes to objects. As part of committing the transaction, clients send the transaction log to stores involved in the transaction. The stores then check that the state of each object read matches that in the store (typically by checking version numbers), and then perform updates.

To scale up a distributed computing system of this sort, it is important to be able to add storage nodes across which persistent data and client requests can be distributed. As long as a given client transaction accesses data at just one store, and load is balanced across the stores, the system scales well: each transaction can be committed with just one round trip between the client and the accessed store.

In general, however, transactions access information located at multiple stores. For example, consider a web shopping application. A transaction that updates the user's shopping cart may still need to read information shared among many users of the system, such as details of the item purchased.

Accessing multiple stores hurts scalability. To commit such a transaction serializably, it must be known at commit time that all objects read during the transaction were up to date. A two-phase commit (2PC) is used to ensure this is the case. In the first phase (the prepare phase), each store checks that the transaction can be committed and if so, readies the updates to be committed; it then reports to the coordinator whether the transaction is serializable. If the transaction can be committed at every store, all stores are told to commit in the commit phase. Otherwise, the transaction is aborted and its effects are rolled back.

If popular, persistent data is accessed by many clients, the read contention between clients interferes with scalability. Each client committing a transaction must execute a prepare phase at the store of that data. The work done by the prepare phase consists of *write prepares* done on objects that have been updated by the transaction, and *read prepares* on objects that have been read. In both cases, the object is checked to ensure that the version used was up to date.

Read prepares can make the nodes storing popular objects into bottlenecks even when those objects are rarely updated. This is a fundamental limit on scalability of OCC, so a key benefit of warranties is addressing this performance bottleneck. An alternative strategy would be to replicate popular objects across multiple nodes, but keeping replicas in agreement is very costly.

3 The warranty abstraction

A warranty is a time-limited assertion about the state of the system: it is guaranteed to remain true for some fixed period of time. Warranties improve scalability for two reasons: first, because they reduce or eliminate the work needed for read prepares; second, more generally, they enable the distributed caching of computations and enforce a more semantic notion of consistency.

Because warranties make guarantees about the state of the system, they allow transactions to be committed without preparing reads against the objects covered by

[1] The term "optimistic locking" is misleading; locking occurs only during transaction commit. The JPA 2 specification appears to guarantee that objects *written* by a transaction are up to date—but, unfortunately, not the objects *read* unless explicitly locked. Implementations differ in interpretation.

warranties. When all reads to a store involved in a transaction are covered by warranties, that store need not be contacted. Consequently, two-phase commit can be reduced to a one-phase commit in which the prepare and commit phases are consolidated, or even to a zero-phase commit in which no store need be contacted. The result is significantly improved performance and scalability.

In this section, we give a more detailed overview of how warranties work.

- Simple *state warranties* generalize OCC (§3.1) and also, to some extent, leases (§3.2).
- Updates to the system are prevented from invalidating warranties (§3.3), with implications for performance (§3.4).
- Warranty assertions can be expressive, enabling distributed caching of computed results (§3.5).
- Warranties are requested by clients (§3.6) and generated on demand by stores (§3.7).
- Warranties are distributed throughout the system to clients that need them (§3.9).
- The term of warranties can be set automatically, based on run-time measurements (§3.8).

3.1 State warranties

The simplest form of warranty is a *state warranty*, an assertion that the concrete state of an object has a particular value. A warranty is guaranteed to be true (*active*) during the warranty's *term*. At the end of its term, the warranty *expires* and is no longer guaranteed to be true.

For example, a state warranty for an object representing a bank account might be ⟨`assert = {name = "John Doe", bal = $20,345}, exp = 1364412767.1`⟩. Here, the field `assert` specifies the state of the object, and the field `exp` is the time that the warranty expires.

A warranty is issued by a store, and times appearing in the warranties are measured by the clock of the store that issued the warranty. We assume that clocks at nodes are loosely synchronized; well-known methods exist to accomplish this [40].

If a warranty expires before the transaction commits, the warranty may continue to be *valid*, meaning that the assertion it contains is still true even though clients cannot rely on its remaining true. Clients can, however, still use the warranty optimistically and check at commit time that the warranty remains valid.

As can be seen, state warranties generalize optimistic concurrency control. Ordinary OCC equates to always receiving a zero-length warranty for the state of the object read, and using that expired warranty optimistically.

3.2 Warranties vs. leases

Leases [21] have been used in many systems (e.g., [51, 2]) to improve performance. Warranties exploit the key insight of leases that time-limited guarantees increase scalability by reducing coordination overhead. As defined originally by Gray and Cheriton, leases confer time-limited *rights* to access objects in certain ways, and must be held by clients in order to perform the corresponding access. Conversely, warranties are time-limited *assertions* about what is *true* in the distributed system, and are not, therefore, held by any particular set of nodes. Unlike with leases, an expired warranty may be used to access an object optimistically. Gray does sketch in his dissertation [20] how read leases might be integrated into an optimistic transaction processing system, but we are not aware of any detailed design or implementation.

Leases and warranties do partly overlap. Since *read leases* on objects effectively prevent modifying object state, they must enforce assertions regarding the state of that data. Therefore, state warranties can be viewed as read leases that are given to many clients and that cannot be relinquished by those clients.

However, we see a fundamental difference between these two perspectives. The value of the warranty (assertion) perspective is that state warranties naturally generalize to expressive assertions over state—in particular, warranties that specify the results of application-defined computations over the state of potentially many objects.

3.3 Defending warranties

Transactions may try to perform updates that affect objects on which active warranties have been issued. Updates cannot invalidate active warranties without potentially violating transactional isolation for clients using those warranties. Therefore, stores must defend warranties against invalidating updates, a process that has no analogue in OCC.

A warranty can be defended against an invalidating update transaction in two ways: the transaction can either be rejected or delayed. If rejected, the transaction will abort and the client must retry it. If delayed, the updating transaction waits until it can be safely serialized. Rejecting the transaction does not solve the underlying problem of warranty invalidation, so delaying is typically the better strategy if the goal is to commit the update. To prevent write starvation, the store stops issuing new warranties until after the commit. The update also shortens the term of subsequent warranties.

3.4 Performance tradeoffs

Using warranties improves read performance for objects on which warranties are issued, but delays writes to these objects. Such a tradeoff appears to be an unavoidable with strong consistency. For example, in conventional

database systems that use pessimistic locking to enforce consistency, readers are guaranteed to observe consistent states, but update transactions must wait until all read transactions have completed and released their locks. With many simultaneous readers, writers can be significantly delayed. Thus, warranties occupy a middle ground between optimism and pessimism, using time as a way to reduce the coordination overhead incurred with locking.

The key to good performance, then, is to issue warranties that are long enough to allow readers to avoid revalidation but not so long that they block writers more than they otherwise would be blocked.

For applications where it is crucial to have both high write throughput and high read throughput to the same object, replication is essential, and the cost of keeping object replicas in sync makes strong consistency infeasible. However, if weak consistency is acceptable, there is a simple workaround: implement replication by explicitly maintaining the state in multiple objects. Writes can go to one or more persistent objects that are read infrequently, and only by a process that periodically copies them (possibly after reconciliation of divergent states) to a frequently read object on which warranties can be issued. This is a much easier programming task than starting from weak consistency and trying to implement strong consistency where it is needed. The only challenging part is reconciliation of divergent replicas, which is typically needed in weakly consistent systems in any case (e.g., [50, 47, 14]).

3.5 Computation warranties

Warranty assertions are not limited to specifying the concrete state of persistent objects. In general, a warranty assertion is an expression in a language that can describe a computation that operates on persistent objects and that can be evaluated at the store. SQL is one query language that fits this description, but in this work, we integrate assertions more tightly with the programming language. Computation warranties provide guarantees about computations described in terms of method calls.

In current distributed applications, it is common to use a distributed cache such as `memcached` [18] to share data and computation across many nodes. For example, web application servers can cache the text of commonly used web pages or content to be included in web pages. Computation warranties can be used to cache such computed results without abandoning strong consistency.

Example: top N items. Many web applications display the top-ranked N items among some large set (such as advertisements, product choices, search results, poll candidates, or game ladder rankings).

Although the importance of having consistent rankings may vary across applications, there are at least some cases in which the right ranking is important and may have monetary or social impact. Election outcomes matter, product rankings can have a large impact on how money is spent, and game players care about ladder rankings. But at present there is no easy and efficient way to ensure that cached computation results are up to date.

To cache the results of such a computation, we might define a computation `top(n, i, j)`, which returns the set `s` of the `n` top-ranked items whose indices in an array of items lie between `i` and `j`. A warranty of the form `s = top(n, 0, num_items)` then allows clients to share the computation of the top-ranked items within the range.

The reason why the `top` function has arguments `i` and `j` is to permit `top` to be implemented recursively and efficiently using results from subranges, on which further warranties are issued. We discuss later in more detail how this approach allows computation warranties to be updated and recomputed efficiently.

Example: airplane seats. Checking whether airplane flights have open seats offers a second example of a computation that can be worth caching. Because the client-side viewer may be sorting lists of perhaps hundreds of potential flights, flights are viewed much more often than their seating is updated. Scalability of the system would be hurt by read prepares.

Efficient searching over suitable flights can be supported by issuing warranties guaranteeing that at least a certain number of seats of a specified type are available; for a suitable constant number of seats n large enough to make the purchase, a warranty of this form works:

$$\texttt{flight.seats\_available(type)} \geq n$$

This warranty helps searching efficiently over the set of flights on which a ticket might be purchased. It does not help with the actual update when a ticket is purchased on a flight. In this case, it becomes necessary to find and update the actual number of seats available. However, this update can be done quickly as long as the update does not invalidate the warranty.

Like state warranties, computation warranties can be used optimistically even if they expire during the transaction. In this case, the dependencies of the computation described in the warranty must be checked at commit time to ensure that the warranty's assertion remains true, just as objects whose state warranties expire before commit time must be checked. A warranty that is revalidated in this fashion can then be issued as a new warranty.

Like active state warranties, active computation warranties must be defended against invalidation by updates. This mechanism is discussed in Section 5.2.

3.6 Programming with warranties

As clients compute, they request warranties as needed. State warranties are requested automatically when objects are newly fetched by a computation. Computation

warranties can also be generated in a natural way, relying on simple program annotations.

Computation warranties explicitly take the form of logical assertions, so they could be requested by using a template for the desired logical assertion. In the airline seat reservation example above, a query of the form `flight.seats_available(type)` $\geq ?$ could be used to find all available warranties matching the query, and at the same time fill in the "?" with the actual value n found in the warranty. In the case where multiple warranties match, a warranty might be chosen whose duration and value of n are "best" according to application-specific criteria.

We pursue a more transparent way to integrate warranty queries into the language, via memoized function calls. For example, we can define a memoized method with the signature `memoized boolean seats_lb(type,` n`)` that returns whether there are at least n seats of the desired type still available on the flight. The keyword `memoized` indicates that its result is to be memoized and warranties are to be issued on its result. To use these warranties, client code uses the memoized method as if it were an ordinary method, as in the following code:

```
for (Flight f : flights)
  if (f.seats_lb(aisle, seats_needed))
    display_flights.add(f)
```

When client code performs a call to a memoized method, the client automatically checks to see if a warranty for the assertion $? =$ `seats_lb(type, n)` has either been received already or can be obtained. If so, the result of the method call is taken directly from the warranty. If no warranty can be found for the method call, the client executes the method directly.

With appropriate language support, the implementation of such a memoized method is also straightforward:

```
memoized boolean seats_lb(Seat t, int n) {
    return seats_available(t) >= n;
}
```

A language that correctly supports transparent OCC already automatically logs the reads and writes performed on objects; this logging already computes the dependencies of computation warranties.

3.7 Generating warranties

Warranties are issued by stores, because stores must know about warranties in order to defend them against updates that might invalidate them. However, for scalability, it is important to avoid giving the store extra load. Therefore, it only makes sense to generate warranties for some objects and computations: those that are used much more frequently than they are invalidated.

For state warranties, the store already has enough information to decide when to generate a warranty for an object, because it sees both when the object is updated and when it is necessary to check that the version of the object read by a client is up to date. State warranties improve performance by removing the need to do version checks on read objects, but at the cost of delaying updates that would invalidate active warranties. This tradeoff makes sense if the version checks are sufficiently more numerous than the updates.

For computation warranties, the store may be able to infer what warranties are needed from client requests, but it makes more sense to have the client do the computational work. Recall that clients that fail to find a suitable warranty compute the warranty assertion themselves. If the assertion is true, it is the basis of a potential warranty that is stored in the client's local cache and reused as needed during the same transaction. As part of committing the transaction, the client sends such potential warranties to the store, which may issue these warranties, both back to this client and to other clients. The decision whether to issue a warranty properly depends on whether issuing the warranty is expected to be profitable.

3.8 Setting warranty terms

Depending on how warranty terms are set, warranties can either improve or hurt performance. However, it is usually possible to automatically and adaptively set warranty terms to achieve a performance increase.

Warranties improve performance by avoiding read prepares for objects, reducing the load on stores and on the network. If *all* read and write prepares to a particular store can be avoided, warranties eliminate the need even to coordinate with that store.

Warranties can hurt performance primarily by delaying writes to objects. The longer a warranty term is, the longer the write is delayed. If warranty terms are set too long, writers may experience unacceptable delays. A good rule of thumb is that we would like writers to be delayed no more than they would be by read locks in a system using pessimistic locks.

Excessively long warranties may also allow readers to starve writers, although starvation is mitigated because new warranties are not issued while writers are blocked waiting for a warranty to expire. Note that with pure OCC, writers can block readers by causing all read prepares to fail [43]; thus, warranties shift the balance of power away from writers and toward readers, addressing a fundamental problem with OCC.

To find the right balance between the good and bad effects of warranties, we take a dynamic, adaptive approach. Warranty terms are automatically and individually set by stores that store the relevant objects. Fortunately, stores observe enough to estimate whether war-

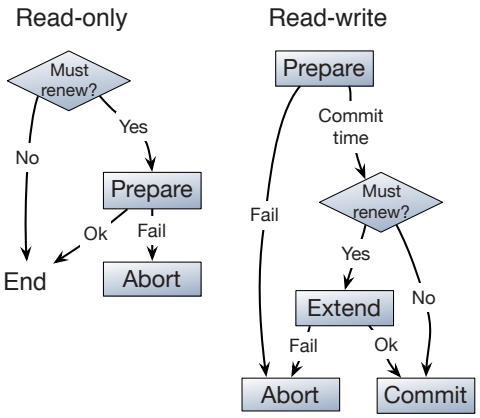


Figure 1: Warranty commit protocol for read-only and read-write transactions.

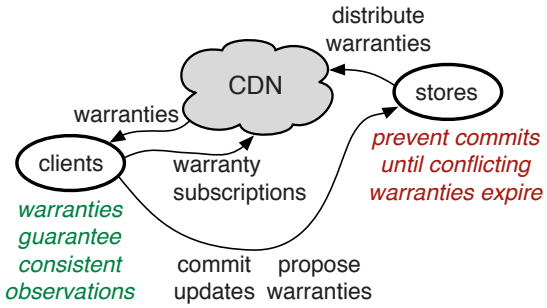


Figure 2: Warranty distribution architecture.

ranty terms are likely to be profitable. Stores see both read prepares and write prepares. If the object receives many read prepares and few or no write prepares, a state warranty on that object is likely to be profitable. A similar observation applies to computation warranties.

To determine whether to issue a warranty for an object, and its warranty term L in the case where a warranty is issued, the system plugs measurements of object usage into a simple system model. The system measures the rate W of writes to each object, and when there is no warranty issued on the object, it also measures the rate R of reads to the object. Both rates are estimated using an exponentially weighted moving average (EWMA) [28] of the intervals between reads and writes. We modify EWMA to exponentially decay historical read-prepare data during warranty periods, when read prepares cannot be observed. Empirically, this modification improves the accuracy of rate estimation. To lower the overhead of monitoring, unpopular objects are flagged and given lower-cost monitoring as long as they remain unpopular.

To ensure that the expected number of writes delayed by a warranty is bounded by a constant $k_1 < 1$ that controls the tradeoff between read and write transactions. The warranty term is set to k_1/W with a maximum warranty L_{max} used to bound write delays. Our goal is that warranties are profitable: they should remove load from the store, improving scalability. A warranty eliminates roughly RL read prepares over its term L, but adds the cost of issuing the warranty and some added cost for each write that occurs during the term. The savings of issuing a warranty is positive if each write to an object is observed by at least k_2 reads for some value k_2, giving us a condition $RL \geq k_2$ that must be satisfied in order to issue a warranty. The value for constant k_2 can be derived analytically using measurements of the various costs, or set empirically to optimize performance.

This way to set terms for state warranties also works for computation warranties, with the following interpretation: uses of a computation warranty are "reads" and updates to its dependencies are "writes".

The tension between write latency and read throughput can also be eased by using warranty refresh in addition to a maximum warranty term. The term L is computed as above, but warranties are issued to clients with a shorter term corresponding to the maximum acceptable update latency. The issuing store proactively refreshes each such warranty when it is about to expire, so the warranty stays valid at clients throughout its term.

3.9 Distributing warranties

Warranties can be used regardless of how they get to clients and can be shared among any number of clients. Therefore, a variety of mechanisms can be used to distribute warranties to clients.

One option for warranty distribution is to have clients directly query stores for warranties, but this makes the system less scalable by increasing load on stores. As shown in Figure 2, Stores will be less loaded if warranties are distributed via a content distribution network (CDN) that clients query to find warranties.

Going a step further, applications can *subscribe* to warranties that match a given pattern, as shown in Figure 2. Stores automatically *refresh* warranties with later expiration times before the old warranties expire, by pushing these extended warranties either directly to clients or into the CDN. Warranty refresh makes it feasible to satisfy client requests with shorter warranty terms, consequently reducing write latency.

This strategy for achieving high availability and high durability differs from that used in many current distributed storage systems, which use replication to achieve high availability, low latency, and durability. Those three goals are handled separately here. Distributing warranties through a CDN makes data objects highly available with low latency, without damaging consistency. Because the authoritative copies of objects are located at stores, a write to an object requires a round-trip to its store; the latency this introduces is ameliorated by the

Stores	Stores written	Unexpired?	Phases: Warranties	OCC
1+	0	Y	0	1
1+	0	N	1	1
1	1	Y/N	1	1
2+	1	Y	1	2
2+	1	N	2	2
2+	2+	Y	2	2
2+	2+	N	3	2

Table 1: Warranties require fewer phases than traditional OCC in some cases (highlighted).

support for relatively large transactions, in which communication with stores tends to happen at the end of transactions rather than throughout.

To achieve high durability, stores should be implemented using replication, so that each "store" mentioned in this paper is actually a set of replicas. Since wide-area replication of stores implementing strong consistency will have poor performance, we assume store replicas are connected with low latency.

4 Transactions and warranties

Warranties improve the performance of OCC by reducing the work needed during the prepare phase and by allowing phases to be eliminated entirely.

4.1 The warranty commit protocol

When a transaction completes, the client performs a modified two-phase commit, illustrated in Figure 1 for both read-only and read-write transactions. In the prepare phase, the client sends the write set of the transaction (if any), along with any warranties in the read set whose term has expired. If all warranties in the read set can be renewed, the transaction may commit. Since outstanding warranties may cause the updates to be delayed, the store responds with a *commit time* indicating when the commit may be applied successfully.

When the client receives a commit time from all stores, it checks to ensure the terms of the warranties it holds exceed the maximum commit time. If not, it attempts to renew these warranties beyond the commit time in an additional *extend* phase. If active warranties are obtained for all dependencies, the client sends the commit message, and the stores commit the updates at the specified time.

4.2 Avoiding protocol phases

While a two-phase commit is required in the general case, performance can be improved by eliminating or combining phases when possible. For read-only transactions, the commit phase is superfluous, and clients executing transactions that involve only one store can combine the prepare and commit phases into one round-trip. The optimizations to 2PC that warranties make possible are summarized in Table 1.

The read-only (rows 1–2) and single-store optimizations (row 3) are available with or without warranties. However, unexpired warranties enable eliminating additional phases, shown by the two rows highlighted in gray.

Row 1 shows that read-only transactions whose read set is covered by unexpired warranties may commit without communicating with stores—a zero-phase commit. This optimization matters because for read-biased workloads, most transactions will be read-only.

Row 4 shows that transactions that read from multiple stores but write to only one store may commit in a single phase if their read set is fully warrantied. This single-phase optimization pays off if objects are stored in such a way that writes are localized to a single store. For example, if a user's information is located on a single store, transactions that update only that information will be able to exploit this optimization.

While warranties usually help performance, they do not strictly reduce the number of phases required to commit a transaction. Transactions performing updates to popular data may have their commits delayed. Since the commit time may exceed the expiration time of warranties used in the transaction, the additional *extend* phase may be required to renew these warranties beyond the delayed commit time, as shown in the final row.

