UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Data Center End-host Stack for Predictable Low Latency and Dynamic Network Topologies

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Rishi Kapoor

Committee in charge:

Professor George Porter, Co-Chair
Professor Amin Vahdat, Co-Chair
Professor George C. Papen
Professor Alex C. Snoeren
Professor Geoffrey M. Voelker

2015

The Dissertation of Rishi Kapoor is approved and is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2015

DEDICATION

To my parents, my teachers and my family.

EPIGRAPH

There is more to life than simply increasing its speed.

Mahatma Gandhi

Prediction is very difficult, especially about the future.

Internet quote, often attributed to Niels Bohr

If you can’t measure something, you can’t understand it ... and you can’t improve it.

H. James Harrington

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

As we continue to make technological advancements some (or most) of the work done in this dissertation will be irrelevant down the road but the thing that will remain constant will be the love and support of the people who helped me reach this point. I am grateful to all of them.

I am indebted to my advisors Amin Vahdat and George Porter for their constant guidance, support and encouragement without which this would not have been possible. Amin inspired me to join grad school and also work with him after the grad school. Discussions with Amin have been always been thought provoking and were turning point for many of my projects. Amin leads by example and it would be a dream come true to follow in his foot-steps. George is always around when you need him and still lets you run with the ball. George has helped turn nascent ideas into papers, helped in writing, sat through countless practice talks, and many more. I was extremely fortunate to work with both of them and learn from their perspectives.

I would also like to thanks my committee members Geoffrey Voelker, Alex Snoeren and George Papen for their regular feedback and encouragement. Geoff has been a constant support while carefully nudging me to the right path. Geoffs' and Alexs' attention to detail made my graphs and papers more beautiful.

I am thankful to several of my collaborators and colleagues who made this journey enjoyable and at the same time intellectually simulating. Thank you Mike Conley, Sambit Das, Alex Forensich, He Liu, Feng Lu, Sivasankar Radhakrishnan and Malveeka Tewari. Lonnie wrote the FPGA packet mirroring functionality which was used in the BulletTrains experiment. Alex Forensich implemented the FPGA statistics module for the Scheduling paper. I have benefited from working with several senior PhD students who were patient with a naive and inquisitive grad student. Thank you Harsha Madhyastha, James Anderson, Alex Rasmussen, Mohammad Al-Fares and members of the DCSwitch group.

Several other professors have helped me reach the doorstep of PhD. George Candea gave me an opportunity and early exposure to research that convinced me to apply for graduate school in US. Sudeep Sanyal and Sugata Sanyal pushed me to work towards publications while I was an undergraduate.

I would like to thank all my friends who kept me in high spirits and made this journey memorable. Often it was their enthusiasm that kept me going. I am grateful to my extended family in India for being always supporting and encouraging. My younger brother Rohit graciously stepped in to fill in for my share of responsibilities and ensured that I can pursue my dreams without any other worries. Finally, last but not least, I can't express enough gratitude for my parents sacrifices, love and support. They have inculcated values of hard work, discipline and persistence that was the most useful skill.

Chapter 1, 2, 4, 5, 6, 8 in part, contains material as it appears in Kapoor, Rishi; Porter, George; Tewari, Malveeka; Voelker, Geoffrey M. ; Vahdat, Amin. "Chronos: Predictable Low Latency for Data Center Applications", Proceedings of the ACM Symposium on Cloud Computing (SOCC), San Jose, CA, October 2012 The dissertation author was the primary investigator and author on this paper.

Chapter 1, 2, 3, 8 in part, contains material as it appears in Kapoor, Rishi; Snoeren, Alex C.; Voelker, Geoffrey M. ; Porter, George. "Bullet Trains: A study of NIC burst behavior at microsecond timescales", Proceedings of ACM CoNEXT, Santa Barbara, CA, December 2013. The dissertation author was the primary investigator and author on this paper.

Chapter 1, 7 in part, contains material that has prepared for submission for publication. Liu, He; Kapoor, Rishi; Tewari, Malveeka; Forencich, Alex; Zhang, Sen; Savage, Stefan; Voelker, Geoffrey M.; Papen, George; Snoeren, Alex C.; George, Porter. "Scheduling Circuits in a Packet World". The dissertation author is the second author on this paper.