5 Computation warranties

A computation warranty is a guarantee until time t of the truth of a logical formula ϕ, where ϕ can mention computational results such as the results of method calls. We focus here on the special case of warranties generated by memoized function calls, where ϕ has the form $o.f(\vec{x}) =?$ for some object o on which method f is invoked using arguments $\vec{x}$, producing a value to be obtained from the warranty. Note that the value returned by f need not be a primitive value. In the general case, it may be a data structure built from both new objects constructed by the method call and preexisting objects.

Our goal is that warranties do not complicate programmer reasoning about correctness and consistency. Therefore, when f is a memoized method, a computation of the form $v = o.f(\vec{x})$ occurring in a committed transaction should behave identically whether or not a warranty is used to obtain its value. This principle has several implications for how computation warranties work. It means that only some computations make sense as computation warranties, and that updates must be prevented from invalidating active warranties.

5.1 Memoizable computations

To ensure that using a computation warranty is equivalent to evaluating it directly, we impose three restrictions.

First, computation warranties must be deterministic: given equivalent initial state, they must compute equivalent results. Therefore, computations using a source of nondeterminism, such as input devices or the system clock, do not generate computation warranties.

Second, we prevent memoization of any computation that has observable side effects. Side effects are considered to be observable only when they change the state of objects that existed before the beginning of the memoized computation.

Importantly, this definition of "observable" means that memoized computations are allowed to create and initialize new objects as long as they do not modify pre-existing ones. For example, the top-N example from Section 3.5 computes a new object representing a set of items, and it may be convenient to create the object by appending items sequentially to the new set. Warranties on this kind of side-effecting computation are permitted. Enforcing this definition of the absence of side effects is straightforward in a system that already logs which objects are read and written by transactions.

Third, a memoized function call reads from some set of objects, so updates to those objects may change its result, and may occur even during the same transaction that performed the function call. At commit time, the transaction's write set is intersected with the read set of each potential warranty. If the intersection is nonempty, the potential warranty is invalidated.

5.2 Defending computation warranties

Once a computation warranty is requested by a worker and issued by a store, the store must ensure that the value of the call stays unchanged until the warranty expires.

Revalidation A conservative way to defend warranties against updates would be to delay all transactions that update objects used by the warranty. This approach is clearly safe because of the determinism of the warranty computation, but it would prevent too many transactions from performing updates, hurting write availability. Instead, we attempt to *revalidate* affected warranties when each update arrives. The store reruns the warranty computation and checks whether the result is equivalent to the result stored in the warranty.

For primitive values and references to pre-existing objects (not created by the warranty computation), the result must be unchanged. Otherwise, two results are considered equivalent if they are semantically equal per the `equals()` method, which operates as in Java.

Warranty dependencies In general, a warranty computation uses and thus depends on other warranties, whether state warranties or general computation warranties. For example, if the method `top` is implemented recursively (see Figure 3), the warranty for a call to `top` depends on warranties for its recursive calls. The dependencies between warranties form a tree in which computation warranties higher in the tree depend on warranties lower down, and the leaves are state warranties.

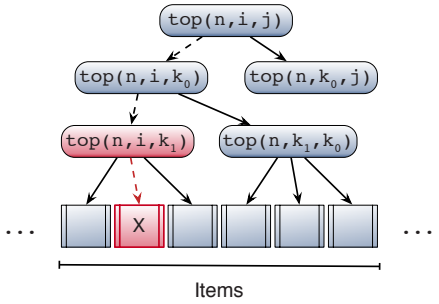


Figure 3: An update to X causes a semantic warranty to be invalidated, but the updated value for the re-evaluated method does not invalidate other warranties.

Any warranty that has not expired must be defended against updates that could invalidate it. Defense is easy when the term of a warranty is contained within (a subset of) the terms of all warranties it depends on, including state warranties on all direct references to objects, because the validity of the higher-level warranty is implied by the defense of the lower-level warranties.

In general, however, a warranty can have a longer term than some of its dependencies. Updates to those dependencies must be prevented if they invalidate the warranty, even if they are expired warranties. Conversely, it is possible to allow updates to warranty dependencies that do not invalidate the warranty. The implication is that it is often feasible to give higher-level warranties longer terms than one might expect given the rate of updates to their dependencies.

For example, consider the recursive call tree for the method `top(n, i, j)` shown in Figure 3. If the request to see the top n items among the entire set is very popular, we would like to issue relatively long computation warranties for that result. Fortunately, updates to items (shown at the leaves of the call tree) that change their ranking might invalidate some of the warranties in the tree, but most updates will affect only a small part of the tree. Assuming that lower levels of the tree have short warranties, most updates need not be delayed much.

5.3 Reusing computation warranty values

In the case where the warranty computation created new objects, it may be crucial for correctness of the computation that the objects returned by the warranty are distinct from any existing objects. This desired semantics is achieved when using a warranty computation result by making a copy of all objects newly created during the warranty computation. These objects are explicitly iden-

tified in the warranty.

Computation warranties are used whenever available to the client, to avoid performing the full computation. If the client is holding an expired warranty, or obtains an expired warranty from the CDN, it can use that expired warranty optimistically. At commit time, the expired warranty is revalidated during the prepare phase, exactly like a read prepare.

5.4 Creating computation warranties

Whenever code at a client makes a call to a memoized method, the client searches for a matching computation warranty. If the client is not already holding such warranty, it may search using a CDN, if available, or request the warranty directly from the appropriate store.

If the client cannot find an existing computation warranty, it performs the warranty computation itself. It starts a new transaction and executes the method call. As the call is evaluated, the transaction's log keeps track of all reads, writes, and object creations performed by the call. When the call is completed, the result is recorded and the log is checked to verify that the call does not violate any of the restrictions outlined above. If the warranty is still valid, the call, value, and transaction log are gathered to form a complete warranty proposal.

At commit time, if the warranty proposal has not already been invalidated by an update to its read set, the proposal is sent to the store. The store looks at the request and, using the same mechanism as for state warranties, sets a warranty term. For state warranties, terms are set individually for each object, but here the warranty identity is defined by the entire set of arguments to the memoized method. Finally, the computation warranty is issued to the requesting client and the store begins to defend the new warranty or warranties proposed by the client.

6 Implementation

To evaluate the warranty mechanism, we extended the Fabric secure distributed object system [38]. Fabric provides a high-level programming model that, like the Java Persistence API, presents persistent data to the programmer as language-level objects. Language-level objects may be both persistent and distributed. It implements linearizability using OCC.

Fabric also has many security-related features—notably, information flow control—designed to support secure distributed computation and also secure mobile code [5]. The dynamic security enforcement mechanisms of Fabric were not turned off for our evaluation, but they are not germane to this paper.

We extended the Fabric system and language to implement the mechanisms described in this paper. Our extended version of Fabric supports both state warranties and computation warranties. Computation warranties were supported by extending the Fabric language with memoized methods. Client (worker) nodes were extended to use warranties during computation and to evaluate and request computation warranties as needed. The Fabric dissemination layer, a CDN, was extended to distribute warranties and to support warranty subscriptions. Fabric workers and stores were extended to implement the new transaction commit protocols, and stores were extended to defend and revalidate warranties.

The previously released version of Fabric (0.2.1) contains roughly 44,000 lines of (non-blank, non-comment) code, including the Fabric compiler and the run-time systems for worker node, store nodes, and dissemination nodes, written in either Java or the Fabric intermediate language. In total, about 6,900 lines of code were added or modified across these various system components to implement warranties.

Fabric ships objects from stores to worker nodes in object groups rather than as individual objects. State warranties are implemented by attaching individual warranties to each object in the group.

Some features of the warranties design have not been implemented; most of these features are expected to improve performance further. The single-store optimization of the commit protocol has been implemented for base Fabric, but rows 3–5 of Table 1 have not been implemented for warranties. The warranty refresh mechanism is also not yet implemented.

To simplify the work needed to defend computation warranties, the current implementation only generates warranties for computations that involve objects from a single store. Also, our implementation does not use the dissemination layer to distribute computation warranties.

7 Evaluation

We evaluated warranties against existing OCC mechanisms, and other transactional mechanisms, primarily using three programs. First, we used the multiuser OO7 benchmark [13]. Second, we used versions of Cornell's deployed Course Management System [10] (CMS) to examine how warranties perform with real systems under real-world workloads. Both of these programs were ported to Fabric in prior work [38]. Third, we developed a new benchmark that simulates a component of a social network in which users have subscribers.

7.1 Multiuser OO7 benchmark

The OO7 benchmark was originally designed to model a range of applications typically run using object-oriented databases. The database consists of several modules, which are tree-based data structures in which each leaf of the tree contains a randomly connected graph of 20 objects. In our experiments we used the "SMALL" sized database. Each OO7 transaction performs 10 ran-

dom traversals on either the *shared* module or a *private* module specific to each client. When the traversal reaches a leaf of the tree, it performs either a read or a write action. These are relatively heavyweight transactions compared to many current benchmarks; each transaction reads about 460 persistent objects and modifies up to 200 of them. By comparison, if implemented in a straightforward way with a key-value store, each transaction would perform hundreds of get and put operations. Transactions in the commonly used TPC-C benchmark are also roughly an order of magnitude smaller [52], and in the YCSB benchmarks [54], smaller still.

Because OO7 transactions are relatively large, and because of the data's tree structure, OO7 stresses a database's ability to handle read and write contention. However, since updates only occur at the leaves of the tree, writes are uniformly distributed in the OO7 specification. To better model updates to popular objects, we modified traversals to make read operations at the leaves of the tree exhibit a power-law distribution with $\alpha = 0.7$ [11]. Writes to private objects are also made power-law distributed, but remain uniformly distributed for public objects.

7.2 Course Management System

The CS Course Management System [10] (CMS) is a 54k-line Java web application used by the Cornell computer science department to manage course assignments and grading. The production version of the application uses a conventional SQL database; when viewed through the JPA, the persistent data forms an object graph not dissimilar to that of OO7. We modified this application to run on Fabric. To evaluate computation warranties, we memoized a frequently used method that filters the list of courses on an overview page.

We obtained a trace from Cornell's production CMS server from three weeks in 2013, a period that encompassed multiple submission deadlines for several courses. To drive our performance evaluation, we took 10 common action types from the trace. Each transaction in the trace is a complete user request including generation of an HTML web page, so most request types access many objects. Using JMeter [30] as a workload generator, we sampled the traces, transforming query parameters as necessary to map to objects in our test database with a custom JMeter plugin.

7.3 Top-subscribers benchmark

The third benchmark program simulates a relatively expensive analytics component of a social network in which users have subscribers. The analytics component computes the set of 5 users with the largest number of subscribers, using the memoized top-N function described in Section 3.5. The number of subscribers per user is again determined by a power-law distribution with $\alpha = 0.7$. The workload consists of a mix of two operations: 98% compute the list of top subscribers, corresponding to viewing the home page of the service; 2% are updates that randomly either subscribe or unsubscribe some randomly chosen user. This example explores the effectiveness of computation warranties for caching expensive computed results.

7.4 Comparing with Hibernate/HSQLDB

To provide a credible baseline for performance comparisons, we also ported our implementation of CMS to the Java Persistence API (JPA) [12]. We ran these implementations with the widely used Hibernate implementation of JPA 2, running on top of HyperSQL (HSQLDB), a popular in-memory database in `READ COMMITTED` mode. For brevity, we refer to Hibernate/HSQLDB as *JPA*. For JPA, we present results only for a single database instance. Even in this single-store setting, and even with Hibernate running in its optimistic locking mode, which does not enforce serializability, Fabric significantly outperforms JPA in all of our experiments. (Note that JPA in optimistic locking mode is in turn known to outperform JPA with pessimistic locking, on read-biased workloads [49, 17]). This performance comparison aims to show that Fabric is a good baseline for evaluating the performance of transactional workloads: its performance is competitive with other storage frameworks offering a transactional language-level abstraction.

7.5 Experimental setup

Our experiments use a semi-open system model. An open system model is usually considered more realistic [48] and a more appropriate way to evaluate system scalability. Worker nodes execute transactions at exponentially distributed intervals at a specified *average request rate*. Consequently, each worker is usually running many transactions in parallel. Overall system throughput is the total of throughput from all workers. To find the maximum throughput, we increase the average request rate until the target throughput cannot be achieved.

The experiments are run on a Eucalyptus cluster. Each store runs on a virtual machine with a dual core processor and 8 GB of memory. Worker machines are virtual machines with 4 cores and 16 GB of memory. The physical processors are 2.9 GHz Intel Xeon E5-2690 processors.

The parameters k_1 and k_2 (Section 3.8) are set to 0.5 and 2.0, respectively; the maximum warranty term was 10 s. Performance is not very sensitive to k_1 and k_2.

7.6 Results

We evaluated scalability using the OO7 benchmark with different numbers of stores. A "shared store" was reserved for the assembly hierarchies of all modules. The component parts of the modules were distributed evenly across the remaining stores. Only shared composite parts

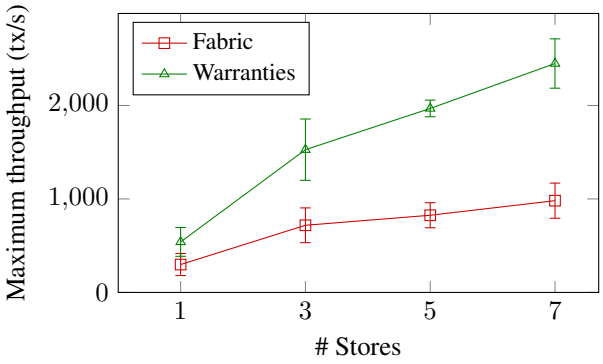


Figure 4: OO7 maximum throughput on a 2%-write workload as the number of stores increases. Warranties allow throughput to scale up with more stores.

were placed on the shared store. Results presented are the average of three runs.

Figure 4 shows maximum throughput in total transactions committed per second by 36 workers, as the number of stores increases. Error bars show the standard deviation of the measurements. As expected, adding stores has little effect on maximum throughput in base Fabric because the shared store is a bottleneck. Warranties greatly reduce load on the shared store allowing us to add roughly 400 tx/s per additional store. Note that the plot only counts committed transactions; the percentage of aborted transactions for Fabric at maximum throughput ranges from 2% to 6% as the number of stores increases from 3 to 7; with warranties, from 4% up to 15%.

Table 2 reports on the performance of the CMS application in various configurations. The first three rows of Table 2 show that Fabric, without or without warranties, delivers more than an order of magnitude performance improvement over JPA. Although the JPA implementation enforces weaker consistency, Fabric's more precise

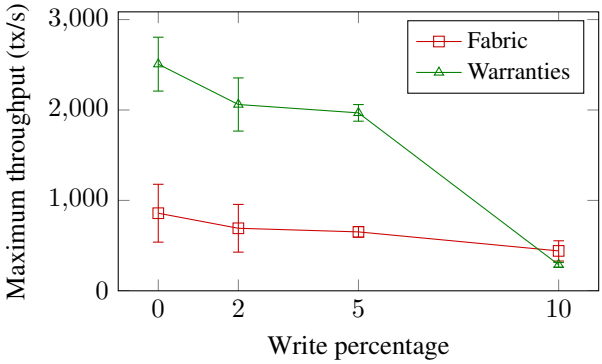


Figure 5: Effect of write percentage on OO7 maximum throughput on 3 stores with 24 workers.

System	Stores	Tput (tx/s)	Latency (ms)
JPA	1	72 ± 12	211 ± 44
Fabric	1	3032 ± 144	143 ± 120
Warranties	1	4142 ± 112	27 ± 27
Comp. Warranties	1	4088 ± 189	114 ± 30
Fabric	3	4090 ± 454	311 ± 175
Warranties	3	5886 ± 124	35 ± 4

Table 2: CMS throughput and latency on various systems. Both are averaged over 10 s at max throughput.

object invalidation helps performance as contention increases. Warranties help improve performance further, even in a single-store configuration.

To evaluate how the system scales for a more realistic workload, we also ran CMS with 3 stores using Fabric and Warranties. Two stores each held data for multiple courses, while the third store contained metadata. As Table 2 shows, Warranties scale better than Fabric with the additional stores.

Increases in throughput would be less compelling if they came at the cost of high latency. Table 2 also reports the latency measured with the CMS workload on the various systems. Fabric has similar latency with or without warranties. Because CMS was not designed with computation warranties in mind, the functions we designated to be memoized turn out not to have a significant impact on performance. They are relatively cheap to evaluate on cached objects, and the bookkeeping for computation warranties adds no noticeable overhead.

Figure 5 shows how the performance of warranties is affected by the fraction of update transactions. Four different workload mixes were measured, each having a 94:6 shared-to-private traversal ratio and a 1:10 shared-to-private write ratio. When more than 10% of the transactions are updates, the cost of maintaining and issuing warranties in the current implementation is too high to obtain a performance improvement. The latencies at some of these throughputs are higher than Fabric's, but still relatively low. At 2% and 5% writes, the latency of warranties is about 400 ms higher than Fabric's but nearly the same as Fabric's at 0% and 10% writes.

Warranties can result in delaying transactions that are attempting to write to an object that has a warranty. We call this *write delay*. For all of the runs depicted in Figure 5, the median write delay is 0 ms. However, some fraction of transactions are forced to wait until one or more warranties expire. The more read-biased the transaction, the more frequently this happens. In the 2%-write workload, 70% of read-write transactions see no write delay. In the 10%-write workload, 82% see no write delay. Among those that encounter write delay, the delay is roughly uniformly distributed from 0 up to the max warranty length.

	Tput	Median Latency	95th pct Write Delay
Fabric	17 ± 5	568 ± 500	N/A
Warranties	26 ± 7	1239 ± 644	623 ± 387
Comp. Warranties	343 ± 14	12 ± 3	16 ± 5

Table 3: Top-N benchmark: maximum throughput (tx/s), latency (ms), and 95th percentile write delay (ms).

7.7 Computation warranties

To further evaluate the impact of computation warranties, we ran the top-N benchmark with Fabric, state warranties, and with computation warranties. Because the performance of the recursive top-N strategy on Fabric and on state warranties was very poor, we used an alternate implementation that performed better on those configurations. Table 3 shows the average across three runs of the maximum throughput and the corresponding latency achieved in the system without any operations failing to commit during a 15 minute period. Computation warranties improve throughput by more than an order of magnitude. Since the computation warranty is on the value of the top 5 accounts rather than on each individual value used in computing the result, writes are not delayed as heavily as they are when using only state warranties.

8 Related work

Many mechanisms for enforcing concurrency control have been proposed in the literature: locks, timestamps, versions, logs, leases, and many others [33, 22, 34, 46, 7, 21]. Broadly speaking, these can be divided into optimistic and pessimistic mechanisms. The monograph by Bernstein, Hadzilacos, and Goodman provides a broad overview from the perspective of databases [8]. Warranties are an optimistic technique, allowing clients to concurrently operate on shared data.

Haerder [24] divides mechanisms for validating optimistic transactions into "forward" and "backward" techniques. Backward validation is a better choice for the distributed setting [3], so Fabric uses backward validation: transactions are aborted in the prepare phase if any object in the read set has been modified.

Traditionally, most systems adopted *serializability* or *linearizability* as the gold standard of strong consistency [42, 8, 25]. But many recent systems have sacrificed serializability in pursuit of scalable performance. Vogels [53] discusses this trend and surveys various formal notions of *eventual consistency*. Much prior work aims to provide a consistency guarantee that is weaker than serializability; for example, causal consistency (e.g., [44, 39]) and *probabilistically-bounded staleness* [6]. Because this paper is about strong consistency, we do not discuss this prior work in depth.

Leveraging application-level information to guide implementations of transactions was proposed by Lamport [33] and explored in Garcia-Molina's work on *semantic types* [19], as well as recent work on *transactional boosting* [26] and *coarse-grained transactions* [31]. Unlike warranties, these systems use mechanisms based on commuting operations. A related approach is *red–blue consistency* [36], in which red operations must be performed in the same order at each node but blue operations may be reordered.

Like warranties, Sinfonia [4] aims to reduce client–server round trips without hurting consistency. It does this through *mini-transactions*, in which a more general computation is piggybacked onto the prepare phase. This optimization is orthogonal to warranties.

Warranties borrow from leases [21] the idea of using expiring guarantees, though important differences are discussed in Section 3.2. In fact, the idea of expiring state guarantees occurs prior to leases in Lampson's global directory service [35]. We are not aware of any existing system that combines optimistic transactions with leases or lease-like mechanisms, against which we could meaningfully compare performance.

A generalization of leases, *promises* [23, 29] is a middleware layer that allows clients to specify resource requirements via logical formulas. A resource manager considers constraints across many clients and issues time-limited guarantees about resource availability. Scalability of promises does not seem to have been evaluated.

The tracking of dependencies between computation warranties, and the incremental updates of those warranties while avoiding unnecessary invalidation, is close to the update propagation technique used in self-adjusting computation [1], realized in a distributed setting. Incremental update of computed results has also been done in the setting of MapReduce [9].

The TxCache system [45] provides a simple abstraction for caching and reusing results of functions operating over persistent data from a single storage node in a distributed system. As with the Fabric implementation of computation warranties, functions may be marked for memoization. TxCache does not ensure that memoized calls have no side effects, so memoized calls may not behave like real calls. Memoized results are not shared across clients. Compared to Fabric, TxCache provides a weaker consistency guarantee, transactional consistency, requiring that all transactions operate over data that is consistent with a prior snapshot of the system.

Escrow transactions [41] have some similarities to computation warranties. They generalize transactions by allowing commit when a predicate over state is satisfied. Certain updates (incrementing and decrementing values) may take place even when other transactions may

be updating the same values, as long as the predicate still holds. Compared to computation warranties, escrow transactions support very limited predicates over state, and their goal is different: to permit updates rather than to allow the result of a computation to be widely reused.

9 Conclusions

Strong consistency tends to be associated with the very real performance problems of pessimistic locking. While optimistic concurrency control mechanisms deliver higher performance for typical workloads, read prepares on popular objects are still a performance bottleneck. Warranties generalize OCC in a way that reduces the read-prepare bottleneck. Warranties address this bottleneck by allowing stores to distribute warranties on popular objects, effectively replicating their state throughout the system. Warranties can delay update transactions, but our results suggest that the delay is acceptable. Effectively, warranties generalize OCC in a way that adjusts the balance of power between readers and writers, substantially increasing overall performance. Computation warranties improve performance further by supporting `memcached`-like reuse of computations—but without losing strong consistency.

Acknowledgments

We would especially like to thank Robert Soulé for help setting up experiments and Nate Foster for good suggestions. Chin Isradisaikul also had good ideas for presentation, and we thank our shepherd Yuan Yu. We thank Hakim Weatherspoon for the use of Fractus cloud infrastructure provided by an AFOSR DURIP award, grant FA2386-12-1-3008.

This project was funded partly by the Office of Naval Research (grant N00014-13-1-0089), by MURI grant FA9550-12-1-0400, by a grant from the National Science Foundation (CCF-09644909), and by an NDSEG Fellowship. This paper does not necessarily reflect the views of any of these sponsors.

References

[1] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proc. 35th ACM Symposium on Principles of Programming Languages (POPL)*, pages 309–322, 2008.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, December 2002.

[3] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 23–34, San Jose, CA, May 1995.

[4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proc. 21st ACM Symp. on Operating System Principles (SOSP)*, pages 159–174, October 2007.

[5] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *Proc. IEEE Symp. on Security and Privacy*, pages 191–205, May 2012.

[6] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 5(8):776–787, April 2012.

[7] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM CSUR*, 13(2):185–221, 1981.

[8] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987. Available at `http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx`.

[9] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquini. Incoop: MapReduce for incremental computations. In *ACM Symp. Cloud Computing*, October 2011.

[10] Chavdar Botev et al. Supporting workflow in a course management system. In *Proc. 36th ACM Technical Symposium on Computer Science Education (SIGCSE)*, pages 262–266, February 2005.

[11] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *INFOCOM*, 1999.

[12] Heiko Böck. *Java Persistence API*. Springer, 2011.

[13] Michael Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proc. 9th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 414–426, 1994.

[14] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st SOSP*, 2007.

[15] EclipseLink. `http://www.eclipse.org/eclipselink`.

[16] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Comm. of the ACM*, 19(11):624–633, November 1976. Also published as IBM RJ1487, December, 1974.

[17] Gavin King et al. Hibernate developer guide. Hibernate Community Documentation. `http://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html`.

[18] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, August 2004.

[19] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS*, 8(2):186–213, June 1983.

[20] Cary G. Gray. *Performance and Fault-Tolerance in a Cache for Distributed File Service*. PhD thesis, Stanford University, December 1990.

[21] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symp. on Operating System Principles (SOSP)*, pages 202–210, 1989.

[22] Jim N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems, an Advanced Course*, volume 60 of *LNCS*, pages 393–481. Springer-Verlag, 1978.

[23] Paul Greenfield, Alan Fekete, Julian Jang, Dean Kuo, and Surya Nepal. Isolation support for service-based applications: A position paper. In *Proc. 3rd CIDR*, pages 314–323, 2007.

[24] T. Haerder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, June 1984.

[25] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. Technical Report CMU-CS-88-120, Carnegie Mellon University, Pittsburgh, Pa., 1988.

[26] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proc. 13th PPoPP*, pages 207–216, February 2008.

[27] Hibernate. `http://www.hibernate.org`.

[28] J. Stuart Hunter. The exponentially weighted moving average. *Journal of Quality Technology*, 18:203–210, 1986.

[29] J. Jang, A. Fekete, and P. Greenfield. Delivering promises for web services applications. In *Proc. 5th ICWS*, pages 599–606, July 2007.

[30] JMeter. `http://jmeter.apache.org`.

[31] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. Coarse-grained transactions. In *Proc. 37th POPL*, pages 19–30, January 2010.

[32] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[33] L. Lamport. Towards a theory of correctness for multi-user data base systems. Report CA-7610-0712, Mass. Computer Associates, Wakefield, MA, October 1976.

[34] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[35] Butler W. Lampson. Designing a global name service. In *Proc. 5th PODC*, pages 1–10, August 1986.

[36] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. 10th OSDI*, October 2012.

[37] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 318–329, June 1996.

[38] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proc. 22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, 2009.

[39] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proc. 23rd ACM Symp. on Operating System Principles (SOSP)*, 2011.

[40] D. L. Mills. Network time protocol (version 3) specification, implementation and analysis. Network Working Report RFC 1305, March 1992.

[41] P. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems (TODS)*, 11(4):405–430, December 1986.

[42] Christos H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 26(4):631–653, October 1979.

[43] Peter Peinl and Andreas Reuter. Empirical comparison of database concurrency control schemes. In *Proc. 9th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 97–108, 1983.

[44] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, St. Malo, France, October 1997.

[45] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *Proc. 9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2010.

[46] David P. Reed. Naming and synchronization in a decentralized computer system. Technical Report MIT-LCS-TR-205, Massachusetts Institute of Technology, 1978.

[47] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM CSUR*, 37(1):42–81, March 2005.

[48] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *Proc. 3rd Conf. on Networked Systems Design & Implementation (NSDI)*, pages 18–31, Berkeley, CA, USA, 2006. USENIX Association.

[49] ObjectDB Software. ObjectDB 2.3 developer's guide. `http://www.objectdb.com/java/jpa/persistence/lock`.

[50] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Mike J. Spreitzer. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. 15th ACM Symp. on Operating System Principles (SOSP)*, pages 172–183, December 1995.

[51] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: a scalable distributed file system. In *Proc. 16th ACM Symp. on Operating System Principles (SOSP)*, pages 224–237, 1997.

[52] TPC-C. `http://www.tpc.org/tpcc/`.

[53] Werner Vogels. Eventually consistent. *CACM*, 52(1):40–44, January 2009.

[54] Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

Tierless Programming and Reasoning for Software-Defined Networks

Tim Nelson Andrew D. Ferguson Michael J. G. Scheer Shriram Krishnamurthi
Brown University

Abstract

We present Flowlog, a *tierless* language for programming SDN controllers. In contrast to languages with different abstractions for each program tier—the control-plane, data-plane, and controller state—Flowlog provides a unified abstraction for all three tiers. Flowlog is reminiscent of both SQL and rule-based languages such as Cisco IOS and JunOS; unlike these network configuration languages, Flowlog supports programming with mutable state. We intentionally limit Flowlog's expressivity to enable built-in verification and proactive compilation despite the integration of controller state. To compensate for its limited expressive power, Flowlog enables the reuse of external libraries through callouts.

Flowlog proactively compiles essentially all forwarding behavior to switch tables. For rules that maintain controller state or generate fresh packets, the compiler instructs switches to send the minimum amount of necessary traffic to the controller. Given that Flowlog programs can be stateful, this process is non-trivial. We have successfully used Flowlog to implement real network applications. We also compile Flowlog programs to Alloy, a popular verification tool. With this we have verified several properties, including program-correctness properties that are topology-independent, and have found bugs in our own programs.

1 Introduction

In a software-defined network (SDN), switches delegate their control-plane functionality to logically centralized, external controller applications. This split provides several advantages including a global view of network topology, and the use of general-purpose programming languages for implementing network polices. These general-purpose languages require an interface to the switch hardware, such as OpenFlow [20], which also provides a basic abstraction of the switch's flow tables.

To best use this interface, recent research has produced domain-specific languages like NetCore [21] that can be proactively compiled to flow tables. While this exclusive focus on flow tables simplifies compilation, it hurts expressivity. NetCore, for instance, can describe a forwarding policy, but lacks the ability to reference (let alone change) state on the controller.

Instead, the programmer must write a multi-tier program: a wrapper in a general-purpose language that maintains control-plane state and dynamically creates new, stateless data-plane policies to describe current forwarding goals. The data-plane policies must also specify which packets the switches should send to the controller, but—due to the multi-tier, multi-language nature of these programs—there is no structured connection between policies that describe packets the controller receives, and the arbitrary code in a callback function that consumes them. This gap can lead to bugs in how switches update the controller, resulting in incorrect controller state or network policies. Moreover, if packets are delivered to the controller needlessly, performance suffers.

To better support controller programming, we have created Flowlog, a *tierless* network programming language. As in web-programming, where a program contains multiple tiers such as client-side JavaScript, a server-side program, and a database, an SDN system also has multiple tiers: flow rules on switches, a controller program, and a data-store for controller state. By incorporating all of these tiers, a single, unified Flowlog program describes both control- and data-plane behavior.

Flowlog also provides built-in support for program verification. Because controller programs pose a single point of failure for the entire network, verification tools are invaluable to SDN developers. Prior SDN controller analysis work has often focused on the switch rules themselves, either statically [2, 12, 19, 26] or dynamically, as each update is sent to the switches [25]. However, most SDN analyses focus on trace properties: statements about the end-to-end behavior of packets in the network (e.g.,

a lack of routing loops). While these analyses are useful, they generally must be performed with respect to a given topology, which limits their flexibility, reusability, and scalability. In contrast, our reasoning focuses on properties independent of network topology.

We have limited Flowlog's expressive power to support both tierlessness and verification, while retaining enough expressivity to be useful for real-world programming. A limited language poses obvious problems for developers, both in expressing their needs and in reusing existing code. Flowlog therefore provides interfaces and abstractions for interacting with external programs. Programmers are free to invoke existing, full-featured libraries as needed, depending on their analysis goals. This is in contrast to most policy languages: in Flowlog, the restricted language itself forms the primary program, calling the external code rather than being called by it. This approach has been successful in SQL, where database queries are in the "limited" language and user-defined functions are in "full" languages. Our work explores such a strategy for network programming. Our contributions are:

1. We present the tierless Flowlog language (Section 3) and demonstrate its expressive power on real-world examples (Section 2). The language includes SQL-like relational state. It also provides abstractions for interaction with external code, via either asynchronous *events* or synchronous *remote tables*. Section 6 describes its implementation.

2. We show how, in spite of Flowlog's tierless merging of data- and control-plane behavior, programs can be proactively compiled to flow table rules (Section 4). The compilation process extends beyond mere packet forwarding; it also filters packets that may trigger state updates or cause event output and notifies the controller only as necessary.

3. We automatically compile Flowlog programs to the Alloy [10] verifier (Section 5). We focus on analyses that are independent of network topology, which are especially helpful when the topology is virtual, in flux, or unknown. We show that this process is aided by Flowlog's tierlessness as well as its limited expressiveness. We are able to verify properties in less than a second with minimal developer input. This verification support has helped us find surprising errors in our own Flowlog programs.

2 Flowlog by Example

We introduce Flowlog with illustrative examples. These demonstrate Flowlog's tierless nature, along with its expressive power, concision, and ability to support real-world development needs. We have also designed Flowlog to be amenable to both sound (i.e., no false positives) and complete (i.e., no bugs are missed) verification. To meet these goals, Flowlog bans loops and recursion, and has a logical semantics that we leverage for sound and, in many cases, complete verification (Section 5). Moreover, no recursion means that Flowlog programs always terminate on each incoming event.

Stolen Laptop Detector Let us write an application to help campus police track down stolen laptops. It must accept signals from campus police that report a laptop stolen or recovered, and if a stolen laptop is seen sending packets, the program must alert the police, saying which switch the laptop is connected to and when the packet was seen. (For brevity, we do not demonstrate rate limiting of alerts or restriction of alerts to edge-router traffic. Both of these tasks can be accomplished in Flowlog.)

Without a tierless programming language, expressing this program would require many pieces, possibly using multiple languages: a database or data structures to manage controller state; a remote-procedure call (RPC) library, or similar solution, for handling events; and policy-generation code that produces fresh rules on the switches that forward traffic and check for stolen laptops on the network. In Flowlog, all of these components share the same abstraction. Suppose we have a table `stolen` that tracks the MAC addresses of all currently stolen laptops. Then, the heart of the program is just the following rules:

```
1  ON stolen_report(sto):
2    INSERT (sto.mac) INTO stolen;
3  ON stolen_cancel(rec):
4    DELETE (rec.mac) FROM stolen;
5  ON packet_in(p):
6    DO notify_police(sto) WHERE
7      sto.mac = pkt.dlSrc AND
8      sto.time = time AND
9      sto.swid = pkt.locSw AND
10     stolen(pkt.dlSrc) AND
11     get_time(time);
12   DO forward(new) WHERE
13     new.locPt != p.locPt;
```

The program describes several kinds behavior that appear disparate. It: (a) adds addresses to a table when laptops are reported stolen (lines 1–2); (b) removes addresses when laptops are recovered (lines 3–4); (c) notifies police when a packet appears from a stolen laptop (lines 5–11); and (d) floods packets (lines 12–13); this trivial example of forwarding introduces syntax which we will use later. With Flowlog's tierless abstraction, we express all of this in four concise rules.

Every program has similar rules that describe how to handle each packet; these rules are written in a syntax reminiscent of SQL, and the semantics is correspondingly relational. In addition, most programs will have state that must be updated in reaction to packets and other stimuli.

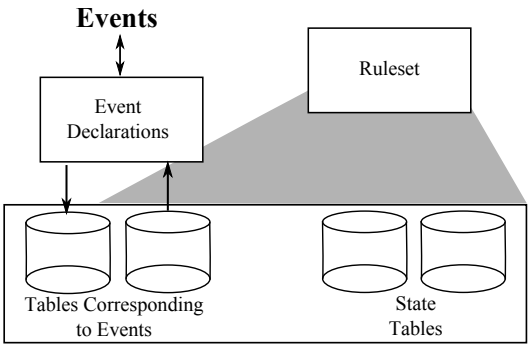


Figure 1: Flowlog system diagram. Events are represented by tuples in event tables. Rules interact with the entire database.

These stimuli are external events, whose implementations can be arbitrary, and whose interfaces are explicitly declared or built in. We now describe the program declarations that support these rules. (Figure 1 shows the different components of Flowlog.)

We have already seen `stolen`, the internal controller state. We expose the current time with an external table `get_time` (which always contains just one entry):

```
1 TABLE stolen(switchid);
2 REMOTE TABLE get_time(int);
```

External tables are managed by arbitrary external programs; in our examples, we used OCaml programs running on the controller machine. Each table declares its type, which is used for error-checking and optimization.

Next, we define the shape of incoming and outgoing events. We must handle two kinds of notifications from the police, and send them one:

```
1 EVENT stolen_report {mac: macaddr};
2 EVENT stolen_cancel {mac: macaddr};
3 EVENT stolen_found {mac: macaddr, swid:
      switchid, time: int};
```

Flowlog processes incoming events one at a time. Incoming events are placed in an identically named table, causing dependent rules to be re-evaluated. Outgoing events are also represented as tables, and when the ruleset adds a tuple to such a table, Flowlog sends an event with the tuple's content. These tables are therefore similar to named pipes in Unix. We require one such named pipe to send `stolen_found` events to the police server:

```
1 OUTGOING notify_police(stolen_found)
2   THEN SEND TO 127.0.0.1:5050;
```

If a Flowlog program inserts tuples into the `notify_police` table, these tuples will be transformed into `stolen_found` events and sent to a process listening on `127.0.0.1:5050`. Incoming `stolen_report` and `stolen_cancel` events are automatically inserted into their identically named tables. For instance, the `stolen_report` table will contain arriving `stolen_report` events, and the `packet_in` table will hold incoming packets. Notice we did not declare an outgoing `forward` table. This is because Flowlog automatically creates outgoing tables for common packet-handling behavior, as detailed in Table 1.

The remote table `get_time` is populated by querying `127.0.0.1:9091`, and should be refreshed every second:

```
1 REMOTE TABLE get_time
2   FROM time AT 127.0.0.1:9091
3   TIMEOUT 1 seconds;
```

The `TIMEOUT` field (line 3) is vital for performance and correct proactive compilation. A numeric timeout gives a window during which the results can be cached. Flowlog also provides a `NEVER` keyword, meaning the external callout has no side-effects, and thus its results can be cached indefinitely. A default, empty timeout requires updating the remote table every time the program is evaluated.

The reader may wonder whether this program can be compiled to stateless rules and installed in switches. After all, it contains a rule only the controller can handle, since it involves notifying campus police. But this rule only fires when `stolen(p.dlSrc)` is true, so Flowlog's proactive compiler instructs switches to send packets to the controller only if their source-MAC field has been registered as stolen. Furthermore, Flowlog automatically updates the switches every time a new theft is reported; no code to that effect is needed.

Network Information Base Next, we show how Flowlog can be used to compute a network information base, or NIB [15]. We begin with topology discovery, using Flowlog's ability to process timer notifications and emit new packets to run an LLDP-like protocol. First, we react to switch registration to obtain identifiers for every port (omitting table declarations):

```
1 ON switch_port_in(swpt):
2   INSERT (swpt.sw, swpt.pt)
3     INTO switch_has_port;
```

Thus, the `switch_has_port` table will hold every switch-port pair that registers. Next, we set up a 10-second event loop (we exclude the timer declaration):

```
1 ON startup(empty_event):
2   DO start_timer(10, "tNIB");
3 ON timer_expired(timer)
4   WHERE timer.id = "tNIB":
5   DO start_timer(10, "tNIB");
```

The first rule uses Flowlog's built-in `startup` event to start the loop, and the second rule continues it. The constraint `timer.id = "tNIB"` accounts for situations where multiple timers may be in use. The same timer also causes known switches to issue probe packets from each port:

INCOMING Table	Corresponding EVENT (with fields)	Description
packet_in	packet{locSw, locPt, dlSrc, dlDst, dlTyp, nwSrc, nwDst, nwProtocol}	packet arrival
switch_port	switch_port{sw, pt}	switch registration
switch_down	switch_down{sw}	switch down
E	E	Any incoming event E

OUTGOING Table	Corresponding EVENT	Description
emit	packet	Emit a new packet
forward	packet	Forward with modifications (triggered by packets only)

Table 1: Built-in **INCOMING** and **OUTGOING** tables. The `locSw` and `locPt` fields denote the packet's (switch and port) location.

```
1  ON timer_expired(timer)
2    WHERE timer.id = "tNIB":
3    DO emit(new) WHERE
4       switch_has_port(new.locSw,new.locPt)
5       AND new.dlTyp = 0x1001
6       AND new.dlSrc = new.locSw
7       AND new.dlDst = new.locPt;
```

We use `dlTyp = 0x1001` (line 5) to mark probe packets. The current switch and port IDs are smuggled in the MAC address fields of the probe (lines 6-7). (We omit the rule that initiates the same probe emission process on switch registration.)

We obtain knowledge of the switch topology from probe reception:

```
1  ON packet_in(p) WHERE p.dlTyp = 0x1001:
2    INSERT (p.dlSrc, p.dlDst,
3            p.locSw, p.locPt) INTO ucST;
```

The table name `ucST` denotes *under construction* switch topology; at any point, it contains a topology based on the probes seen so far this cycle. We empty `ucST` on every cycle, and maintain a `switchTopology` table that stores the value of the last complete `ucST` before it is deleted. Flowlog programs routinely use this strategy of building up a helper table over an execution cycle, separating in-progress results from the last complete result set:

```
1  ON timer_expired(timer)
2  WHERE timer.id = "tNIB":
3    DELETE (sw1,pt1,sw2,pt2) FROM ucST WHERE
4      ucST(sw1, pt1, sw2, pt2);
5    DELETE (sw1,pt1,sw2,pt2)
6      FROM switchTopology WHERE
7      switchTopology(sw1, pt1, sw2, pt2);
8    INSERT (sw1,pt1,sw2,pt2)
9      INTO switchTopology WHERE
10     ucST(sw1, pt1, sw2, pt2);
```

Though Flowlog does not allow recursion (Section 3), we can use a similar approach to compute reachability on the network. This program computes a fresh reachability table (`ucTC`, for under-construction transitive closure) as each probe arrives:

```
1  ON packet_in(p) WHERE p.dlTyp = 0x1001
2    AND dstSw = p.locSw AND srcSw = p.dlSrc:
3    INSERT (srcSw, dstSw) INTO ucTC;
4    INSERT (sw, dstSw) INTO ucTC
5      WHERE ucTC(sw, srcSw);
6    INSERT (srcSw, sw) INTO ucTC
7      WHERE ucTC(dstSw, sw);
8    INSERT (sw1, sw2) INTO ucTC
9      WHERE ucTC(sw1, srcSw)
10     AND   ucTC(dstSw, sw2);
```

The program works as follows: for every probe packet received, it concludes that its source switch and its arrival switch are connected (line 3). It also extends existing reachability in both directions (lines 4–7). Finally, it must account for packets connecting two cliques of reachability (lines 8–10).