Chapter 2, 7 in part, contains material as it appears in Liu, He; Lu, Feng; Forencich, Alex; Kapoor, Rishi; Tewari, Malveeka; Voelker, Geoffrey M.; Papen, George; Snoeren, Alex C.; George, Porter. “Circuit Switching Under the Radar with REACToR”, 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Seattle, WA, April 2014 The dissertation author was the fourth author on this paper.

VITA

2003-2007	B.Tech in Information Technology, Indian Institute of Information Technology, Allahabad, India
2009-2012	M.S in Computer Science, University of California, San Diego
2011-2015	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

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, Proceedings of the 11th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), Seattle, WA, April 2014.

Bullet Trains: A study of NIC burst behavior at microsecond timescales, Rishi Kapoor, Alex C. Snoeren, Geoffrey M. Voelker, and George Porter, Proceedings of ACM CoNEXT, Santa Barbara, CA, December 2013.

Dahu: Commodity Switches for Direct Connect Data Center Networks, Sivasankar Radhakrishnan, Malveeka Tewari, Rishi Kapoor, George Porter, and Amin Vahdat, Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), San Jose, California, October 2013.

Chronos: Predictable Low Latency for Data Center Applications, Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat, Proceedings of the ACM Symposium on Cloud Computing (SOCC), San Jose, CA, October 2012.

ThemisMR: An I/O-Efficient MapReduce, Alexander Rasmussen, Michael Conley, Rishi Kapoor, Vinh The Lam, George Porter, and Amin Vahdat, Proceedings of the ACM Symposium on Cloud Computing (SOCC), San Jose, CA, October 2012

NetBump: User-extensible Active Queue Management with Bumps on the Wire. Mohammad Al-Fares, Rishi Kapoor, George Porter, Sambit Das, Hakim Weatherspoon, Balaji Prabhakar and Amin Vahdat Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Austin, Texas, October 2012.

xOMB: Extensible Open Middleboxes with Commodity Servers James William Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat, Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Austin, Texas, October 2012

scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs, Harsha V. Madhyastha, John C. McCullough, George Porter, Rishi Kapoor, Stefan Savage, Alex C. Snoeren, and Amin Vahdat, USENIX ;login: 37(3), June 2012

scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs. Harsha V. Madhyastha, John C. McCullough, George Porter, Rishi Kapoor, Stefan Savage, Alex C. Snoeren, and Amin Vahdat. To appear in Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12), San Jose, CA, February 2012

ABSTRACT OF THE DISSERTATION

A Data Center End-host Stack for Predictable Low Latency and Dynamic Network Topologies

by

Rishi Kapoor

Doctor of Philosophy in Computer Science

University of California, San Diego, 2015

Professor George Porter, Co-Chair
Professor Amin Vahdat, Co-Chair

The scale of modern data centers enables developers to deploy applications across thousands of servers. The variety of applications and the scale of operations impose onerous challenge of meeting application performance requirements while maintaining efficiency. Today, data center operators typically over-provision the network and run services at low utilization to rein in latency outliers, thus decreasing efficiency. This large scale inefficiency results in high monetary, energy, and management expenses.

This dissertation focuses on redesigning the end-host network stack to improve network efficiency and achieve low latency and latency variation at high utilizations . We begin by studying traffic emanating from modern servers across a variety of data center applications. We find that traffic is highly bursty, which contradicts the network flow model where traffic is uncorrelated. We use this observation to design networks that can benefit from bursty behavior.

Second, in data center applications, predictability in service time and controlled latency, especially tail latency, are essential for building performant applications. Current practice has been to run such services at low utilization to rein in latency outliers, which decreases efficiency. To combat this, we present Chronos, which is a framework to reduce end-host latency and latency variation. Chronos reduces Memcached latency by a factor of 20 compared to typical deployments.

Third, a range of new data center switch designs incorporating wireless or optical circuits depend on fast reconfiguration of the underlying topology. These hybrid designs assume a perfect, closed-loop control plane which end-host network stacks cannot provide today. We present the design and implementation of a closed-loop control plane using only software changes at the end host operating system that enable these topologies to support unmodified applications running over TCP.

Taken together, these contributions demonstrate we can meet performance requirements of data center applications while running data centers at high levels of efficiency.

Chapter 1

Introduction

The Internet has changed the way we communicate with each other and with our surroundings. It is estimated that there are more than 2.9 billion people connected to the Internet and for the majority, the Internet is still an abstract notion where they store and access emails, photos, music and other bits of information. As the Internet has so seamlessly integrated in our lives, the Internet services that help us to navigate information and maintain and create new social and professional links are taken for granted. Many of the everyday services we use, such as e-commerce, payment of bills, and banking, now run on Internet.