It is instructive to compare this algorithm to the standard two-rule Datalog program for transitive-closure [1, p. 274]. The extra rules arise because we are *not* computing transitive-closure in the usual sense. Here, we do not have the luxury of assuming that we possess the entire connection table in advance; we must compute reachability on-the-fly as new information arrives. This difference leads to added complexity. In fact, an initial version of this program lacked the final rule, and so failed to faithfully compute reachability in some cases; we found this bug using our verification tool (Section 5).

Once we have network reachability, we can compute a spanning tree for the network:

```
1  ON packet_in(p) WHERE p.dlTyp = 0x1001
2    AND dstSw = p.locSw AND dstPt = p.locPt
3    AND srcSw = p.dlSrc AND srcPt = p.dlDst:
4    INSERT (srcSw, srcPt) INTO ucTree
5      WHERE NOT ucTC(srcSw, dstSw)
6      AND   NOT ucTC(dstSw, srcSw);
7    INSERT (dstSw, dstPt) INTO ucTree
8      WHERE NOT ucTC(srcSw, dstSw)
9      AND   NOT ucTC(dstSw, srcSw);
```

Again, we see unsurprising parallels to distributed protocols. Ordinary spanning tree algorithms have the luxury of working with the entire graph at once, and thus are often able to build a connected tree at every step. We do not have that luxury here: probes may arrive in any order, and we must build a forest of trees that, should links go down, may not even be connected. We also must add a pair of rules, one for each direction of the branch.

```
block ::= ON <id> ( <id> ) [WHERE rformula]
          : rules
rules ::= rule | rule rules
rule  ::= do_act | ins_act | del_act
do_act  ::= DO <id> ( termlist )
            [WHERE rformula] ;
ins_act ::= INSERT ( termlist )
            INTO <id> [WHERE rformula] ;
del_act ::= DELETE ( termlist )
            FROM <id> [WHERE rformula] ;
term ::= <num> | <string> | <id> | <id>.<id> | ANY
termlist ::= term | term , termlist
rformula ::= <id> ( termlist ) | term = term |
             NOT rformula | rformula AND rformula |
             rformula OR rformula | ( rformula )
```

Figure 2: Syntax of Flowlog rulesets. A program is a succession of **ON** blocks. Optional arguments are in square brackets. Capitalized tokens and punctuation are reserved constants.

Of course, this spanning tree is not necessarily the best possible one; we only compute the first such tree to be exposed by probe packets. Better tree-generation algorithms can be written or accessed via external code. Numerous other data, such as the location of connected hosts, can also be gathered, but are omitted for space.

Given a spanning tree for the network—whether it is computed in Flowlog or obtained from external code—we can construct a "smart" learning switch application in Flowlog that does not suffer from the usual issues with cyclic topologies.

Other Examples We have implemented additional applications in Flowlog, which are available in our repository.[1] These examples include an ARP proxy, a stateful firewall, and an application (which we use in-house) to facilitate access to Apple TV devices across subnets.

3 The Flowlog Language

As seen in Section 2, every Flowlog program contains a declarative *ruleset* that governs controller behavior and a set of *declarations* for the program's state tables and incoming/outgoing interface. Figure 2 gives the concrete syntax of Flowlog rulesets.

Declarations A program declares **EVENT**s, state **TABLE**s, and interfaces for **INCOMING** and **OUTGOING** tables. Most **INCOMING** and some **OUTGOING** declarations are made automatically when events are declared. Declaring a table as **REMOTE** informs Flowlog that the table represents a callout to external code, and that the ruleset will not maintain that table's state. Every event declaration is equipped with a set of field names for that event type.

[1] `http://cs.brown.edu/research/plt/dl/flowlog/`

Every internal table and interface table is equipped with a type, given as a vector of type names (e.g., "switchid")—one for each column in the table. **REMOTE TABLE** and **OUTGOING** declarations must also be provided with additional information, as we saw in Section 2.

Rulesets A ruleset contains a set of **ON** blocks of rules. While we allow multiple rules within the same **ON** block for conciseness, without loss of generality we will pretend that every rule has its own **ON** block. Each rule indicates an action to be taken when the **ON**-specified trigger is seen: either to **INSERT** or **DELETE** a tuple from the controller state, or to **DO** an action such as forwarding a packet. Finally, rules and triggers have an optional **WHERE** clause, which adds additional constraints; these are always an expression involving only the tables declared in **TABLE**s, never those declared as **INCOMING** or **OUTGOING**. These rules determine a function that maps controller state and incoming events to a new state and set of outgoing events.

Each rule defines a logical implication stating that, should its body be satisfied, its action should be as well. Figure 3 shows how we arrive at this *rule clause* for each rule. If the rule's action is **DO**, then the resulting clause inserts tuples into the outgoing table directly. If the rule's action modifies an n-ary table R via the **INSERT** or **DELETE** keywords, the clause uses n-ary helper tables R_{add} or R_{del}, which hold the tuples to be added to and removed from the controller state after an event is processed.

If S is a controller state, let $S^{\uparrow}$ represent the unique least expansion of S that satisfies all rule clauses. That is, while S contains a table for each **TABLE**, $S^{\uparrow}$ also contains ephemeral R_{add} and R_{del} tables for each **TABLE** as well as tables for each **OUTGOING** declaration. These **OUTGOING** tables are consumed by Flowlog and dictate which outgoing events it should send. The ephemeral tables for each state table R dictate its value in the next state as follows:

$$R_{next} = (R^S \setminus R_{del}^{S^{\uparrow}}) \cup R_{add}^{S^{\uparrow}}$$

In other words, **INSERT** overrides **DELETE** in Flowlog—the next state's R contains the pre-state's R, minus R_{del}, plus R_{add}.

4 Proactive Compilation for Flowlog

While Flowlog programs receive many kinds of input—both packets and external notifications—packets remain the most common and time-sensitive stimuli. No software-defined network can scale to the level required by large networks if it sends every arriving packet to the controller for instructions. This complicates the implementation of a tierless SDN language; rather than simply have all packets be processed by the controller in accordance with the program, the Flowlog runtime is forced to solve three related, but distinct, challenges:

ON `IN(in)` **DO** `OUT(o1, ..., ok)` **WHERE** `rf`	$\forall in, o_1, ...o_k\ \exists e_1, ..., e_k\ OUT(o_1, ...o_k) \leftarrow IN(in) \wedge T_{fmla}(\texttt{rf})$
ON `IN(in)` **INSERT** `(o1, ..., ok)` **INTO** `R` **WHERE** `rf`	$\forall in, o_1, ...o_k\ \exists e_1, ..., e_k\ R_{add}(o_1, ...o_k) \leftarrow IN(in) \wedge T_{fmla}(\texttt{rf})$
ON `IN(in)` **DELETE** `(o1, ..., ok)` **FROM** `R` **WHERE** `rf`	$\forall in, o_1, ...o_k\ \exists e_1, ..., e_k\ R_{del}(o_1, ...o_k) \leftarrow IN(in) \wedge T_{fmla}(\texttt{rf})$

(All rules existentially quantify variable occurrences that are free and not in $\{in, out_1, ..., out_k\}$; hence the e_is.)

T_{fmla}(**NOT** `f`)	=	$\neg T_{fmla}$(`f`)
T_{fmla}(`f1` **AND** `f2`)	=	T_{fmla}(`f1`) $\wedge$ T_{fmla}(`f2`)
T_{fmla}(`t1 = t2`)	=	T_{term}(`t1`) $= T_{term}$(`t2`)
T_{fmla}(`P(t1, ..., tk)`)	=	$\exists x_1, ..., x_k\ P(T_{term}(\texttt{t1}), ..., T_{term}(\texttt{tk}))$
		x_1, ..., x_k are the fresh variables introduced by `ANY`s in the original formula.

T_{term}(`c`)	=	c
T_{term}(`x`)	=	x
T_{term}(`x.fld`)	=	$fld(x)$
T_{term}(`ANY`)	=	x_{fresh}

Figure 3: Rule-formula to formula (T_{fmla}) and Rule-term to term (T_{term}) transformation functions. Without loss of generality, we provide every rule with its own **ON** trigger and assume that disjunction in rule bodies has been removed, resulting in multiple rules. x_{fresh} denotes a fresh variable.

1. It must compile a program's *forwarding behavior* to equivalent OpenFlow [20] rules whenever possible, including references to both local and remote state;

2. it must discern which packets can trigger non-forwarding behavior, such as emission of an event or a state change, and produce OpenFlow rules that send those packets—and only those packets—to the controller; and,

3. if a rule uses features that are not supported by switch tables (we detail these cases later), the compiler determines the class of packets that must be sent to the controller for correct handling. This situation applies to both forwarding and non-forwarding rules.

As Section 3 demonstrated, Flowlog rules can involve equality between packet fields, negation, database state, and numerous other features not supported by OpenFlow. Moreover, rules can contain existentially quantified variables that, at first glance, require searching and backtracking in the state to properly handle. Simply put, Flowlog rules are strictly more powerful than OpenFlow 1.0 flow rules. It is therefore reasonable to wonder: can a non-trivial amount of Flowlog really be compiled faithfully? This section will show it can be. Figure 4 shows the compilation dataflow.

Our proactive ruleset compiler has three stages:

1. First (Section 4.1), it simplifies each rule and identifies the compilable forwarding rules. Both non-forwarding rules (state updates, emission of fresh packets, etc.) and non-compilable forwarding rules require switches to send packets to the controller; fortunately, these notifications can be extensively filtered based on the program's structure.

2. Second (Section 4.2), it partially evaluates the ruleset at the current controller state, producing a new ruleset that has no references to state tables. Since the original ruleset defines a function that accepts a state and an incoming tuple and returns a set of outgoing tuples, the resulting ruleset depends only on the incoming tuple.

3. Finally (Section 4.3), it compiles the new ruleset automatically to flow table rules in two steps. First, it converts to NetCore [21], a stateless forwarding policy language for OpenFlow. Second, it applies NetCore's compiler to produce flow table rules.

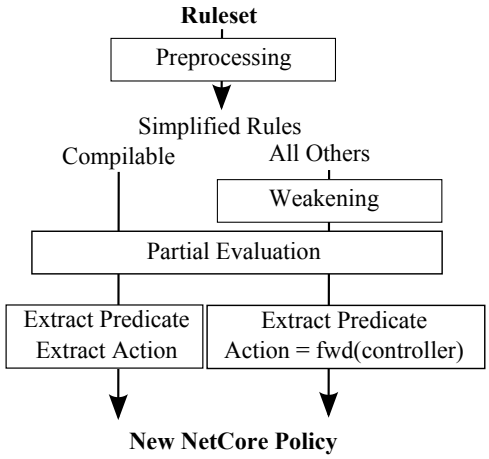


Figure 4: Flowlog's compilation process. Rules are first preprocessed before being checked for compilability, then (if uncompilable) weakened before being partially evaluated in the current state. After partial evaluation, the rule is re-written as a stateless NetCore policy.

Example: Forwarding For intuition into the compilation process, consider the following example rule.

```
1 ON packet_in(p):
2   DO forward(new) WHERE
3     learned(p.locSw,new.locPt,p.dlDst);
```

This rule says: "Forward `p` on a port corresponding to its location and destination, provided the controller has learned that correspondence". It compiles to the

following rule clause:

$$\forall p, new\,.\,forward(new) \Longleftarrow \\ learned(locSw(p), locPt(new), dlDst(p)) \\ \wedge packet\_in(p)$$

Suppose the current state contains $learned = \{\langle 1,2,3\rangle, \langle 1,3,2\rangle, \langle 1,4,3\rangle\}$. Then the *learned* expression in the above clause is equivalent to:

$$((locSw(p) = 1 \wedge locPt(new) = 2 \wedge dlDst(p) = 3) \vee \\ (locSw(p) = 1 \wedge locPt(new) = 4 \wedge dlDst(p) = 3) \vee \\ (locSw(p) = 1 \wedge locPt(new) = 3 \wedge dlDst(p) = 2))$$

Re-written as a NetCore policy, this is just:

```
(filter (locSw = 1 and dlDst = 3);
  fwd(2) | fwd(4)) +
(filter (locSw = 1 and dlDst = 2); fwd(3))
```

This policy can remain in place in flow tables until such time as the `learned` table changes.

Example: State Change Flowlog provides a "see-every-packet" abstraction. For instance, the following program appears to execute entirely on the controller:

```
1 ON packet_in(p):
2   INSERT (p.locSw, p.locPt, p.dlSrc)
3     INTO learned WHERE
4     NOT learned(p.locSw, p.locPt,
5                 p.dlSrc);
```

With the exception of Maple [32], existing languages with this abstraction require the programmer to carefully maintain separate logic for packet forwarding and controller notifications. In contrast, the Flowlog runtime handles controller notification automatically; the only packets the controller needs are those that provably alter the controller state. Flowlog's compiler automatically builds and deploys a NetCore policy that applies to all such packets. For example, suppose the current state is $learned = \{\langle 1,2,3\rangle, \langle 1,3,2\rangle\}$. The following NetCore policy ensures the controller sees the packets it needs to, and no more:

```
if not ((locSw = 1 and locPt = 2 and dlSrc = 3) or
        (locSw = 1 and locPt = 3 and dlSrc = 2))
   then fwd(controller)
```

4.1 Simplification and Compilability

Before compiling a ruleset, Flowlog removes unnecessary variables. For example, if `p` is the incoming packet, the condition `learned(p.locSw, y, p.dlSrc) and x=p.locPt and y=x` would be rewritten as `learned(p.locSw, p.locPt, p.dlSrc)`. This process eliminates hidden dependencies, simplifying compilation.

Each rule is then subjected to a compilability check. Table 2 lists the conditions under which a rule cannot be compiled. If a rule fails one or more tests, it either triggers an outright error or must be handled by the controller. For instance, a rule that compares the incoming packet's layer-2 source and destination fields is easily expressed in Flowlog as `p.dlSrc = p.dlDst` and can be checked reactively by the controller, but is not supported by OpenFlow 1.0 forwarding tables.

Finally, to reduce the number of packets that must be sent to the controller, Flowlog *weakens* the **WHERE** condition of each uncompilable rule to obtain a compilable overapproximation. A rule clause is a conjunction of literals (i.e., positive or negative assertions about state or equality), and weakening removes objectionable literals (Table 2) from the clause. Removing parts of a conjunction yields a new formula that is implied by the original, so it is a sound overapproximation.

4.2 Partial Evaluation

Partial evaluation removes references to state tables within each rule, replacing them with simple equalities involving only constants and variables. Figure 5 defines the partial evaluation function (T_{pe}), and other transformation functions used below. Once partial evaluation is complete, the compiler distributes out any disjunctions introduced by partially evaluating *positive* literals, resulting in a new set of clause formulas. This is done so new equalities constraining the outgoing packet, if any, are immediately available at the top level of the conjunction. (Disjunctions coming from negative literals are left in place; this is safe since outgoing packet fields that occur in negated table references are forbidden.) Any clauses that were partially evaluated to a contradiction are removed.

4.3 Extracting NetCore Policies

The policies that the proactive compiler produces have two parts: a stateless filtering condition on packets (the predicate) and the set of actions to apply when the predicate matches. NetCore predicates support the essential Boolean operators—`or`, `and`, `not`—as well as *filters* over header fields and switch identifiers.

An equivalent NetCore policy for each clause is created using the T_{pred} (extract predicate) and T_{act} (extract action) functions defined in Figure 5. Predicate extraction only involves the incoming packet; other literals map to the trivially true predicate `all`. Non-forwarding rule clauses are always assigned the send-to-controller action. For each forwarding rule clause, the compiler extracts an action assertion such as "forward on port 3". Since

Condition	Example	Explanation
(a) Forbidden new-packet field assignment	`new.nwProto = 5`	Not allowed in OpenFlow 1.0
(b) Different fields in old-to-new assignment	`new.dlSrc = old.dlDst`	Not allowed in OpenFlow 1.0
(c) Negatively constrained new-packet field	`new.dlSrc != 5` or **not** `R(new.dlSrc)`	Forbid packet avalanche
(d) Reflection on incoming packet in equality	`old.dlSrc = old.dlDst`	Not allowed in OpenFlow 1.0
(e) Non-assignment condition of new packet	`new.locPt = new.dlSrc`	For compilation speed
(f) Multi-way join on state tables	`R(3,X)` **and** `R(X,4)`	For compilation speed

Table 2: Situations that cause a rule to be weakened and dealt with by the controller. In a forwarding rule, (a–b) are forbidden at compile time. The "flood" condition, `new.locPt != p.locPt`, is the sole exception to (c); other forms would cause a plethora of outgoing packets. (d–f) are allowed at compile time, but force weakening of forwarding rules. By eliminating complex join conditions from compilation (e–f), we avoid the necessity of solving a search problem to compile rules; after preprocessing, existential variables appear in compiled rules only as placeholders for "don't-care" positions in rule formulas.

Partial Evaluation: $States \times Formulas \rightarrow Formulas$

$$T_{pe}(S, R(t_1,...,t_n)) = \bigvee_{\langle c_1,...,c_n\rangle \in R^S} (t_1 = c_1 \wedge ... \wedge t_n = c_n)$$
$$T_{pe}(S, t_1 = t_2) = t_1 = t_2$$
$$T_{pe}(S, \neg\alpha) = \neg T_{pe}(S,\alpha)$$
$$T_{pe}(S, \beta \vee \gamma) = T_{pe}(S,\beta) \vee T_{pe}(S,\gamma)$$
$$T_{pe}(S, \beta \wedge \gamma) = T_{pe}(S,\beta) \wedge T_{pe}(S,\gamma)$$

Predicate Extraction: $Rule\ Clauses \rightarrow Pred$

$$T_{pred}(oldpkt.fld = c) = \texttt{fld = c}$$
$$T_{pred}(t = c) = \texttt{all}$$
$$T_{pred}(\neg\alpha) = \texttt{not}\ T_{pred}(\alpha)$$
$$T_{pred}(\beta \wedge \gamma) = T_{pred}(\beta)\ \texttt{and}\ T_{pred}(\gamma)$$

Action Extraction: $Rule\ Clauses \rightarrow 2^{Action}$

$$T_{act}(newpkt.locPt = c) = \{\texttt{fwd(c)}\}$$
$$T_{act}(newpkt.fld = c) = \{\texttt{set(fld, c)}\}$$
$$T_{act}(\neg\alpha) = \emptyset$$
$$T_{act}(\beta \wedge \gamma) = T_{act}(\beta) \cup T_{act}(\gamma)$$

Figure 5: Transformation functions used during compilation.

contradictions were removed (Section 4.2), only one such assertion is made per clause. Clauses containing inequalities of the form `new.locPt != old.locPt` are added to the predicate in a final pass after the forwarding action is extracted. Once a predicate and action has been obtained for each clause, the compiler assembles the final policy by generating a sub-policy for each action that filters on the disjunction of all matching predicates, and then taking the union of those sub-policies.

5 Verification

To verify Flowlog programs, we use the Alloy Analyzer [10]. Alloy has a first-order relational language, which makes it a good match for Flowlog's first-order relational semantics. In addition, Alloy is automated and generates counterexamples when properties fail to verify.

We have created a compiler from Flowlog rulesets to Alloy specifications. The conversion is fully automated, although users must provide types (e.g., IP address) for constants used in the original program; this type information is used for optimization. By default, the compiler abstracts out the caching process and treats **REMOTE TABLE**s as constant tables. If analysis goals involve remote state, axioms about the behavior of remote code (e.g., "the routing library always gives a viable path") can be added manually.

Because of tierlessness, Alloy models created from Flowlog programs need not consider the eccentricities of the OpenFlow protocol or individual switch rules, and so reasoning benefits from the illusion that all packets are processed by the controller. This simplifies the resulting Alloy models (and improves analyzer performance), and also makes it easier for users to express properties across tiers. Ordinarily, for example, checking dependencies between forwarding behavior and state change would involve expressing the desired behavior for both packets that reach the controller and packets handled by switches; when reasoning about Flowlog, this split is unnecessary.

Inductive Properties An important class of program properties, which we call *inductive*, take the form: "If P holds of the controller state, then no matter what packet arrives, P will continue to hold in the next state." This property serves to prove that P always holds in any reachable state, so long as it holds of the starting state. Many desirable goals can be expressed in this way, and they are often independent of network topology.

To illustrate the power of this class of properties, consider our NIB example (Section 2). A piece of the NIB program gradually computes the transitive closure of the network topology. But does it really compute transitive closure faithfully? As probe packets arrive, the `ucTC` table needs to contain the transitive closure of the graph defined by all the links seen *so far*. Figure 6 shows how to encode this property in Alloy.