The Internet has revolutionized the way we access and also from where we access information. This transition has been remarkably fast over the past decade. For example, in 1990s videos were stored on tapes, then came CDs followed closely by DVDs and Blu-ray. Now, with Internet based video streaming services such as Netflix and YouTube, users can access videos instantly from anywhere in the world. This instant access that was a privilege a few years ago is already a presumed thing and with the growing popularity of cell phones these services are a finger tap away. Many of these services are customized based on user preference.

These services not only need to deliver Gigabytes of data and show customized results and recommendations to the user (“Big-data”), but also deliver them instantly and predictably (predictable latency). Studies have shown over the years, humans value predictability in performance over faster average performance [72, 79] and can discern variations in performance down to 150ms [80].

To support billions of users and a plethora of services, service operators have built server farms (or data centers) across the globe. These facilities are spread across 10,000s of sq. ft with tens of thousands of servers storing petabytes of data. Inside the data center the applications and services are distributed across hundreds of servers. These servers are interconnected through complex networking fabric. The scale of data centers, imposes high monetary, energy, and management costs, placing increased importance on efficiency.

Within data centers, applications have different and stringent performance requirements. For applications such as Web search and Memcached [53], the networks must provide low-latency access to data that is spread across the cluster. For "Big data" applications such as Hadoop [87], networks must deliver high bisection bandwidth. To meet cost efficiency, services and applications often share the same networking and end-host infrastructure. Thus, in addition to meeting the latency and bandwidth requirements of applications, the operators also need to isolate different tenants and applications from each other. For this data center operators deploy a variety of packet processing devices that process data packets to implement several value-added services.

However, to meet these stringent applications requirements data center operators typically over-provision the network and run these applications at low utilization to rein in latency outliers, thus decreasing efficiency. This large scale inefficiency results in high monetary, energy, and management costs. In this work, we focus on leveraging the end-host stack i.e. applications, operating system and NICs to improve network efficiency and application performance. Higher levels of efficiency can be leveraged to either perform more work for each application, scan more data to return better results, or to do the same amount of work on a smaller machine footprint, reducing capital and operating expenses.

In the remainder of this chapter, we begin by describing trends and challenges in data center and how it affects overall data center efficiency. In this context, we present our hypothesis and describe the approach we took to solve these challenges. We end the chapter by providing the organization of this thesis.

1.1 Background and challenges

1.1.1 Internet services and tail latency

Modern Web applications often rely on composing the results of a large number of subservice invocations. For example, an end-user response may be built incrementally from dependent requests to networked services such as caches or key-value stores. Or, a set of requests can be issued in parallel to a large number of servers (e.g., Web search indices) to locate and retrieve individual data items spread across thousands of machines. Hence, the 99^{th} percentile of latency typically defines service level objectives (SLOs): when hundreds or thousands of individual remote operations are involved, the tail of the performance distribution, rather than the average, determines service performance. Being driven by the tail increases development complexity and reduces application quality [71].

To meet SLOs, Web service designers must consider the composition of the latency of a large number of small requests. For example, to fulfill interactive requests from end users, many modern Web applications must locate and retrieve thousands of individual data items spread across thousands of machines. In social networking, generating the “news feed” or Inbox for a user might involve retrieving thousands of individual photos, news items, status feeds, and advertisements with little to no spatial locality among the thousands of servers hosting the data items [48]. As another example, performing a Web search involves retrieving portions of the inverted index from hundreds to thousands of servers before performing a join and ranking across the retrieved content [24].

Since each of these individual operations involves retrieving a relatively small amount of data from a remote server, it is the latency of individual operations—the union of application, operating system, network interface, and network fabric latency—that matters for user satisfaction rather than any sustained throughput. At the same time, a particular operation cannot complete until *all* of the operations required to build a page complete, potentially requiring access to thousands of servers. This means that it is the tail latency that determines the performance of an individual operation and certainly tail latency that is critical to defining what is possible for achieving target SLOs.

Web service architecture has thus evolved to under-utilize available compute and server performance. Predictability is more important than sustained throughput and the higher the average level of utilization, the more variation in the performance of individual operations, and hence the longer the tail.

1.1.2 Reconfigurable topologies

As more applications and services move to data centers, the data center operators must scale up their infrastructure to support billions of users and services. Today, large-scale data center installations are limited by the ability to provide sufficient internal network connectivity. Delivering scalable packet-switched interconnects that can support the continually increasing data rates required between literally hundreds of thousands of servers is an extremely challenging problem that is only getting harder. Fundamentally, the packet-switching technology underlying current data-center interconnects limits their ability to scale: implementing control logic that is capable of deciding how to forward each packet individually is costly at present, and will rapidly cease to be feasible as link data rates increase.

Researchers have attempted to address this issue by adopting the superior power and cost scaling enabled by optical circuit switching [33]. Traditionally, circuit switching has been at odds with the packet-switch discipline that many applications depend upon (to provide, for example, low latency connectivity to a large number of destinations). Researchers have tried to address this discrepancy by proposing hybrid architectures that combine packet and circuit-switched interconnects [33, 85].

At their core, these approaches search out large, stable flows and route them over circuits, while forwarding the bulk of the traffic through the packet network. Initial designs are limited by the slow switching time (10s of milliseconds) of commercially available MEMS-based optical circuit switching technology, which makes it necessary to combine traffic from multiple end hosts to get traffic aggregates that remain stable at the timescales required (seconds) to achieve reasonable levels of circuit efficiency.

These “hybrid” networks propose to schedule appropriately large traffic demands via a high-rate circuit switch and handle any remaining traffic with a low-rate packet switch. All recent proposals for such hybrid designs assume a perfect closed-loop control plane. In practice, the performance of any hybrid network is critically dependent on all aspects of the closed-loop control plane including the speed of the demand estimation, how that demand is used to calculate the schedule in near real-time, and the ability to synchronize endpoints across circuits. However, researchers have stopped short of addressing the resulting closed-loop control plane problem.

1.1.3 Data center end-host traffic pattern

To achieve different application performance requirements data center operators deploy a variety of packet processing technologies and protocols spanning all layers of the network stack. Individual data flows are forwarded through a variety of middleboxes [74], which process data packets to implement a variety of value-added services. Most recently, the adoption of software-defined networking (SDN) means that software controllers are involved in providing basic connectivity services within the network—a task previously assigned to custom, high-performance hardware devices [58]. At the same time, link rates continue to increase, first from 1 Gbps to 10 Gbps, and now 10 Gbps to 40 Gbps, 100 Gbps [2], and beyond. Given the increasing speed of network links, and increasing complexity of packet processing tasks, handling these data flows is a growing challenge [74].

The result of each of these trends is that more devices, running a mixture of software and hardware, sit on the data plane, handling packets. Designing efficient and performant packet processing devices, either in software or hardware, relies on having an understanding of the traffic that will transit them. Each of these devices has to carefully manage internal resources, including TCAM entries, flow entry caches, and CPU resources, all of which are affected by the arrival pattern of packets.

1.2 Hypothesis and approach

In the previous section we described some of the challenges facing data center network operators. Towards tackling these challenges, we make the following hypothesis: That by redesigning the end-host stack it is possible to overcome following challenges

- Designing cost-efficient networking devices.
- Building systems with predictable low latency at high levels of utilization.
- Supporting rapidly reconfigurable topologies.

At high level, we harness the fact that a single administrative entity owns the entire end-host stack and the networking interconnect, thus enabling us to customize the end-host stack. Our approach has been to conduct a series of systematic measurement studies to identify key properties and performance bottleneck(s) in data center systems and use these observations to design and build more performant and efficient systems. We begin by analyzing network traffic emanating out of an end-host at microsecond timescales and identify challenges associated with bursty traffic. We further extend the measurement study to understand effects of latency variation on data-center communication patterns and find out components that contribute to variance in latency in data center end-hosts. Using these observations, we design and built two systems that tackle previously mentioned challenges.

In the next section we look at our contribution towards tackling these challenges.

1.3 Contributions

1.3.1 A study of NIC burst behavior at microsecond timescales

To address the first challenge, we analyze the output of a modern end-host server running a variety of data center workloads. With this study our aim is to better understand the network dynamics of modern data center servers and applications, and through that information, to enable more efficient packet processing across the network stack.

We begin by identify the causes of bursty traffic across the end-host stack in a data center setting. Next, we performed a traffic analysis of a set of testbed servers with different bandwidth-constrained applications, NIC settings, disk settings and OS features. We find that traffic is highly bursty, which contradicts the network flow model where traffic is uncorrelated. We use this observation to We further find that the level of this burstiness is largely outside of application control, and independent of the behavior of higher level applications. These short term bursts have both positive and negative implications and we leverage this observation to design networks that can benefit from bursty behavior.