Running this analysis on an older version of the NIB program revealed a missing rule: we had failed to account for the case in which two mutually unreachable sub-networks become connected by an incom-

```
all st: State, st2: State, ev: EVpacket |
  transition[st, ev, st2] and
  ev.dltyp = C_0x1001 and
  (st.uctc = ^(st.uctc)) implies
    st2.uctc =
      ^(st.uctc + (ev.dlsrc -> ev.locsw))
```

Figure 6: Example Alloy property: "For all states (`st`) with `ucTC` transitively closed, the program only transitions to states (`st2`) with a transitively closed extension of `ucTC` by the arriving probe packet (`ev`)'s src/dst". Recall that the source switch ID is in the packet's `dlSrc` field. The `^` operator denotes transitive closure in Alloy. We chose `C_` to prefix constant identifiers.

Property	Time(ms)	B
NIB		
Reachability computed correctly (4sw)	40	
Reachability (with bug)	35	
Spanning tree never has cycles	58	✓
Timer correctly updates persistent tables	23	✓
Correctly capture host location changes	65	✓
Stolen Laptop		
Only police can un-flag a laptop	4	✓
Learning Switch		
≤ 1 port learned per host per switch	14	✓
Only switch failure can restart flooding	14	✓

Table 3: Example properties with time to verify or find a counterexample. The **B** column shows whether sufficient bounds could be established, as described in Section 5. Alloy 4.2/3.1 GHz Core i5/8 GB RAM.

ing probe packet. That is, we were missing this rule:

```
1 ON packet_in(p) WHERE p.dlTyp = 0x1001
2   AND dstSw = p.locSw AND srcSw = p.dlSrc:
3   INSERT (sw1, sw2) INTO ucTC
4     WHERE ucTC(sw1, srcSw)
5     AND   ucTC(dstSw, sw2);
```

This rule is necessary due to the subtle nature of computing reachability from each probe in succession, rather than having access to the entire table and applying recursion. Alloy was able to demonstrate this bug on a network of only four switches.

Using this method, we have successfully verified properties of (or found bugs in) multiple Flowlog programs including the NIB, the stolen laptop alert program, and a MAC learning switch. Table 3 lists several along with the time they took to verify: well under a second.

Completeness Alloy performs *bounded* verification: it requires a size bound for each type, which it uses to limit the search for a counterexample. For instance, for our NIB verification we might instruct Alloy to search up to four switches (irrespective of the wiring between them), three distinct MAC addresses, etc. Because of its bounded nature, Alloy is not in general *complete*: it will fail to find counterexamples larger than the given bound. Since individual properties are a result of purpose-specific program goals, properties and their associated bounds must be entered manually.

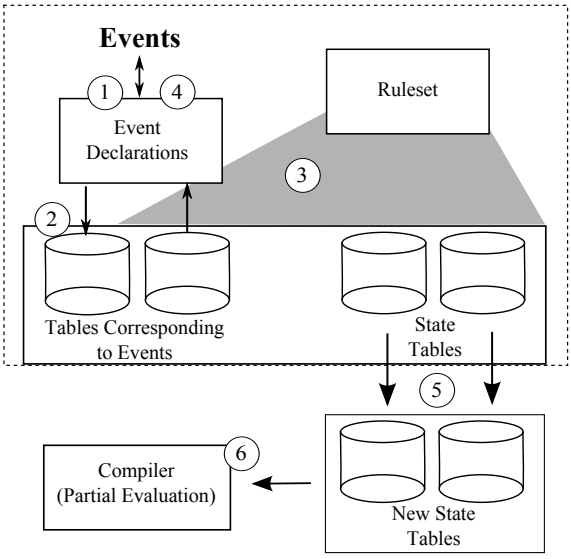


Figure 7: Flowlog's workflow for responding to events. The boxed portion of the diagram appeared as Figure 1.

Fortunately, for many common types of analyses, we can exploit prior work [23] to compute (small) size bounds that are sufficient to find a counterexample, should one exist. All but one of the properties we verified is amenable to this technique; the exception is reachability, because the technique does not support transitive-closure (Table 3). Yet broad experience with Alloy indicates that many bugs can be found with fairly small bounds (e.g., four switches for our transitive-closure bug). Moreover, bounds on other objects (e.g., non-switches) can still be produced for all the inductive properties that we tested.

6 Implementation and Performance

The current Flowlog implementation uses OpenFlow 1.0 [20] and Frenetic [5] for packet-handling, Thrift RPC (`thrift.apache.org`) for orchestrating events and remote state, and the XSB [28] Prolog engine for evaluation. Flowlog is implemented in OCaml.

Figure 7 sketches the controller's workflow. When an event arrives (1) the controller converts it into a tuple and places it in the appropriate input table via XSB's `assert` command (2). Then, for each outgoing and state-modification table, the controller queries XSB to obtain a set of outgoing tuples (3) which are converted to events (4). Then each state-modification tuple is `asserted` or

`retracted` to result in the new state (5). Finally, proactive compilation (6) is performed on the new state, producing a NetCore policy.

External Events Flowlog evaluation is triggered by a general set of events; the runtime must watch for more than just packet arrivals. For instance, the runtime sends an event to the controller whenever a switch registers or goes down, and as seen in Section 2, external applications may also interact with Flowlog through events. Our hypothetical campus police-officer informs Flowlog to register a stolen laptop by using a small application (around 100 lines, most of which is boilerplate) that uses Thrift to send an asynchronous message to Flowlog.

Remote Tables and Caching Flowlog rules reference a database of relational facts. As seen in Section 3, tables can be declared either as local or remote. A local table is managed internally by the controller (via `assert` and `retract` statements to XSB), while a remote table is merely an abstraction over callouts to external code. Like events, these callouts use Thrift RPC to interact with external code. Unlike events, callouts are synchronous. Callouts have the form of an ordinary state-referencing formula, $R(t_1,...,t_n)$, but each t_i must be either a constant value or a variable. After Flowlog queries the correct external application, the reply contains a set of tuples of constants—one constant for each variable in the query.

Although the rules see no distinction between local and remote tables, in practice it would be impractical or impossible to obtain *entire* remote tables (such as an infinitely large table that represents the addition of numbers). Therefore, Flowlog obtains tuples from external code only when they are needed by a rule. A naïve implementation could simply obtain remote tuples every time they were required; however, that would mean forwarding rules could not be compiled if they referred to external code. Instead, we cache remote tuples for the declared time-to-live. Since we maintain the remote cache in XSB, when it comes time to react to an event, the controller handles remote and local state in exactly the same way: via XSB.

When an event arrives, we invalidate cached tuples whose time-to-live has expired. If the expired tuples were used in a compiled policy, we force an update of the cache and provide switches with the new policy. External programs are expected to not update their internal state or otherwise provide inconsistent results within the **TIMEOUT** values of their Flowlog definitions.

Handling Overlapping Rules In some SDN applications, switches will forward a packet on the data plane and also send it to the controller. If we kept the pre-compilation ruleset unmodified on the controller, this could lead to packet duplication due to the compiled forwarding rules: packets would be forwarded once by the switch tables, and forwarded again by the same rule's action on the controller.

Since the compilability of a rule is independent of controller state, we can determine which rules these are at program startup. We then leave these rules out of what we pass to XSB, and disallow the controller from taking duplicate actions. However, since there is a delay between the controller's state change and corresponding rules being installed on the switches, this is not a perfect solution: packets may be in-transit during deployment of the new policy. This issue is not unique to Flowlog, and has been noted by others [26].

Performance and Scalability Flowlog is proactively compiled to switch rules whenever possible. From the traffic forwarding perspective, therefore, Flowlog is largely dependent on what is supported in switch hardware. We have confirmed experimentally that, as one would hope, the controller receives no unnecessary packets. For instance, a Flowlog learning switch never sends the controller a packet once that packet's source location is learned, and eventually the controller is not burdened at all. Packet-counts were confirmed by both `ifconfig` counters and packet events seen by Flowlog. We used `ping` packets and a Mininet [16]-hosted virtual network to simulate network traffic. We tested on tree topologies with 3 and 7 switches as well as a cyclic 3-switch topology to test controller robustness, and have begun testing on larger topologies as well.

Because forwarding in Flowlog is as fast as hardware allows, one scalability question remains: since Flowlog compiles the controller state into NetCore policies, how does this scale as the controller's database grows? The size of the NetCore policy we produce depends on how table references appear in the ruleset. The compiler produces a policy fragment for each rule, whose size is proportional to the number of clauses generated by partial evaluation (Section 4.2). Partial evaluation replaces state table references in each rule with the disjunction of every matching tuple in that table. The largest number of clauses produced by a rule that references tables R_1 through R_k is $|R_1| \times \cdots \times |R_k|$, which is the best achievable in the worst case. (This ensues because we don't need to lift negated disjunctions, a technical detail that we lack space to describe.) We also simplify the resulting policies, which further reduces their size in practice. To convert policies to flow-table rules, we rely upon NetCore's optimizing compiler [21].

To evaluate the quality of the NetCore policies our compiler produces, we ran a Flowlog learning-switch program, using `dpctl dump-flows` to count the maximum num-

ber of table entries it produced on per switch. We then did the same with the OCaml learning-switch module in the Frenetic repository. The Frenetic example produced a maximum of 25 rules per switch for the 3-switch tree and 81 rules per switch for the 7-switch tree. Our program initially produced 40 rules and 108 rules respectively. The increased number of rules was because Flowlog did not make use of OpenFlow's built-in flood action, whereas the OCaml program did. After adding an optimization to our learning-switch program that forced the use of the flood action, we saw the same number of table entries as with the Frenetic version. This indicates that our compiler can match the scalability of existing programs that use NetCore.

7 Related Work

Our work here draws on a prior workshop paper [24]. The previous version mentioned, yet did not describe or implement, external events and state. Our proactive compiler, conversion to Alloy, topology-independent verification, and implementation are also new. We compare other SDN programming languages side-by-side with Flowlog in Table 4, and discuss each in detail below.

FML [9] provides a stateful, rule-based idiom for forwarding policies. It too disallows recursion and admits negation. FML can read from, but not modify, the underlying system state. It responds to new flows reactively, whereas Flowlog proactively compiles to switch tables whenever possible. FML does not provide abstractions for external code, and does not address verification.

Frenetic [5] is a functional-reactive (FRP) language that responds to fine-grained network events. While Frenetic was originally reactively compiled, it has since been extended with proactive compilation for stateless NetCore [21] policies. State and interaction with external code must be managed by OCaml wrapper applications. Frenetic also includes switch-based events and rich query constructs; Flowlog lacks abstractions for queries, yet provides more general events. Verification of full Frenetic programs has not been addressed, although Gutz, et al. [8] use model-checking to prove slice isolation properties of NetCore policies. Our runtime is currently implemented atop Frenetic's OCaml library and uses NetCore's optimizing compiler. Guha et al. [7] have created a verified compiler for a subset of NetCore; our compiler has only been tested, not proven correct.

Pyretic [22] implements NetCore [21] in Python, and introduced sequential composition of network programs. Though we do not address program composition explicitly, sequential and parallel composition are roughly analogous to Flowlog's relational join and union, respectively. Since its initial publication, Pyretic has been extended with proactive compilation. Verification of Pyretic programs has not been discussed.

Flog [11] is stateful and rule-based. It allows recursion but not explicit negation in rule bodies; negation is implied in some cases by rule-overriding. Flog has no notion of callouts or external events unrelated to packets, and the paper does not address verification. Flog works at the microflow level, whereas Flowlog is proactive.

Procera [31] is another FRP SDN language. As Procera is embedded in Haskell, programs have access to general state and external callouts. Procera allows programs to react to external events, but does not directly support issuing events or external queries. Like Frenetic, Procera provides query functionality that Flowlog does not. To our knowledge, Procera programs have not been verified, and it is unclear whether the flow constraints they generate are proactively compiled.

Nlog [14], a rule-based configuration language, is part of a larger project on network virtualization. When system state changes, an Nlog program dictates a new virtual forwarding policy. Nlog's inability to modify controller state means that it is not "tierless". Like Flowlog, Nlog is also proactively compiled to flow tables, and our relational abstraction for callouts is similar to Nlog's. Verification of Nlog programs has not been discussed.

Maple [32] is a controller platform that unifies control- and data-plane logic; Flowlog goes further by also integrating controller state, creating a tireless abstraction. Unlike in Flowlog, Maple programs are compiled reactively, and have not been verified.

A number of other rule-based languages also merit discussion, although they were not built for SDNs and do not compile to flow tables. NDLog [17] and OverLog [18] are declarative, distributed programming languages. In these languages, each tuple in the relational state resides on a particular switch. This is in contrast to the single controller state assumed by Flowlog. These languages support recursion as in ordinary Datalog. Wang, et al. [33] verify NDLog programs via interactive theorem provers, and some of the properties they verify are topology-independent. Our compiler to Alloy requires far less user effort and is simplified by Flowlog's lack of recursion. Alvaro, et al. [3] present Dedalus, a variant of Datalog with a notion of time. Our treatment of state change is similar to theirs, except that theirs is complicated by recursion. Active networking [30] is a forerunner of SDN where packets carry programmatic instructions for switches. Like SDN, active networking has inspired language design efforts. One such is ANQL [27], a SQL-like language for defining packet filters and triggering external code. Flowlog echoes ANQL's view of packets as entries in a database, but supports more general external stimuli. Verification of ANQL has not been discussed.

There is also a rich landscape of related SDN verification. Canini, et al. [4] find bugs in Python controller

Language	Type	State	Rec?	Neg?	Compilation	Reasoning?	Callouts
Flog [11]	Rule-Based	✓	✓	✗	Reactive	✗	✗
FML [9]	Rule-Based	✓	✗	✓	Reactive	✗	✗
Frenetic [5]	FRP	$✓_{pol}$	✗	✓	Reactive	✗	✗
Frenetic OCaml Environment	Functional	$✓_{PL}$	$✓_{PL}$	✓	via NetCore	$✗_{PL}$	$✓_{PL}$
NetCore [21]	DSL	✗	✗	✓	Proactive	✓	✗
Nlog [14]	Rule-Based	✓	✗	✗	Proactive	✗	✓
NOX [6]	Imperative	$✓_{PL}$	$✓_{PL}$	✓	Manual	$✗_{PL}$	$✓_{PL}$
Procera [31]	FRP	✓	✗	✓	Unclear	✗	✗
Pyretic [22]	Imperative	$✓_{pol}$	$✓_{PL}$	✓	Proactive	✗	$✓_{PL}$
Flowlog	Rule-Based	✓	✗	✓	Proactive	✓	✓

Table 4: SDN language comparison. **Rec?** and **Neg?** mean recursion and negation, respectively. A ✓ means that a feature is present and a ✗ that it is not; $✓_{PL}$ denotes that a feature's presence is due to embedding in a Turing-complete programming language. In the **State?** column, $✓_{pol}$ indicates that maintaining a stateful forwarding policy is possible, but that general state requires wrappers in a Turing-complete language. In the **Reasoning?** column, a $✗_{PL}$ indicates that *sound* reasoning is made non-trivial by Turing-completeness, and a ✗ means that verification has not been attempted.

programs. Their work required the creation of a purpose-built model-checker. Although their tool successfully finds bugs, it is limited by the undecidability of Python code analysis. Flowlog's expressivity is deliberately limited to avoid this concern. Skowyra et al. [29] use model-checking to find bugs in SDN programs sketched in their prototyping language, but focus solely on verification, not execution. Other reasoning tools [2, 8, 13, 19, 25, 34] analyze a fixed, stateless forwarding policy, either statically or at runtime. Including transitions between states, as we do, is necessarily more complex.

8 Discussion and Conclusion

To our knowledge, Flowlog is the first tierless SDN programming language, the first stateful rule-based language for SDNs that proactively compiles to flow-table rules, and the first such language to provide rich interfaces to external code. Tierlessness simplifies the process of SDN programming and simultaneously enables cross-tier verification of the SDN system.

Since tierlessness precludes manual handling of flow-table rules, automatic flow-table management is necessarily a key part of any tierless SDN language. There are other such strategies besides proactive compilation; a prototype version of Flowlog simply sent all packets to the controller. However, a proactive approach minimizes controller interaction and thus shows that a tierless language can be performant.

In order to support efficient, proactive compilation and verification, we opted to limit Flowlog's expressive power. Even with these limitations, we have built non-trivial applications. Moreover, events and remote tables allow Flowlog programs to access, when necessary, external code in languages of arbitrary power.

Future Work It is possible to strengthen Flowlog without abandoning our limitations on expressive power. We plan to migrate to a newer version of OpenFlow soon, which removes several uncompilable constructs in Table 2. Flowlog's general event framework could support a query system like that seen in Frenetic [5] and other languages. Flowlog's relational idiom supports the addition of new features. For instance, we have recently added address masking (e.g., matching packets coming from `10.0.0.0/24`) to Flowlog by taking advantage of the fact that masks are simply relations over IP addresses.

There are also several promising directions to take verification in Flowlog. For instance, we suspect that Flowlog's restrictions could enable the sound and complete use of techniques like symbolic execution for verifying trace properties. Another important analysis, change-impact—which describes the *semantic* consequences of a program change—is undecidable for general-purpose languages, yet is decidable for Flowlog.

Acknowledgments We thank the anonymous reviewers for their comments. We are grateful to Daniel J. Dougherty, Kathi Fisler, Rodrigo Fonseca, Nate Foster, Arjun Guha, Tim Hinrichs, Jonathan Mace, Sanjai Narain and others at Applied Communication Sciences, Joshua Reich, and David Walker, for discussions and feedback. We are grateful to Jennifer Rexford for several enlightening conversations, for shepherding this paper, and for her trenchant analysis of drinking styles. We thank the Alloy, Frenetic, and XSB teams for excellent software that we could build upon. This work is partially supported by the NSF. Andrew Ferguson is supported by an NDSEG Fellowship. Michael Scheer was supported by a Brown University Undergraduate Research Award.

References

[1] ABITEBOUL, S., HULL, R., AND VIANU, V. *Foundations of Databases*. Addison-Wesley, 1995.

[2] AL-SHAER, E., AND AL-HAJ, S. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *Workshop on Assurable and Usable Security Configuration* (2010).

[3] ALVARO, P., MARCZAK, W. R., CONWAY, N., HELLERSTEIN, J. M., MAIER, D., AND SEARS, R. Dedalus: Datalog in time and space. In *Datalog Reloaded 2010* (2010), pp. 262–281.

[4] CANINI, M., VENZANO, D., PEREŠÍNI, P., KOSTIĆ, D., AND REXFORD, J. A NICE way to test OpenFlow applications. In *Networked Systems Design and Implementation* (2012).

[5] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A network programming language. In *International Conference on Functional Programming (ICFP)* (2011).

[6] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an operating system for networks. *ACM Computer Communication Review 38*, 3 (July 2008), 105–110.

[7] GUHA, A., REITBLATT, M., AND FOSTER, N. Machine-verified network controllers. In *Programming Language Design and Implementation (PLDI)* (2013).

[8] GUTZ, S., STORY, A., SCHLESINGER, C., AND FOSTER, N. Splendid isolation: A slice abstraction for software-defined networks. In *Workshop on Hot Topics in Software Defined Networking* (2012).

[9] HINRICHS, T., GUDE, N., CASADO, M., MITCHELL, J., AND SHENKER, S. Practical declarative network management. In *Workshop: Research on Enterprise Networking (WREN)* (2009).

[10] JACKSON, D. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, April 2006.

[11] KATTA, N. P., REXFORD, J., AND WALKER, D. Logic programming for software-defined networks. In *Workshop on Cross-Model Design and Validation (XLDI)* (2012).

[12] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *Networked Systems Design and Implementation* (2012).

[13] KHURSHID, A., ZHOU, W., CAESAR, M., AND GODFREY, P. B. VeriFlow: Verifying network-wide invariants in real time. In *Workshop on Hot Topics in Software Defined Networking* (2012).

[14] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., GUDE, N., INGRAM, P., JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network Virtualization in Multi-tenant Datacenters. In *Networked Systems Design and Implementation* (2014).

[15] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., AND SHENKER, S. Onix: a distributed control platform for large-scale production networks. In *Operating Systems Design and Implementation* (2010).

[16] LANTZ, B., HELLER, B., AND MCKEOWN, N. A network in a laptop: Rapid prototyping for software-defined networks. In *Workshop on Hot Topics in Networks* (2010).

[17] LOO, B. T., CONDIE, T., GAROFALAKIS, M. N., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative networking. *Communications of the ACM 52*, 11 (2009), 87–95.

[18] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing declarative overlays. In *Symposium on Operating Systems Principles* (2005).

[19] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with Anteater. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (2011).

[20] 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 38*, 2 (Mar. 2008), 69–74.

[21] MONSANTO, C., FOSTER, N., HARRISON, R., AND WALKER, D. A compiler and run-time system for network programming languages. In *Principles of Programming Languages (POPL)* (2012).

[22] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing software-defined networks. In *Networked Systems Design and Implementation* (2013).