1.3.2 Predictable Low Latency for Data Center Applications

To address the second problem of delivering predictable low latency for data center applications, we begin by analyzing impact of tail latency on data center traffic patterns. Next, we analyze sources of latency and latency variation, exposing application bottlenecks with user-level networking APIs. Based on these observations, we design Chronos, a communication framework that leverages both user-level networking APIs and NIC-level request dispatch. Chronos directs incoming requests to concurrent application threads in a way that drastically reduces, and in some cases eliminates, application lock contention. Chronos also provides an extensible load balancer that spreads incoming requests across available processing cores to handle skewed request patterns while still delivering low latency response time. Finally, We evaluate the resulting performance of three representative applications on a testbed with 50 servers.

1.3.3 Closed-loop control plane for reconfigurable topologies

Finally, to support a range of new data center switch designs incorporating optical circuit technologies that are enabling fast reconfiguration of the underlying topology. We propose and experimentally evaluate a practical first-generation closed-loop control plane for a hybrid network using only software changes at the end-host operating system that enable theses topologies to support unmodified applications running over TCP. We show how the network traffic demand can be estimated in the host stack and communicated to a controller. We demonstrate that the transport protocol TCP is not affected if the underlying reconfigurable technology is rapidly switching, but performs poorly if reconfigurable topology has paths of different capacity. Finally, we show that MPTCP is better suited for reconfigurable topology with multiple paths and demonstrate that our closed-loop control plane can respond to dynamic application demands.

1.4 Organization

In the remainder of this thesis, we first give background and related work. Second, we present a study of NIC burst behavior for bandwidth-intensive data center applications at microsecond timescales (chapter 3). Then we study the impact of tail latency on latency-sensitive data center application (chapter 4). Next, we analyze sources of latency and latency variation in data center applications (chapter 5). Based on our observations in chapter 4 and chapter 5, we present Chronos, a framework using user-level networking APIs that leverages NIC support to reduce lock contention and perform efficient load balancing to reduce the tail latency in data center networks and evaluate the resulting performance of three representative applications on a testbed with 50 servers (chapter 6). Next, we focus on challenges associated with network interconnect bandwidth and present a closed-loop system for reconfigurable topology and experimentally evaluate it (chapter 7). Finally, we summarize our findings and end the discussion with our system limitations and open problems.

1.5 Acknowledgements

This chapter in part, contains material as it appears in Kapoor, Rishi; Porter, George; Tewari, Malveeka; Voelker, Geoffrey M. ; Vahdat, Amin. “Chronos: Predictable Low Latency for Data Center Applications”, Proceedings of the ACM Symposium on Cloud Computing (SOCC), San Jose, CA, October 2012 The dissertation author was the primary investigator and author on this paper.

Kapoor, Rishi; Snoeren, Alex C.; Voelker, Geoffrey M. ; Porter, George. “Bullet Trains: A study of NIC burst behavior at microsecond timescales”, Proceedings of ACM CoNEXT, Santa Barbara, CA, December 2013. The dissertation author was the primary investigator and author on this paper.

This chpater in part, contains material that has prepared for submission for publication. Liu, He; Kapoor, Rishi; Tewari, Malveeka; Forencich, Alex; Zhang, Sen; Savage, Stefan; Voelker, Geoffrey M.; Papen, George; Snoeren, Alex C.; George, Porter. "Scheduling Circuits in a Packet World". The dissertation author is the second author on this paper.

Chapter 2

Background and Related Work

In this chapter we give background upon which the rest of dissertation is built. We start by giving historical context to study of traffic bursts in network, followed by giving an overview of different layers of the end-host stack that accounts for bursty traffic. In the second part of the chapter, we give a detailed overview of Memcached, a popular Key-Value store deployed at several large Internet companies. Given Memcached wide adoption and deployment, understanding and improving its latency performance will enable more responsive and efficient applications for existing deployments. We follow this by a discussion on existing work on achieving low latency and latency variation. and building a closed-loop control plane for dynamic network topologies.

2.1 Study of traffic patterns

In data centers designing efficient and performant packet processing devices, either in software or hardware, requires a deep understanding of the traffic that will transit them. Each of these devices has to carefully manage internal resources, including TCAM entries, flow entry caches, and CPU resources, all of which are affected by the arrival pattern of packets.

There have been several studies which have looked at traffic patterns in deployed networks. Nearly thirty years ago, Jain and Routhier [42] analyzed the behavior of shared bus Ethernet traffic at MIT, and found that such traffic did not exhibit a Poisson distribution. Instead, they found that packets followed a “train” model. In the train model, packets exhibit strong temporal and spacial locality, and hence many packets from a source are sent to the same destination back-to-back. In this model packets arriving within a parameterized inter-arrival time are called “cars”. If the inter-arrival time of packets/cars is higher than a threshold, then the packets are said to be a part of different trains. The inter-train time denotes the frequency at which applications initiate new transfers over the network, whereas the inter-car time reflects the delay added by the generating process and operating system, as well as CPU contention and NIC overheads. In Chapter 3, we re-evaluate modern data center networks to look for the existence and cause of bursts/trains (two terms that we will use interchangeably).

A variety of studies investigate the presence of burstiness in deployed networks. In the data center, Benson et al. [16] examine the behavior of a variety of real-world workloads, noting that traffic is often bursty at short time scales, and exhibits an ON/OFF behavior.

2.1.1 Sources of bursts

Next, we highlight the different layers of the network stack that account for bursts, including the application, the transport protocol, the operating system, and the underlying hardware (summarized in Table 2.1).

2.1.2 Application

Ultimately, applications are responsible for introducing data into the network. Depending on the semantics of each individual application, data may be introduced in large chunks or in smaller increments. For real-time workloads like the streaming services YouTube and Netflix, server processes might write data into socket buffers in

Table 2.1. Sources of network bursts.

Batching source	Examples	Reason for batching
Application	Streaming Services (Youtube, Netflix)	Large send sizes
Transport	TCP	ACK compression, window scaling
OS	GSO/GRO	Efficient CPU & memory bus utilization
NIC	TSO/LRO	Reduce per packet head
Disk drive	Splice syscall, disk read-ahead	Maximize disk throughput

fine-grained, ON/OFF patterns. For example, video streaming servers often write 64 KB to 2 MB of data per connection [37, 67].

On the other hand, many “Big Data” data center applications are limited by the amount of data they can transfer per unit time. Distributed file systems [12, 36] and bulk MapReduce workloads [29, 68] maximize disk throughput by reading and writing files in large blocks. Because of the large reads these systems require, data is sent to the network in large chunks, potentially resulting in bursty traffic due to transport mechanisms, as described in the next subsection. NFS performs similar optimizations by sending data to the client in large chunks, in transfer sizes specified by the client.

2.1.3 Transport

At the transport layer, the widely used TCP protocol exhibits, and is the source of, a considerable amount of traffic bursts [19, 3, 43, 9]. Aggarwal et al. [3] and Jiang et al. [43] identify aspects of the mechanics of TCP, including slow start, loss recovery, ACK compression, ACK reordering, unused congestion window space, and bursty API calls, as the sources of these overall traffic bursts. Previous studies in the wide area include Jiang et al. [44], who examined traffic bursts in the Internet and found TCP’s self-clocking nature and in-path queuing to result in ON/OFF traffic patterns.

Beyond the basic protocol, some improvements have an impact on its resulting burst behavior. One such example is TCP window scaling (introduced, e.g., in Linux 2.6), in which the TCP window representation is extended from 16-bits to 32-bits. The resulting increase in the potential window allows unacknowledged data to grow as large as 1 GB, resulting in potentially large traffic bursts.

2.1.4 Operating system

The operating system plays a large role in determining the severity of traffic bursts though a number of mechanisms. First, the API call boundary between the application and OS can result in bursty traffic. Second, a variety of OS-level parameters affect the transport and NIC hardware behavior, including maximum and minimum window sizes, window scaling parameters, and interrupt coalescing settings. Finally, the in-kernel mechanisms Generalized Receive Offload (GRO) and Generalized Segmentation Offload (GSO) can be used to reduce CPU utilization, at the expense of potentially increased burstiness. GSO sends large "virtual" packets down the network stack and divides them into MTU-sized packets just before handing the data to the NIC. GRO coalesces a number of smaller packets into a single, larger packet in the OS. Instead of a large number of smaller packets being passed through the network stack (TCP/IP), a single large packet is passed.

2.1.5 Hardware

The behavior of hardware on either the sender or receiver affects the burstiness of network traffic. We now review a few of these sources.