[23] NELSON, T., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. Toward a more complete Alloy. In *International Conference on Abstract State Machines, Alloy, B, and Z* (2012).

[24] NELSON, T., GUHA, A., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. A balance of power: Expressive, analyzable controller programming. In *Workshop on Hot Topics in Software Defined Networking* (2013).

[25] PORRAS, P., SHIN, S., YEGNESWARAN, V., FONG, M., TYSON, M., AND GU, G. A security enforcement kernel for OpenFlow networks. In *Workshop on Hot Topics in Software Defined Networking* (2012).

[26] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (2012).

[27] ROGERS, C. M. ANQL - an active networks query language. In *International Working Conference on Active Networks* (2002).

[28] SAGONAS, K., SWIFT, T., AND WARREN, D. S. XSB as an efficient deductive database engine. In *International Conference on the Management of Data* (1994).

[29] SKOWYRA, R., LAPETS, A., BESTAVROS, A., AND KFOURY, A. Verifiably-safe software-defined networks for CPS. In *High Confidence Networked Systems (HiCons)* (2013).

[30] TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. A survey of active network research. *IEEE Communications Magazine* (1997).

[31] VOELLMY, A., KIM, H., AND FEAMSTER, N. Procera: A language for high-level reactive network control. In *Workshop on Hot Topics in Software Defined Networking* (2012).

[32] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)* (2013).

[33] WANG, A., BASU, P., LOO, B. T., AND SOKOLSKY, O. Declarative network verification. In *Practical Aspects of Declarative Languages* (2009).

[34] 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 *IEEE Conference on Computer Communications* (2005).

Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags

Seyed Kaveh Fayazbakhsh[*] *Luis Chiang*[†] *Vyas Sekar*[*] *Minlan Yu*[‡] *Jeffrey C. Mogul*[⋆]
[*]*Carnegie Mellon University* [†]*Deutsche Telekom Labs* [‡]*USC* [⋆]*Google*

Abstract

Middleboxes provide key security and performance guarantees in networks. Unfortunately, the dynamic traffic modifications they induce make it difficult to reason about network management tasks such as access control, accounting, and diagnostics. This also makes it difficult to integrate middleboxes into SDN-capable networks and leverage the benefits that SDN can offer.

In response, we develop the FlowTags architecture. FlowTags-enhanced middleboxes export tags to provide the necessary causal context (e.g., source hosts or internal cache/miss state). SDN controllers can configure the *tag generation* and *tag consumption* operations using new FlowTags APIs. These operations help restore two key SDN tenets: *(i)* bindings between packets and their "origins," and *(ii)* ensuring that packets follow policy-mandated paths.

We develop new controller mechanisms that leverage FlowTags. We show the feasibility of minimally extending middleboxes to support FlowTags. We also show that FlowTags imposes low overhead over traditional SDN mechanisms. Finally, we demonstrate the early promise of FlowTags in enabling new verification and diagnosis capabilities.

1 Introduction

Many network management tasks are implemented using custom *middleboxes*, such as firewalls, NATs, proxies, intrusion detection and prevention systems, and application-level gateways [53, 54]. Even though middleboxes offer key performance and security benefits, they introduce new challenges: (1) it is difficult to ensure that "service-chaining" policies (e.g., web traffic should be processed by a proxy and then a firewall) are implemented correctly [49, 50], and (2) they hinder other management functions such as performance debugging and forensics [56]. Our conversations with enterprise operators suggest that these problems get further exacerbated with the increasing adoption of virtualized/multi-tenant deployments.

The root cause of this problem is that traffic is modified by dynamic and opaque middlebox behaviors. Thus, the promise of software-defined networking (SDN) to enforce and verify network-wide policies (e.g., [39, 40, 44]) does not extend to networks with middleboxes. Specifically, middlebox actions violate two key SDN tenets [24, 32]:

1. ORIGINBINDING: There should be a strong binding between a packet and its "origin" (i.e., the network entity that originally created the packet);
2. PATHSFOLLOWPOLICY: Explicit policies should determine the paths that packets follow.[1]

For instance, NATs and load balancers dynamically rewrite packet headers, thus violating ORIGINBINDING. Similarly, dynamic middlebox actions, such as responses served from a proxy's cache, may violate PATHSFOLLOWPOLICY. (We elaborate on these examples in §2.)

Some might argue that middleboxes can be eliminated (e.g., [26, 54]), or that their functions can be equivalently provided in SDN switches (e.g., [41]), or that we should replace proprietary boxes by open solutions (e.g, [20, 52]). While these are valuable approaches, practical technological and business concerns make them untenable, at least for the foreseeable future. First, there is no immediate roadmap for SDN switches to support complex stateful processing. Second, enterprises already have a significant deployed infrastructure that is unlikely to go away. Furthermore, these solutions do not fundamentally address ORIGINBINDING and PATHSFOLLOWPOLICY; they merely shift the burden elsewhere.

We take a pragmatic stance that we should attempt to integrate middleboxes into the SDN fold as "cleanly" as possible. Thus, our focus in this paper is to systematically (re-)enforce the ORIGINBINDING and PATHSFOLLOWPOLICY tenets, even in the presence of dynamic middlebox actions. We identify *flow tracking* as the key to policy enforcement.[2] That is, we need to reliably associate additional contextual information with a traffic flow as it traverses the network, even if packet headers and

[1] A third SDN tenet, HIGHLEVELNAMES, states that network policies should be expressed in terms of high-level names. We do not address it in this work, mostly to retain backwards compatibility with current middlebox configuration APIs. We believe that HIGHLEVELNAMES can naturally follow once we restore the ORIGINBINDING property.

[2] We use the term "flow" in a general sense, not necessarily to refer to an IP 5-tuple.

contents are modified. This helps determine the packet's true endpoints rather than rewritten versions (e.g., as with load balancers), and to provide hints about the packet's provenance (e.g., a cached response).

Based on this insight, we extend the SDN paradigm with the *FlowTags* architecture. Because middleboxes are in the best (and possibly the only) position to provide the relevant contextual information, FlowTags envisions simple extensions to middleboxes to add *tags*, carried in packet headers. SDN switches use the tags as part of their flow matching logic for their forwarding operations. Downstream middleboxes use the tags as part of their packet processing workflows. We retain existing SDN switch interfaces and explicitly decouple middleboxes and switches, allowing the respective vendors to innovate independently.

Deploying FlowTags thus has two prerequisites: (P1) adequate header bits with SDN switch support to match on tags and (P2) extensions to middlebox software. We argue that (P1) is possible in IPv4; quite straightforward in IPv6; and will become easier with recent OpenFlow standards that allow flexible matching [9] and new switch hardware designs [23]. As we show in §6, (P2) requires minor code changes to middlebox software.

Contributions and roadmap: While some of these arguments appeared in an earlier position paper [28], several practical questions remained w.r.t. (1) policy abstractions to capture the dynamic middlebox scenarios; (2) concrete controller design; (3) the viability of extending middleboxes to support FlowTags; and (4) the practical performance and benefits of FlowTags.

Our specific contributions in this paper are:

- We describe controller–middlebox interfaces to configure tagging capabilities (§4), and new controller policy abstractions and rule-generation mechanisms to explicitly configure the tagging logic (§5).
- We show that it is possible to extend five software middleboxes to support FlowTags, each requiring less than 75 lines of custom code in addition to a common 250-line library. (To put these numbers in context, the middleboxes we have modified have between 2K to over 300K lines of code.) (§6).
- We demonstrate that FlowTags enables new verification and network diagnosis methods that are otherwise hindered due to middlebox actions (§7).
- We show that FlowTags adds little overhead over SDN mechanisms, and that the controller is scalable (§8).

§9 discusses related work; §10 sketches future work.

2 Background and Motivation

In this section we present a few examples that highlight how middlebox actions violate OriginBinding and PathsFollowPolicy, thus making it difficult to enforce network-wide policies and affecting other management tasks such as diagnosis. We also discuss why some seemingly natural strawman solutions fail to address our requirements.

2.1 Motivating Scenarios

Attribution problems: Figure 1 shows two middleboxes: a NAT that translates private IPs to public IPs and a firewall configured to block hosts H_1 and H_3 from accessing specific public IPs. Ideally, we want administrators to configure firewall policies in terms of original source IPs. Unfortunately, we do not know the private-public IP mappings that the NAT chooses dynamically; i.e., the OriginBinding tenet is violated. Further, if only traffic from H_1 and H_3 should be directed to the firewall and the rest is allowed to pass through, an SDN controller cannot install the correct forwarding rules at switches S_1/S_2, as the NAT changes the packet headers; i.e., PathsFollowPolicy no longer holds.

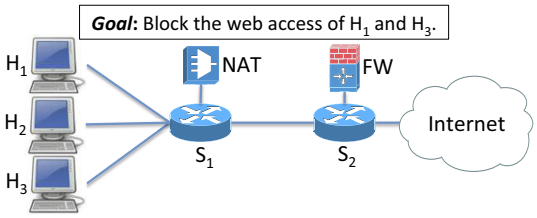


Figure 1: Applying the blocking policy is challenging, as the NAT hides the true packet sources.

Network diagnosis: In Figure 2, suppose the users of hosts H_1 and H_3 complain about high network latency. In order to debug and resolve this problem (e.g., determine if the middleboxes need to be scaled up [30]), the network administrator may use a combination of host-level (e.g., X-Trace [29]) and network-level (e.g., [3]) logs to break down the delay for each request into per-segment components as shown. Because OriginBinding does not hold, it is difficult to correlate the logs to track flows [50, 56].

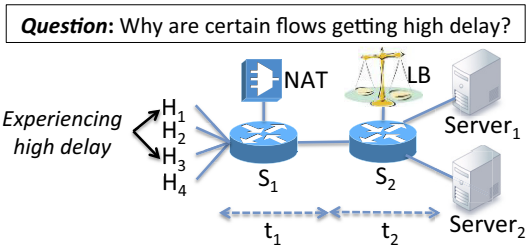


Figure 2: Middlebox modifications make it difficult to consistently correlate network logs for diagnosis.

Data-dependent policies: In Figure 3, the light IPS checks simple features (e.g., headers); we want to route suspicious packets to the heavy IPS, which runs deeper

analysis to determine if the packet is malicious. Such a triggered architecture is quite common; e.g., rerouting suspicious packets to dedicated packet scrubbers [12]. The problem here is that ensuring PATHSFOLLOWPOLICY depends on the *processing history*; i.e., did the light IPS flag a packet as suspicious? However, each switch and middlebox can only make processing or forwarding decisions with its link-local view.

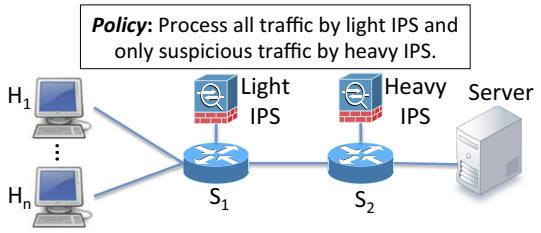


Figure 3: S_2 cannot decide if an incoming packet should be sent to the heavy IPS or the server.

Policy violations due to middlebox actions: Figure 4 shows a proxy used in conjunction with an access control device (ACL). Suppose we want to block H_2's access to `xyz.com`. However, H_2 may bypass the policy by accessing cached versions of `xyz.com`, thus evading the ACL. The problem, therefore, is that middlebox actions may violate PATHSFOLLOWPOLICY by introducing unforeseen paths. In this case, we may need to explicitly route the cached responses to the ACL device as well.

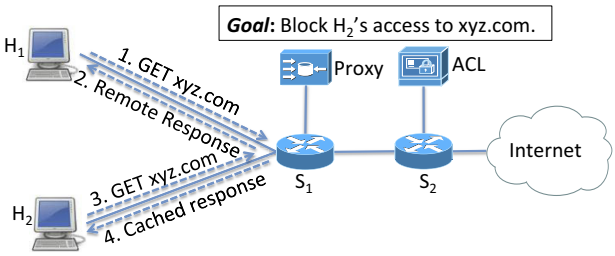


Figure 4: Lack of visibility into the middlebox context (i.e., cache hit/miss in this example) makes policy enforcement challenging.

2.2 Strawman Solutions

Next, we highlight why some seemingly natural strawman solutions fail to address the above problems. Due to space constraints, we discuss only a few salient candidates; Table 1 summarizes their effectiveness in the previously-presented examples.

Placement constraints: One way to ensure ORIGINBINDING/PATHSFOLLOWPOLICY is to "hardwire" the policy into the topology. In Figure 1, we could place the firewall before the NAT. Similarly, for Figure 3 we could connect the light IPS and the heavy IPS to S_1, and configure the light IPS to emit legitimate/suspicious packets on different output ports. S_1 can then use the incoming port to determine if the packet should be sent to the heavy IPS. This coupling between policy and topology, however, violates the SDN philosophy of decoupling the control logic from the data plane. Furthermore, this restricts flexibility to reroute under failures, load balance across middleboxes, or customize policies for different workloads [50].

Strawman solution	Attribution (Figure 1)	Diagnosis (Figure 2)	Data-dependent policy (Figure 3)	Policy violations (Figure 4)
Placement	Yes, if we alter policy chains	No	If both IPSes are on S_1 & Light IPS has 2 ports	Yes
Tunneling (e.g, [38, 36])	No	No	Need IPS support	No
Consolidation (e.g., [52])	Not with separate modules	No	Maybe, if shim is aware	
Correlation (e.g., [49])	Not accurate, lack of ground truth, and high overhead			

Table 1: Analyzing strawman solutions vs. the motivating examples in §2.1.

Tunneling: Another option to ensure PATHSFOLLOWPOLICY is to set up tunneling rules, for example, using MPLS or virtual circuit identifiers (VCIs). For instance, we could tunnel packets from the "suspicious" output of the light IPS to the heavy IPS in Figure 3. (Note that this requires middleboxes to support tunnels.) Such topology/tunneling solutions may work for simple examples, but they quickly break for more complex policies; e.g., if there are more outputs from the light IPS. Note that even by combining placement+tunneling, we cannot solve the diagnosis problem in Figure 2, as it does not provide ORIGINBINDING.

Middlebox consolidation: At first glance, it may seem that we can ensure PATHSFOLLOWPOLICY by running *all* middlebox functions on a consolidated platform [20, 52]. While consolidation provides other benefits (e.g., reduced hardware costs), it has several limitations. First, it requires a significant network infrastructure change. Second, it merely shifts the burden of PATHSFOLLOWPOLICY to the internal routing "shim" that routes packets between the modules. Finally, if the individual modules are provided by different vendors, diagnosis and attribution is hard, as this shim cannot ensure ORIGINBINDING.

Flow correlation: Prior work attempts to heuristically correlate the payloads of the traffic entering and leaving middleboxes to correlate flows [49]. However, this approach can result in missed/false matches too of-

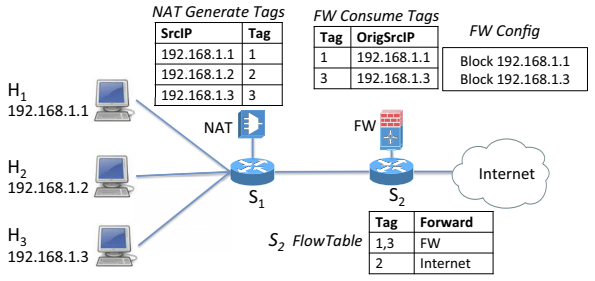


Figure 5: Figure 1 augmented to illustrate how tags can solve the attribution problem.

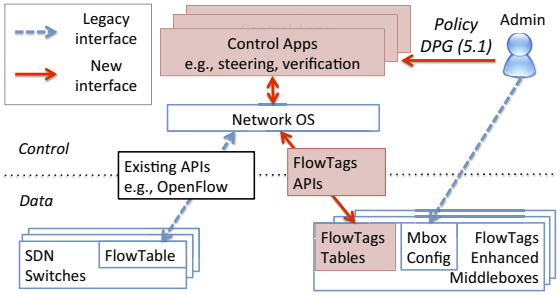


Figure 6: Interfaces between different components in the FlowTags architecture.

ten to be useful for security applications [49]. Also, such "reverse engineering" approaches fundamentally lack ground truth. Finally, this process has high overhead, as multiple packets per flow need to be processed at the controller in a stateful manner (e.g., when reassembling packet payloads).

As Table 1 shows, none of these strawman solutions can address all of the motivating scenarios. In some sense, each approach partially addresses some *symptoms* of the violations of ORIGINBINDING and PATHSFOLLOWPOLICY, but does not address the *cause* of the problem. Thus, despite the complexity they entail in terms of topology hacks, routing, and middlebox and controller upgrades, they have limited applicability and have fundamental correctness limitations.

3 FlowTags Overview

As we saw in the previous section, violating the ORIGINBINDING and PATHSFOLLOWPOLICY tenets makes it difficult to correctly implement several network management tasks. To address this problem, we propose the FlowTags architecture. In this section, we highlight the main intuition behind FlowTags, and then we show how FlowTags extends the SDN paradigm.

3.1 Intuition

FlowTags takes a first-principles approach to ensure that ORIGINBINDING and PATHSFOLLOWPOLICY hold even in the presence of middlebox actions. Since the middleboxes are in the best (and sometimes the only) position to provide the relevant context (e.g., a proxy's cache hit/miss state or a NAT's public-private IP mappings), we argue that middleboxes need to be extended in order to be integrated into SDN frameworks.

Conceptually, middleboxes add tags to outgoing packets. These tags provide the missing bindings to ensure ORIGINBINDING and the necessary processing context to ensure PATHSFOLLOWPOLICY. The tags are then used in the data plane configuration of OpenFlow switches and other downstream middleboxes.

To explain this high-level idea, let us revisit the example in Figure 1 and extend it with the relevant tags and actions as shown in Figure 5. We have three hosts $H_1 - H_3$ in an RFC1918 private address space; the administrator wants to block the Internet access for H_1 and H_3, and allow H_2's packets to pass through without going to the firewall. The controller (not shown) configures the NAT to associate outgoing packets from H_1, H_2, and H_3 with the tags 1, 2, and 3, respectively, and adds these to prespecified header fields. (See §5.3). The controller configures the firewall so that it can *decode* the tags to map the observed IP addresses (i.e., in "public" address space using RFC1918 terminology) to the original hosts, thus meeting the ORIGINBINDING requirement. Similarly, the controller configures the switches to allow packets with tag 2 to pass through without going to the firewall, thus meeting the PATHSFOLLOWPOLICY requirement. As an added benefit, the administrator can configure firewall rules w.r.t. the original host IP addresses, without needing to worry about the NAT-induced modifications.

This example highlights three key aspects of FlowTags. First, middleboxes (e.g., the NAT) are *generators* of tags (as instructed by the controller). The packet-processing actions of a FlowTags-enhanced middlebox might entail adding the relevant tags into the packet header. This is crucial for both ORIGINBINDING and PATHSFOLLOWPOLICY, depending on the middlebox.

Second, other middleboxes (e.g., the firewall) are *consumers* of tags, and their processing actions need to decode the tags. This is necessary for ORIGINBINDING. (In this simple example, each middlebox only generates or only consumes tags. In general, however, a given middlebox could both consume and generate tags.)

Third, SDN-capable switches in the network use the tags as part of their forwarding actions, in order to route packets according to the controller's intended policy, ensuring PATHSFOLLOWPOLICY holds.

Note that the FlowTags semantics apply in the context of a *single administrative domain*. In the simple case, we set tag bits to NULL on packets exiting the domain.[3]

[3]More generally, if we have a domain hierarchy (e.g., "CS dept" and "Physics dept" and "Univ" at a higher level), each sub-domain's egress switch can rewrite the tag to only capture higher-level semantics (e.g, "CS" rather than "CS host A"), without revealing internal details.

This alleviates concerns that the tag bits may accidentally leak proprietary topology or policy information. When incoming packets arrive at an external interface, the gateway sets the tag bits appropriately (e.g., to ensure stateful middlebox traversal) before forwarding the packet into the domain.

3.2 Architecture and Interfaces

Next, we describe the interfaces between the controller, middleboxes, switches, and the network administrator in a FlowTags-enhanced SDN architecture.

Current SDN standards (e.g., OpenFlow [45]) define the APIs between the controller and switches. As shown in Figure 6, FlowTags adds three extensions to today's SDN approach:

1. **FlowTags APIs** between the controller and FlowTags-enhanced middleboxes, to programmatically configure their tag generation and consumption logic (§4).
2. **FlowTags controller modules** that configure the tagging-related generation/consumption behavior of the middleboxes, and the tag-related forwarding actions of SDN switches (§5).
3. **FlowTags-enhanced middleboxes** consume an incoming packet's tags when processing the packet and generate new tags based on the context (§6).

FlowTags requires neither new capabilities from SDN switches, nor any direct interactions between middleboxes and switches. Switches continue to use traditional SDN APIs such as OpenFlow. The only interaction between switches and middleboxes is indirect, via tags embedded inside the packet headers. We take this approach for two reasons: (1) to allow switch and middlebox designs and their APIs to innovate independently; and (2) to retain compatibility with existing SDN standards (e.g., OpenFlow). Embedding tags in the headers avoids the need for each switch and middlebox to communicate with the controller on every packet when making their forwarding and processing decisions.

We retain existing configuration interfaces for customizing middlebox actions; e.g., vendor-specific languages or APIs to configure firewall/IDS rules. The advantage of FlowTags is that administrators can configure these rules without having to worry about the impact of intermediate middleboxes. For example, in the first scenario of §2.1, FlowTags allows the operator to specify firewall rules with respect to the original source IPs. This provides a cleaner mechanism, as the administrator does not need to reason about the space of possible header values a middlebox may observe.[4]

[4]Going forward, we want to configure the middlebox rules to ensure the HIGHLEVELNAMES as well [24].

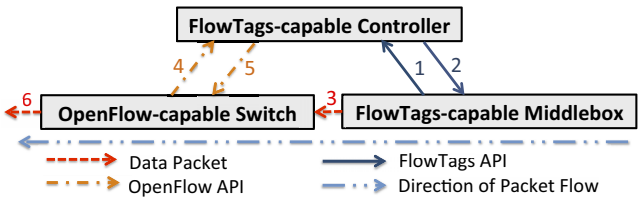


Figure 7: Packet processing walkthrough for tag generation: 1. Tag Generation Query, 2. Tag Generation Response, 3. Data Packet, 4. Packet-in Message, 5. Modify Flow Entry Message, 6. Data Packet (to next on-path switch).

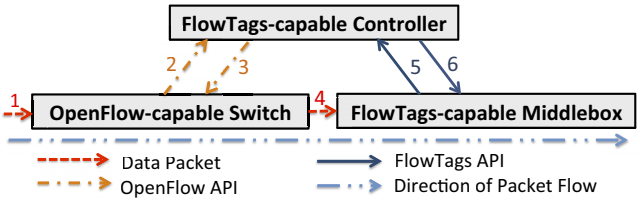


Figure 8: Packet processing walkthrough for tag consumption: 1. Data Packet, 2. Packet-in Message, 3. Modify Flow Entry Message, 4. Data Packet, 5. Tag Consumption Query, 6. Tag Consumption.

4 FlowTags APIs and Operation

Next, we walk through how a packet is processed in a FlowTags-enhanced network, and describe the main FlowTags APIs. For ease of presentation, we assume each middlebox is connected to the rest of the network via a switch. (FlowTags also works in a topology with middleboxes directly chained together.) We restrict our description to a *reactive* controller that responds to incoming packets, but proactive controllers are also possible.

For brevity, we only discuss the APIs pertaining to packet processing. Analogous to the OpenFlow configuration APIs, we envision functions to obtain and set FlowTags capabilities in middleboxes; e.g., which header fields are used to encode the tag values (§5.3).

In general, the same middlebox can be both a *generator* and a *consumer* of tags. For clarity, we focus on these two roles separately. We assume that a packet, before it reaches any middlebox, starts with a NULL tag.

Middlebox tag generation, Figure 7: Before the middlebox outputs a processed (and possibly modified) packet, it sends the `FT_GENERATE_QRY` message to the controller requesting a tag value to be added to the packet (Step 1). As part of this query the middlebox provides the relevant packet processing context: e.g., a proxy tells the controller if this is a cached response; an IPS provides the processing verdict. The controller provides a tag value via the `FT_GENERATE_RSP` response (Step 2). (We defer tag semantics to the next section.)

Middlebox tag consumption, Figure 8: When a mid-

dlebox receives a tag-carrying packet, it needs to "decode" this tag; e.g., an IDS needs the original IP 5-tuple for scan detection. The middlebox sends the `FT_CONSUME_QRY` message (Step 5) to the controller, which then provides the necessary decoding rule for mapping the tag via the `FT_CONSUME_RSP` message (Step 6).

Switch actions: In Figure 7, when the switch receives a packet from the middlebox with a tag (Step 3), it queries the controller with the `OFPT_PACKET_IN` message (Step 4), and the controller provides a new flow table entry (Step 5). This determines the forwarding action; e.g., whether this packet should be routed toward the heavy IPS in Figure 3. Similarly, when the switch receives a packet in Figure 8 (Step 1), it requests a forwarding entry and the controller uses the tag to decide if this packet needs to be forwarded to the middlebox.

Most types of middleboxes operate at an IP flow or session granularity, and their dynamic modifications typically use a consistent header mapping for all packets of a flow. Thus, analogous to OpenFlow, a middlebox needs to send `FT_CONSUME_QRY` and `FT_GENERATE_QRY` only *once per flow*. The middlebox stores the per-flow tag rules locally, and subsequent packets in the same flow can reuse the cached tag rules.

5 FlowTags Controller

In this section, we discuss how a FlowTags-enhanced SDN controller can assign tags and tags-related "rules" to middleboxes and switches. We begin with a policy abstraction (§5.1) that informs the semantics that tags need to express (§5.2). Then, we discuss techniques to translate this solution into practical encodings (§5.3–§5.4). Finally, we outline the controller's implementation (§5.5).

5.1 Dynamic Policy Graph

The input to the FlowTags controller is the *policy* that the administrator wants to enforce w.r.t. middlebox actions (Figure 6). Prior work on middlebox policy focuses on a *static policy graph* that maps a given *traffic class* (e.g., as defined by network locations and flow header fields) to a *chain* of middleboxes [30, 38, 49]. For instance, the administrator may specify that all outgoing web traffic from location A to location B must go, in order, through a firewall, an IDS, and a proxy. However, this static abstraction fails to capture the ORIGINBINDING and PATHSFOLLOWPOLICY requirements in the presence of traffic-dependent and dynamic middlebox actions. Thus, we propose the *dynamic policy graph* (or *DPG*) abstraction.

A DPG is a directed graph with two types of nodes: (1) *In* and *Out* nodes, and (2) *logical middlebox* nodes. *In* and *Out* nodes represent network ingresses and egresses (including "drop" nodes). Each logical middlebox represents a type of middlebox function, such as "firewall." (For clarity, we restrict our discussion to "atomic" middlebox functions; a multi-function box will be represented using multiple nodes.) Each logical middlebox node is given a configuration that governs its processing behavior for each traffic class (e.g., firewall rulesets or IDS signatures). As discussed earlier, administrators specify middlebox configurations in terms of the unmodified traffic entering the DPG, without worrying about intermediate transformations.

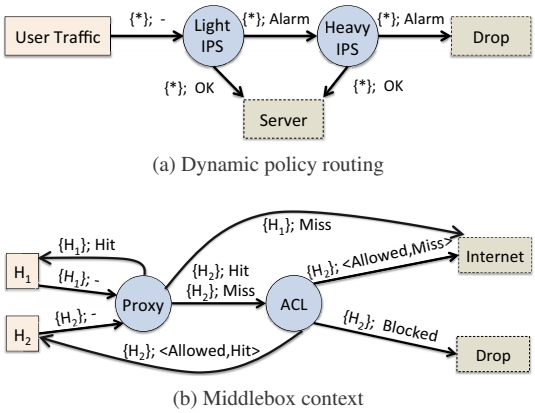


Figure 9: The DPGs for the examples in Figures 3 and 4. Rectangles with solid lines denote "Ingress" nodes and with dotted lines denote "Egress" nodes. Circles denote logical middlebox functions. Each edge is annotated with a $\{Class\};Context$ denoting the traffic class and the processing context(s). All traffic is initialized as "$\{$null$\}$;-".

Each edge in the DPG is annotated with the *condition* $m \rightarrow m'$ under which a packet needs to be steered from node m to node m'. This condition is defined in terms of (1) the traffic class, and (2) the *processing context* of node m, if applicable. Figure 9 shows two DPG snippets:

- **Data-dependent policies:** Figure 9a revisits the example in Figure 3. Here, we want all traffic to be first processed by the light IPS. If the light IPS flags a packet as suspicious, then it should be sent to the heavy IPS. In this case, the edge connecting the light IPS to the heavy IPS is labeled "**, Alarm*", where * denotes the class of "any traffic," and *Alarm* provides the relevant processing history from the light IPS.
- **Capturing effects of middlebox actions:** Figure 9b revisits the example in Figure 4, where we want to apply an ACL only on host H_2's web requests. For correct policy enforcement, the ACL must be applied to both cached and uncached responses. Thus, both "H_2, *Hit*" and "H_2, *Miss*" need to be on the Proxy-to-ACL edge. (For ease of visualization, we do not show the policies applied to the responses coming from the Internet.)

We currently assume that the administrator creates the

DPG based on domain knowledge. We discuss a mechanism to help administrators to generate DPGs in §10.

5.2 From DPG to Tag Semantics

The DPG representation helps us reason about the semantics we need to capture via tags to ensure ORIGINBINDING and PATHSFOLLOWPOLICY.

Restoring ORIGINBINDING: We can ensure ORIGINBINDING if we are always able to map a packet to its original IP 5-tuple *OrigHdr* as it traverses a DPG. Note that having *OrigHdr* is a *sufficient* condition for ORIGINBINDING: given the *OrigHdr*, any downstream middlebox or switch can conceptually implement the action intended by a DPG. In some cases, such as per-flow diagnosis (Figure 2), mapping a packet to the *OrigHdr* might be necessary. In other examples, a coarser identifier may be enough; e.g., just srcIP in Figure 1.

Restoring PATHSFOLLOWPOLICY: To ensure PATHSFOLLOWPOLICY, we essentially need to capture the edge condition $m \rightarrow m'$. Recall that this condition depends on (1) the traffic class and (2) the middlebox context, denoted by C, from logical middlebox m (and possibly previous logical middleboxes). Given that the *OrigHdr* for ORIGINBINDING provides the necessary context to determine the traffic class, the only additional required information on $m \rightarrow m'$ is the context C.

If we assume (until §5.3) no constraints on the tag identifier space, we can think of the controller as assigning a globally unique tag T to each "located packet"; i.e., a packet along with the edge on the DPG [51]. The controller maps the tag of each located packet to the information necessary for ORIGINBINDING and PATHSFOLLOWPOLICY: $T \rightarrow \langle OrigHdr, C \rangle$. Here, the *OrigHdr* represents the original IP 5-tuple of this located packet when it first enters the network (i.e., before any middlebox modifications) and C captures the processing context of this located packet.

In the context of tag consumption from §4, `FT_CONSUME_QRY` and `FT_CONSUME_RSP` essentially "dereference" tag T to obtain the *OrigHdr*. The middlebox can apply its processing logic based on the *OrigHdr*; i.e., satisfying ORIGINBINDING.

For tag generation at logical middlebox m, `FT_GENERATE_QRY` provides as input to the controller: (1) the necessary middlebox context to determine which C will apply, and (2) the tag T of the incoming packet that triggered this new packet to be generated. The controller creates a new tag T' entry for this new located packet and populates the entry $T' \rightarrow \langle OrigHdr', C \rangle$ for this new tag as follows. First, it uses *OrigHdr* (for the input tag T) to determine the value $OrigHdr'$ for T'. In many cases (e.g., NAT), this is a simple copy. In some cases (e.g., proxy response), the association has to reverse the src/dst mappings in *OrigHdr*. Second, it associates the new tag T' with context C. The controller instructs the middlebox, via `FT_GENERATE_RSP`, to add T' to the packet header. Because T' is mapped to C, it supports enforcement of PATHSFOLLOWPOLICY.

5.3 Encoding Tags in Headers

In practice, we need to embed the tag value in a finite number of packet-header bits. IPv6 has a 20-bit *Flow Label* field, which seems ideal for this use (thus answering the question "how should we use the flow-label field?" [19]). For our current IPv4 prototype and testbed, we used the 6-bit DS field (part of the 8-bit ToS), which sufficed for our scenarios. To deploy FlowTags on large-scale IPv4 networks, we would need to borrow bits from fields that are not otherwise used. For example, if VLANs are not used, we can use the 12-bit VLAN Identifier field. Or, if all traffic sets the DF (Don't Fragment) IP Flag, which is typical because of Path MTU Discovery, the 16-bit IP_ID field is available.[5]

Next, we discuss how to use these bits as efficiently as possible; §8 reports on some analysis of how many bits might be needed in practice.

As discussed earlier, tags restore ORIGINBINDING and PATHSFOLLOWPOLICY. Conceptually, we need to be able to distinguish every located packet—i.e., the combination of all flows and all possible paths in the DPG. Thus, a simple upper bound on the number of bits in each packet to distinguish between $|Flows|$ flows on $|DPGPaths|$ processing paths is: $\log_2 |Flows| + \log_2 |DPGPaths|$, where *Flows* is the set of IP flows (for ORIGINBINDING), and *DPGPaths* is the set of possible paths a packet could traverse in DPG (for PATHSFOLLOWPOLICY). However, this grows log-linearly in the number of flows over time and the number of paths (which could be exponential w.r.t. the graph size).

This motivates optimizations to reduce the number of header bits necessary, which could include:

- **Coarser tags:** For many middlebox management tasks, it may suffice to use a tag to identify the logical traffic class (e.g., "CS Dept User") and the local middlebox context (e.g., 1 bit for cache hit or miss or 1 bit for "suspicious"), rather than individual IP flows.
- **Temporal reuse:** We can reuse the tag assigned to a flow after the flow expires; we can detect expiration via explicit flow termination, or via timeouts [3, 45]. The controller tracks active tags and finds an unused value for each new tag.
- **Spatial reuse:** To address ORIGINBINDING, we only need to ensure that the new tag does not conflict with tags already assigned to currently active flows at the middlebox to which this packet is destined. For PATHSFOLLOWPOLICY, we need to: (1) capture the

[5] IP_ID isn't part of the current OpenFlow spec; but it can be supported with support for flexible match options [9, 23].

most recent edge on the DPG rather than the entire path (i.e., reducing from $|DPGPaths|$ to the node degree); and (2) ensure that the switches on the path have no ambiguity in the forwarding decision w.r.t. other active flows.

5.4 Putting it Together

Our current design is a *reactive* controller that responds to `OFPT_PACKET_IN`, `FT_CONSUME_QRY`, and `FT_GENERATE_QRY` events from the switches and the middleboxes.

Initialization: Given an input DPG, we generate a data plane realization *DPGImpl*; i.e., for each logical middlebox *m*, we need to identify candidate *physical middlebox instances*, and for each edge in DPG, we find a switch-level path between corresponding physical middleboxes. This translation should also take into account considerations such as load balancing across middleboxes and resource constraints (e.g., switch TCAM and link capacity). While FlowTags is agnostic to the specific realization, we currently use SIMPLE [49], mostly because of our familiarity with the system. (This procedure only needs to run when the *DPG* itself changes or in case of a network topology change. It does not run for each flow arrival.)

Middlebox event handlers: For each physical middlebox instance PM_i, the controller maintains two FlowTags tables: $CtrlInTagsTable_i$ and the $CtrlOutTagsTable_i$. The $CtrlInTagsTable_i$ maintains the tags corresponding to all *incoming* active flows into this middlebox using entries $\{T \rightarrow OrigHdr\}$. The $CtrlOutTagsTable_i$ tracks the tags that need to be assigned to *outgoing* flows and maintains a table of entries $\{\langle T, C\rangle \rightarrow T'\}$, where T is the tag for the incoming packet, C captures the relevant middlebox context for this flow (e.g., cache hit/miss), and T' is the output tag to be added. At bootstrap time, these structures are initialized to be empty.

The HANDLE_FT_CONSUME_QRY handler looks up the entry for tag T in the $CtrlInTagsTable_i$ and sends the mapping to PM_i. As we will see in the next section, middleboxes keep these entries in a FlowTable-like structure, to avoid look ups for subsequent packets. The HANDLE_FT_GENERATE_QRY handler is slightly more involved, as it needs the relevant middlebox context C. Given C, the *DPG*, and the *DPGImpl*, the controller identifies the next hop physical middlebox $PM_{i'}$ for this packet. It also determines a non-conflicting T' using the logic from §5.3.

Switch and flow expiry handlers: The handlers for `OFPT_PACKET_IN` are similar to traditional OpenFlow handlers; the only exception is that we use the incoming tag to determine the forwarding entry. When a flow expires, we trace the path this flow took and, for each PM_i, delete the entries in $CtrlInTagsTable_i$ and $CtrlOutTagsTable_i$, so that these tags can be repurposed.

5.5 Implementation

We implement the FlowTags controller as a `POX` module [10]. The $CtrlInTagsTable_i$ and $CtrlOutTagsTable_i$ are implemented as hash-maps. For memory efficiency and fast look up of available tags, we maintain an auxiliary bitvector of the active tags for each middlebox and switch interface; e.g., if we have 16-bit tags, we maintain a 2^{16} bit vector and choose the first available bit, using a log-time algorithm [22]. We also implement simple optimizations to precompute shortest paths for every pair of physical middleboxes.

6 FlowTags-enhanced Middleboxes

As discussed in the previous sections, FlowTags requires middlebox support. We begin by discussing two candidate design choices for extending a middlebox to support FlowTags. Then, we describe the conceptual operation of a *FlowTags-enhanced* middlebox. We conclude this section by summarizing our experiences in extending five software middleboxes.

6.1 Extending Middleboxes

We consider two possible ways to extend middlebox software to support FlowTags:

- **Module modification:** The first option is to modify specific internal functions of the middlebox to consume and generate the tags. For instance, consider an IDS with the scan detection module. Module modification entails patching this scan detection logic with hooks to translate the incoming packet headers+tag to the *OrigHdr* and to rewrite the scan detection logic to use *OrigHdr*. Similarly, for generation, we modify the output modules to provide the relevant context as part of the `FT_GENERATE_QRY`.
- **Packet rewriting:** A second option is to add a lightweight shim module that *interposes* on the incoming and outgoing packets to rewrite the packet headers. For consumption, this means we modify the packet headers so that the middlebox only sees a packet with the true *OrigHdr*. For generation, this means that the middlebox proceeds as-is and then the shim adds the tag before the packet is sent out.

In both cases, the administrator sets up the middlebox configuration (e.g., IDS rules) as if there were no packet modifications induced by the upstream middleboxes because FlowTags preserves the binding between the packet's modified header and the *OrigHdr*.

For consumption, we prefer packet rewriting because it generalizes to the case where each middlebox has multiple "consumer" modules; e.g., an IDS may apply scan detection and signature-based rules. For generation,

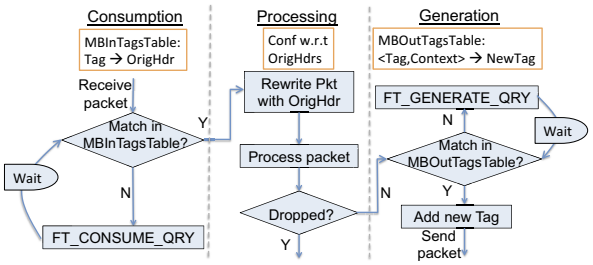


Figure 10: We choose a hybrid design where the "consumption" side uses the packet rewriting and the "generation" uses the module modification approach.

however, packet rewriting may not be sufficient, as the shim may not have the necessary visibility into the middlebox context; e.g., in the proxy cache hit/miss case. Thus, we use module modification in this case.

End-to-end view: Figure 10 shows a simplified view of a FlowTags-enhanced middlebox. In general, *consumption* precedes *generation*. The reason is that the packet's current tag can affect the specific middlebox code paths, and thus impacts the eventual outgoing tags.

Mirroring the controller's $CtrlInTagsTable_i$ and $CtrlOutTagsTable_i$, each physical middlebox i maintains the tag rules in the $MBInTagsTable_i$ and $MBOutTagsTable_i$. When a packet arrives, it first checks if the tag value in the packet already matches an existing tag-mapping rule in $MBInTagsTable_i$. If there is a match, we rewrite packet headers (see above) so that the processing modules act as if they were operating on *OrigHdr*. If there is a $MBInTagsTable_i$ miss, the middlebox sends a `FT_CONSUME_QRY`, buffers the packet locally, and waits for the controller's response.

Note that the tags are logically propagated through the processing contexts (not shown for clarity). For example, most middleboxes follow a connection-oriented model with a data structure maintaining per-flow or per-connection state; we augment this structure to propagate the tag value. Thus, we can causally relate an outgoing packet (e.g., a NAT-ed packet or a proxy cached response) to an incoming packet.

When a specific middlebox function or module is about to send a packet forward, it checks the $MBOutTagsTable_i$ to add the outgoing tag value. If there is a miss, it sends the `FT_GENERATE_QRY`, providing the necessary module-specific context and the tag (from the connection data structure) for the incoming packet that caused this outgoing packet to be generated.

6.2 Experiences in Extending Middleboxes

Given this high-level view, next we describe our experiences in modifying five software middleboxes that span a broad spectrum of management functions. (Our choice was admittedly constrained by the availability of the middlebox source code.) Table 2 summarizes these middleboxes and the modifications necessary.