Disk drives: To maximize disk throughput, the OS and disk controller implement a variety of optimizations to amortize read operations over relatively slow disk seeks, such as read-ahead caches and request reordering (e.g., via the elevator algorithm). Of particular interest is the interaction between the disk and the *splice()* or *sendfile()* system calls. These calls improve overall performance by offloading data transfer between storage devices and the network to the OS, without incurring data copies to userspace. The OS divides these system call requests into individual data transfers. In Linux, these data transfers are the size of the disk read-ahead, which are transferred to the socket in batches and result in a train of packets. The read-ahead parameter is a configurable OS parameter with a default value of 128 KB.

Segmentation offload: TCP Segmentation Offload (TSO) is a feature of the NIC that enables the OS to send large virtual packets to the NIC, which then produces numerous MTU-sized packets. Typical TSO packet sizes are 64 KB. The benefit of this approach is that large data transfers reduces per-packet overhead (e.g., interrupt processing), thereby reducing CPU overhead.

Interrupt Coalescing (IC) and Large Receive Offload (LRO): On the receive side of the NIC driver, IC and LRO both further reduce the CPU overhead of receiving packets at high speed. IC delays receive interrupts until a number of packets have been received, delivering those packets in batches to the OS. LRO combines multiple consecutive packets into a larger, virtual packet in the NIC (as opposed to GRO which combines the packets in the OS). Both potentially affect the burstiness of TCP, as studied in [7, 64, 88].

2.2 Key-Value stores

Key-Value (KV) stores serve as a basic building block for building loosely-coupled distributed systems. Memcached is a popular package supporting the operation of a number of large-scale Web services, and given its wide adoption and deployment, improving its latency performance will enable more responsive and efficient applications for existing deployments.

2.2.1 Memcached overview

We provide an overview of Memcached, a widely used KV-store. It has been deployed at Facebook, Zynga, Twitter, and others [17, 53], with approximately 2,000 Memcached servers in deployment at Facebook [60]. We start by describing Memcached itself,

2.2.2 System description

Memcached is an open source, in-memory KV-store for small chunks of unstructured data. The API is quite simple, consisting of operations that get, set, delete, and manipulate key-value pairs. KV pairs are stored entirely in-memory, with no persistence. During low-memory conditions, KV-pairs are evicted from the cache according to a least-recently used (LRU) discipline. Replication is not provided by the current release.

Memcached deployments split functionality between one or more servers and a number of clients. KV-pairs are partitioned across the set of servers using a hash function shared between the clients and the servers. In Memcached, clients are independent, and issue requests directly to a single server responsible for a particular key based on the shared hash. This simplifies the design of the distributed server tier, since servers do not need to communicate with one another. Thus, scaling the server tier is trivial since Memcached does not need to ensure cache consistency or invalidate data. Clients are responsible for inserting KV-pairs into the cache, as well as deleting or invalidating them according to the semantics of the application.

Internal data layout:

Memcached's internal memory management layer works as follows. KV-pair data is stored in a slab-managed data region. Memcached allocates a configured amount of memory in 1 MB *pages*. These pages are assigned to one or more *slab classes*. Each slab class corresponds to a particular fixed *chunk* size. A chunk holds a single KV-pair, and KV-pairs are assigned to slab classes on a best fit basis. One or more pages are assigned to the different slab classes depending on the sizes of objects inserted by the clients. When a page is assigned to a slab class, it is divided into fixed-sized chunks based on the particular slab class it is assigned to.

Initially, pages consist of empty chunks, available for storing new items. When KV-pairs are deleted from the cache, they are simply marked as deleted, and lazily reclaimed during runtime. When a KV-pair is inserted into a particular slab class (based on its size), Memcached first looks for empty chunks, inserting the item there if possible. If no chunks are empty, then Memcached tries to lazily delete items that have been explicitly deleted, or have expired. If that is not possible, then it will evict an unexpired data item according to an LRU-based replacement policy.

Memcached request handling

We now describe how the cache is manipulated from both the client and server perspective.

Client-side get: To request an item, a client performs the following steps:

1. The client determines the server responsible for a desired KV-pair by hashing the key using a globally shared hash function. In practice, this hash function is simply computed as the value of the key modulo the number of servers. The most recent version relies on consistent hashing.

2. The client issues a *get()* operation to the resulting server. For high-throughput environments, this request is typically issued using UDP [73] to reduce latency and require fewer kernel resources to support a large number of open TCP connections.