Name, Role	Modified / Total LOC	Key Modules	Data Structures
Squid [14], Proxy	75 / 216K	Client and Server Side Connection, Forward, Cache Lookup	Request Table
Snort [13], IDS/IPS	45 / 336K	Decode, Detect, Encode	Verdict
Balance [1], Load Balancer	60 / 2K	Client and Server Connections	n/a
PRADS [11], Monitoring	25 / 15K	Decode	n/a
iptables [6], NAT	55 / 42K	PREROUTING, POSTROUTING	Conn Map

Table 2: Summary of the middleboxes we have added FlowTags support to along with the number of lines of code and the main modules to be updated. We use a common library ($\approx$ 250 lines) that implements routines for communicating to the controller.

Our current approach to extend middleboxes is semi-manual and involved a combination of call graph analysis [7, 17] and traffic injection and logging techniques [2, 4, 5, 15]. Based on these heuristics, we identify the suitable "chokepoints" to add the FlowTags logic. Developing techniques to automatically extend middleboxes is an interesting direction for future work.

- **Squid:** Squid [14] is a popular proxy/cache. We modified the functions in charge of communicating with the client, remote server, and those handling cache lookup. We used the packet modification shim for incoming packets, and applied module modification to handle the possible packet output cases, based on cache hit and miss events.
- **Snort:** Snort [13] is an IDS/IPS that provides many functions—logging, packet inspection, packet filtering, and scan detection. Similar to Squid, we applied the packet rewriting step for tag consumption and module modification for tag generation as follows. When a packet is processed and a "verdict" (e.g., OK vs. alarm) is issued, the tag value is generated based on the type of the event (e.g., outcome of a matched alert rule).
- **Balance:** Balance [1] is a TCP-level load balancer that distributes incoming TCP connections over a given a set of destinations (i.e., servers). In this case, we simply read/write the tag bits in the header fields.
- **PRADS:** PRADS [11] is passive monitor that gathers traffic information and infers what hosts and services exist in the network. Since this is a passive device, we only need the packet rewriting step to restore the (modified) packet's *OrigHdr*.
- **NAT via iptables:** We have registered appropriate tagging functions with iptables [6] hook points, while

Src / Time(s)	DPG path	Notes
H_1 / 0	L-IPS→Internet	–
H_1 / 0.3	L-IPS→Internet	–
H_1 / 0.6	L-IPS→Internet	L-IPS alarm
H_1 / 0.8	L-IPS→H-IPS→Drop	drop

(a) In Figure 3, we configure Snort as the light IPS (L-IPS) to flag hosts sending more than 3 packets/sec and send them to the heavy IPS (H-IPS).

Host / URL	DPG path	Notes
H_1 / Dept	Proxy→Internet	always allow
H_2 / CNN	Proxy→ACL→Internet	miss, allow
H_2 / Dept	Proxy→ACL→Drop	hit, drop
H_1 / CNN	Proxy	hit, allow

(b) In Figure 4, we use Squid as the proxy and Snort as the ACL and block H_2's access to the Dept site.

Figure 11: Request trace snippets for validating the example scenarios in Figure 3 and Figure 4.

it is configured as a source NAT. The goal is to maintain 5-tuple visibility via tagging. We added hooks for tag consumption and tag generation into the PREROUTING and the POSTROUTING chains, which are the input and output checkpoints, respectively.

7 Validation and Use Cases

Next, we describe how we can validate uses of FlowTags. We also discuss how FlowTags can be an enabler for new diagnostic and verification capabilities.

Testing: Checking if a network configuration correctly implements the intended *DPG* is challenging—we need to capture stateful middlebox semantics, reason about timing implications (e.g., cache timeouts), and the impact of dynamic modifications. (Even advanced network testing tools do not capture these effects [39, 57].) Automating this step is outside the scope of this paper, and we use a semi-manual approach for our examples.

Given the DPG, we start from each ingress and enumerate all paths to all "egress" or "drop" nodes. For each path, we manually compose a *request trace* that traverses the required branch points; e.g., will we see a cache hit? Then, we emulate this request trace in our small testbed using Mininet [33]. (See §8 for details.) Since there is no other traffic, we use per-interface logs to verify that packets follow the intended path.

Figure 11 shows an example with one set of request sequences for each scenario in Figures 3 and 4. To emulate Figure 3, we use Snort as the light IPS to flag any host sending more than 3 packets/second as suspicious, and direct such hosts' traffic to the heavy IPS for deep packet inspection (also Snort). Figure 11(a) shows the request trace and the corresponding transitions it triggers.

To emulate Figure 4, we use Squid as the proxy and Snort as the (web)ACL device. We want to route all H_2's web requests through ACL and configure Snort to block

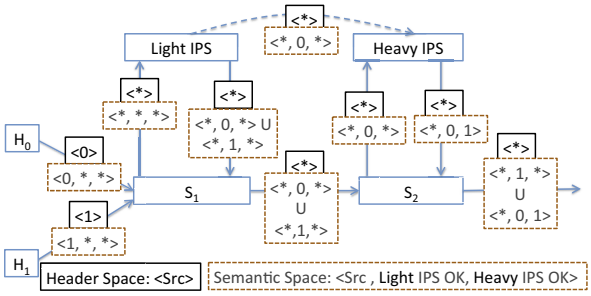


Figure 12: Disconnect between header-space analysis and the intended processing semantics in Figure 3.

H_2's access to the department website. Figure 11(b) shows the sequence of web requests to exercise different *DPG* paths.

We have validated the other possible paths in these examples, and in other scenarios from §2. We do not show these due to space constraints.

FlowTags-enabled diagnosis: We revisit the diagnosis example of Figure 2, with twenty user requests flowing through the NAT and LB. We simulated a simple "red team-blue team" test. One student ("red") synthetically introduced a 100ms delay inside the NAT or LB code for half the flows. The other student ("blue") was responsible for attributing the delays. Because of dynamic header rewriting, the "blue" team could not diagnose delays using packet logs. We repeated the experiment with FlowTags-enhanced middleboxes. In this case, the FlowTags controller assigns a globally unique tag to each request. Thus, the "blue" team could successfully track a flow through the network and identify the bottleneck middlebox using the packet logs at each hop.

Extending verification tools: Verification tools such as Header Space Analysis (HSA) [39] check correctness (e.g., reachability) by modeling a network as the composition of header-processing functions. While this works for traditional switches/routers, it fails for middleboxes, as they operate at higher semantic layers. While a full discussion of such tools is outside the scope of this paper, we present an example illustrating how FlowTags addresses this issue.

Figure 12 extends the example in Figure 3 to show both header-space annotations and DPG-based semantic annotations. Here, a header-space annotation (solid boxes) of $\langle Src \rangle$ describes a packet from *Src*, so $\langle * \rangle$ models a packet from any source. A DPG annotation (dashed boxes) of $\langle Src, L, H \rangle$ describes a packet from *Src* for which *Light IPS* returns L and *Heavy IPS* returns H, so $\langle *, 0, * \rangle$ indicates a packet from any source that is flagged by *Light IPS* as not OK; our policy wants such suspicious packets to go via *Heavy IPS*, while $\langle *, 1, * \rangle$ packets need no further checking.

Recall from §2 that we cannot implement this policy,

in this topology, using existing mechanisms (i.e., without FlowTags). What if we rewired the toplogy by adding the (dashed) link *Light IPS* $\rightarrow$ *Heavy IPS*? Even with this hardwired topology, tools like HSA incorrectly conclude that "all" packets exit the network (the output edge is labeled $\langle * \rangle$), because HSA models middleboxes as "wildcard"-producing blackboxes [39].

FlowTags bridges the gap between "header space," in which verification tools operate, and "semantic space," in which the policy operates. Instead of modeling middleboxes as blackboxes, or reverse-engineering their functions, in FlowTags we treat them as functions operating on tag bits in an (extended) header space. Then, we apply HSA on this extended header space to reason if the network implements the reachability defined by the *DPG*.

8 Performance Evaluation

We frame questions regarding the performance and scalability of FlowTags:

- Q1: What overhead does support for FlowTags add to middlebox processing?
- Q2: Is the FlowTags controller fast and scalable?
- Q3: What is the overhead of FlowTags over traditional SDN?
- Q4: How many tag bits do we need in practice?

Setup: For Q1 and Q2, we run each middlebox and POX controller in isolation on a single core in a 32-core 2.6 Ghz Xeon server with 64 GB RAM. For Q3, we use Mininet [33] on the same server, configured to use 24 cores and 32 GB RAM to model the network switches and hosts. We augment Mininet with middleboxes running as external virtual appliances. Each middlebox runs as a VM configured with 2GB RAM on one core. (We can run at most 28 middlebox instances, due to the maximum number of PCI interfaces that can be plugged in using KVM [8]). We emulate the example topologies from §2, and larger PoP-level ISP topologies from RocketFuel [55]. Our default DPG has an average path length of 3.

Q1 Middlebox overhead: We configure each middlebox to run with the default configuration. We vary the offered load (up to 100 Mbps) and measure the per-packet processing latency. Overall, the overhead was low ($<$1%) and independent of the offered load (not shown). We also analyzed the additional memory and CPU usage using `atop`; it was $< 0.5\%$ across all experiments (not shown).

Q2 Controller scalability: Table 3 shows the running time for the HANDLE_FT_GENERATE_QRY. (This is the most complex FlowTags processing step; other functions take negligible time.) The time is linear as a function of topology size with the baseline algorithms, but almost constant using the optimization to pre-compute reachability information. This implies that a single-thread POX controller can handle $\frac{1}{0.028ms} \approx 35K$ middlebox queries per second (more than three times larger than the peak number of flows per second reported in [24]).

Topology (*#nodes*)	Baseline (*ms*)	Optimized (*ms*)
Abilene (*11*)	0.037	0.024
Geant (*22*)	0.066	0.025
Telstra (*44*)	0.137	0.026
Sprint (*52*)	0.161	0.027
Verizon (*70*)	0.212	0.028
AT&T (*115*)	0.325	0.028

Table 3: Time to run HANDLE_FT_GENERATE_QRY.

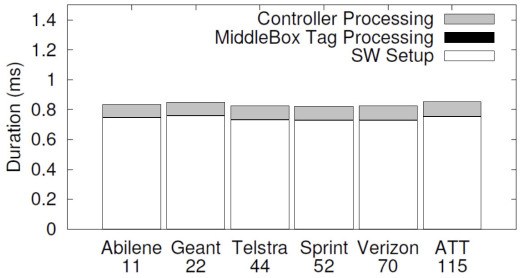


Figure 13: Breakdown of flow processing time in different topologies (annotated with #nodes).

We also varied the DPG complexity along three axes: number of nodes, node degrees, and distance between adjacent DPG nodes in terms of number of switches. With route pre-computation, the controller processing time is independent of the DPG complexity (not shown).

Q3 End-to-end overhead: Figure 13 shows the breakdown of different components of the flow setup time in a FlowTags-enhanced network (i.e., mirroring the steps in Figure 7) for different Rocketfuel topologies. Since our goal is to compare the FlowTags vs. SDN operations, we do not show round-trip times to the controller here, as it is deployment-specific [35].[6] Since all values are close to the average, we do not show error bars. We can see that the FlowTags operations add negligible overhead. In fact, the middlebox tag processing is so small that it might be hard to see in the figure.

We also measure the reduction in TCP throughput a flow experiences in a FlowTags-enhanced network, compared to a traditional SDN network with middleboxes (but without FlowTags). We vary two parameters: (1) controller RTT and (2) the number of packets per flow. As we can see in Table 4, except for very small flows (2 packets), the throughput reduction is $<$4%.

Q4 Number of tag bits: To analyze the benefits of spatial and temporal reuse, we consider the worst case, where we want to diagnose each IP flow. We use packet traces from CAIDA (Chicago and San Jose traces, 2013 [16]) and a flow-level enterprise trace [18]. We sim-

[6] FlowTags adds 1 more RTT per middlebox, but this can be avoided by pre-fetching rules for the switches and middleboxes.

Flow size (#packets)	Reduction in throughput (%)		
	1ms RTT	10ms RTT	20ms RTT
2	12	16.2	22.7
8	2.1	2.8	3.8
32	1.6	2.3	3.0
64	1.5	2.1	2.9

Table 4: Reduction in TCP throughput with FlowTags relative to a pure SDN network.

Configuration (spatial, temporal)	Number of bits	
	CAIDA trace	Enterprise trace
(No spatial, 30 sec)	22	22
(Spatial, 30 sec)	20	20
(Spatial, 10 sec)	18	18
(Spatial, 5 sec)	17	17
(Spatial, 1 sec)	14	14

Table 5: Effect of spatial and temporal reuse of tags.

ulate the traces across the RocketFuel topologies, using a gravity model to map flows to ingress/egress nodes [55].

Table 5 shows the number of bits necessary with different reuse strategies, on the AT&T topology from RocketFuel.[7] The results are similar across other topologies (not shown). We see that temporal reuse offers the most reduction. Spatial reuse helps only a little; this is because with a gravity-model workload, there is typically a "hotspot" with many concurrent flows. To put this in the context of §5.3, using the (Spatial, 1 sec) configuration, tags can fit in the IPv6 FlowLabel, and would fit in the IPv4 IP_ID field.

9 Related Work

We have already discussed several candidate solutions and tools for verification and diagnosis (e.g., [34, 39]). Here, we focus on other classes of related work.

Middlebox policy routing: Prior work has focused on orthogonal aspects of policy enforcement such as middlebox load balancing (e.g., [42, 49]) or compact data plane strategies (e.g,. [27]). While these are candidates for translating the *DPG* to a *DPGImpl* (§5), they do not provide reliable mechanisms to address dynamic middlebox actions.

Middlebox-SDN integration: OpenMB [31] focuses on exposing the internal state (e.g., cache contents and connection state) of middleboxes to enable (virtual) middlebox migration and recovery. This requires significantly more instrumentation and vendor support compared to FlowTags, which only requires externally relevant mappings. Stratos [30] and Slick [21] focus on using SDN to dynamically instantiate new middlebox modules in response to workload changes. The functionality these provide is orthogonal to FlowTags.

[7] Even though the number of flows varies across traces, they require the same number of bits, as the values of $ceil(\log_2(\#flows))$ are the same.

Tag-based solutions: Tagging is widely used to implement Layer2/3 functions, such as MPLS labels or virtual circuit identifiers (VCIs). In the SDN context, tags have been used to avoid loops [49], reduce FlowTable sizes [27], or provide virtualized network views [46]. Tags in FlowTags capture higher-layer semantics to address ORIGINBINDING and PATHSFOLLOWPOLICY. Unlike these Layer2/3 mechanisms where switches are generators and consumers of tags, FlowTags middleboxes generate and consume tags, and switches are consumers.

Tracing and provenance: The idea of flow tracking has parallels in the systems (e.g., tracing [29]), databases (e.g., provenance [58]), and security (e.g., taint tracking [47, 48]) literature. Our specific contribution is to use flow tracking for integrating middleboxes into SDN-capable networks.

10 Conclusions and Future Work

The dynamic, traffic-dependent, and hidden actions of middleboxes make it hard to systematically enforce and verify network-wide policies, and to do network diagnosis. We are not alone in recognizing the significance of this problem—others, including the recent IETF network service chaining working group, mirror several of our concerns [37, 43, 50].

The insight behind FlowTags is that the crux of these problems lies in violation of two key SDN tenets—ORIGINBINDING and PATHSFOLLOWPOLICY—caused by middlebox actions. We argue that middleboxes are in the best (and possibly the only) vantage point to restore these tenets, and make a case for minimally extending middleboxes to provide the necessary context, via tags embedded inside packet headers. We design new SDN APIs and controller modules to configure this tag-related behavior. We showed a scalable proof-of-concept controller, and the viability of adding FlowTags support, with minimal changes, to five canonical middleboxes. We also demonstrated that the overhead of FlowTags is comparable to traditional SDN mechanisms.

We believe that there are three natural directions for future work: automating DPG generation via model refinement techniques (e.g., [25]); automating middlebox extension using appropriate programming-languages techniques; and, performing holistic testing of the network while accounting for switches and middleboxes.

11 Acknowledgments

We would like to thank our shepherd Ben Zhao and the NSDI reviewers for their feedback. This work was supported in part by grant number N00014-13-1-0048 from the Office of Naval Research and by Intel Labs' University Research Office.

References

[1] Balance. http://www.inlab.de/balance.html.

[2] Bit-Twist. http://bittwist.sourceforge.net/.

[3] Cisco systems netflow services export version 9. RFC 3954.

[4] httperf. https://code.google.com/p/httperf/.

[5] iperf. https://code.google.com/p/iperf/.

[6] iptables. http://www.netfilter.org/projects/iptables/.

[7] KCachegrind. http://kcachegrind.sourceforge.net/html/Home.html.

[8] KVM. http://www.linux-kvm.org/page/Main_Page.

[9] Openflow switch specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[10] POX Controller. http://www.noxrepo.org/pox/about-pox/.

[11] PRADS. http://gamelinux.github.io/prads/.

[12] Prolexic. www.prolexic.com.

[13] Snort. http://www.snort.org/.

[14] Squid. http://www.squid-cache.org/.

[15] tcpdump. http://www.tcpdump.org/.

[16] The Cooperative Association for Internet Data Analysis (caida). http://www.caida.org/.

[17] Valgrind. http://www.valgrind.org/.

[18] Vast Challenge. http://vacommunity.org/VAST+Challenge+2013%3A+Mini-Challenge+3.

[19] S. Amante, B. Carpenter, S. Jiang, and J. Rajahalme. Ipv6 flow label update. http://rmv6tf.org/wp-content/uploads/2012/11/rmv6tf-flow-label11.pdf.

[20] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: extensible open middleboxes with commodity servers. In *Proc. ANCS*, 2012.

[21] B. Anwer, T. Benson, N. Feamster, D. Levin, and J. Rexford. A Slick Control Plane for Network Middleboxes. In *Proc. ONS, research track*, 2012.

[22] G. Banga and J. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proc. USENIX ATC*, 1998.

[23] P. Bosshar, 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 *Proc. SIGCOMM*, 2013.

[24] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. SIGCOMM*, 2007.

[25] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV*, 2000.

[26] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: a scalable fault tolerant network manager. In *Proc. NSDI*, 2011.

[27] L. Erran Li, Z. M. Mao, and J. Rexford. CellSDN: Software-defined cellular networks. In *Techinical Report, Princeton University*.

[28] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. FlowTags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *Proc. HotSDN*, 2013.

[29] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: a pervasive network tracing framework. In *Proc. NSDI*, 2007.

[30] A. Gember, 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.

[31] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *Proc. HotNets-XI*, 2012.

[32] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: towards an operating system for networks. In *CCR*, 2008.

[33] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proc. CoNext*, 2012.

[34] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-defined network? In *Proc. HotSDN*, 2012.

[35] B. Heller, R. Sherwood, and N. McKeown. The Controller Placement Problem. In *Proc. HotSDN*, 2012.

[36] X. Jin, L. Erran Li, L. Vanbever, and J. Rexford. Softcell: Scalable and flexible cellular core network architecture. In *Proc. CoNext*, 2013.

[37] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu. Research directions in network service chaining. In *Proc. IEEE SDN4FNS*, 2013.

[38] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. In *Proc. SIGCOMM*, 2008.

[39] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *Proc. NSDI*, 2012.

[40] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[41] J. Lee, J. Tourrilhes, P. Sharma, and S. Banerjee. No More Middlebox: Integrate Processing into Network. In *Proc. SIGCOMM posters*, 2010.

[42] L. Li, V. Liaghat, H. Zhao, M. Hajiaghay, D. Li, G. Wilfong, Y. Yang, and C. Guo. PACE: Policy-Aware Application Cloud Embedding. In *Proc. IEEE INFOCOM*, 2013.

[43] L. MacVittie. Service chaining and unintended consequences. `https://devcentral.f5.com/articles/service-chaining-and-unintended-consequences#.Uvbz0EJdVe9`.

[44] N. McKeown. Mind the Gap: SIGCOMM'12 Keynote. `http://www.youtube.com/watch?v=Ho239zpKMwQ`.

[45] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *CCR*, March 2008.

[46] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proc. NSDI*, 2013.

[47] Y. Mundada, A. Ramachandran, M. B. Tariq, and N. Feamster. Practical Data-Leak Prevention for Legacy Applications in Enterprise Networks. Technical Report `http://hdl.handle.net/1853/36612`, 2011.

[48] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. NDSS*, 2005.

[49] Z. Qazi, C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. SIGCOMM*, 2013.

[50] P. Quinn, J. Guichard, S. Kumar, P. Agarwal, R. Manur, A. Chauhan, N. Leyman, M. Boucadir, C. Jacquenet, M. Smith, N. Yadav, T. Nadeau, K. Gray, B. Mcconnell, and K. Glavin. Network service chaining problem statement. `http://tools.ietf.org/html/draft-quinn-nsc-problem-statement-03`.

[51] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

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

[53] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proc. HotNets*, 2011.

[54] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. SIGCOMM*, 2012.

[55] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of ACM SIGCOMM*, 2002.

[56] W. Wu, G. Wang, A. Akella, and A. Shaikh. Virtual network diagnosis as a service. In *Proc. SoCC*, 2013.

[57] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNext*, 2012.

[58] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proc. SIGMOD*, 2010.