3. The server either returns the key and value if present, or an indication that the key is not present.

4. If the key was not present, then the client will request the value from an authoritative data source (typically a database), and then insert it back into the cache as described below.

Client-side store: Storing data in Memcached proceeds as follows.

1. Initially, the client determines the data to store in the cache, typically by issuing a query to an authoritative data source such as a database.

2. Using the same global hash function as above, the client determines the server responsible for the target key.

3. The client issues a *set()* operation to the server. The server either indicates success, or signals an exception.

Server side: On receiving a request, Memcached is notified by the O/S. Multithreaded Memcached is based on libevent, and the request arrival is enqueued to the library as a new event. The application wakes up all threads, and one of them successfully takes ownership of the event and reads the packet (or socket) payload. The request is parsed to determine the protocol used (e.g., ASCII or binary) as well as the type of operation.

When handling a *get()* request, the application thread must locate the appropriate chunk corresponding to the KV-pair among the slab classes (and associated memory pages). It does this by looking up the key in an in-memory hash table. This table contains a pointer to the appropriate page in its slab class. To support multi-threading, this hash table is protected by a single mutex. Furthermore, the entire set of slab classes is protected by a single mutex as well.

There have been numerous efforts to improve Memcached throughput [17, 73], though none specifically look at improving predictable tail latency. Previous efforts to improve performance of Memcached over RDMA protocol [13, 46] required redesigning Memcached from the ground up and works only for a single threaded implementation.

2.2.3 Other KV-stores

Other KV-stores have been deployed with different storage semantics. Dynamo [32] is an eventually-consistent distributed KV-store supporting a high insertion and query rate while surviving failures of individual components. Other persistent KV-stores include BerkeleyDB [18], LevelDB [50], and Redis [69].

Numerous proposals for managing internal storage of KV-pair data have been made. BufferHash [10] employs multiple hash tables with an in-memory Bloom filter to reduce the number of accesses to flash. HashCache [14] focuses on minimizing memory requirements to support resource-constrained deployments by optimizing data on large, external, persistent storage devices rather than relying on in-memory indexes. FAWN [11] constructs a distributed KV-store relying on consistent hashing to determine the appropriate node, and then relying on a log-structured on-disk data layout indexed by an in-memory hash table to reduce the latency of accesses. FlashStore [30] uses a log structured on-disk layout to reduce flash accesses (and resulting wear), and evicts cold data from the flash tier to disk, using in-memory lookups to decrease latency. SkimpyStash [31] and SILT [51] both aim to reduce the memory requirements of the KV-store. In the former case, SkimpyStash relies on a hash table with linear chaining on collision, persisting those chains to persistent storage. In the case of SILT, a multistore consisting of one of three data layout policies is employed to reduce the memory overhead of storing data to close to minimal. We consider Chronos complimentary to these efforts in that we focus on lowering the latency and latency variation of forwarding requests from the network to the black-box application logic.

2.3 Predictable low latency

In Chronos, we observe that the operating system accounts for more than 90% of end-host latency. We also find that lock contention limits application performance. In this section, we discuss related work in area of operating system improvements and lock contention.

2.3.1 Optimized Network/OS interfaces

A key bottleneck that our work addresses is the kernel and network stack overhead. We share this goal with several academic and industrial efforts. User-level networking was developed to support applications which emit packets at a high rate, and to reduce latency in the kernel [25, 20, 84]. Arsenic [65] proposed installing custom filters in NIC for packet classification. While user-level networking APIs are integral to the early partitioning aspect of our design, Chronos also facilitates per-CPU core load balancing and removing application lock contention through deep-packet inspection using these APIs to reduce application tail latency. Myrinet [21] and Infiniband [40] are examples of low-latency, high bandwidth interconnect fabrics that are often used in high-performance computing clusters. While Myrinet and Infiniband address a key bottleneck, Chronos focuses on commodity Ethernet switching and eliminates latency across the entire end-to-end application path, including application lock contention and hotspots.

2.3.2 Operating System Improvements

There have been various proposals on improving the scalability and performance of the Linux kernel. Corey [22] identified numerous instances of in-kernel data structure sharing which reduced potential parallelism across threads, and proposed address ranges, kernel cores, and sharing to improve kernel performance. In [70], the authors conclude that locking and blocking system calls were significant causes of application performance degradation. Boyd-Wickizer et al. [23] study the scalability of seven applications, including Memcached, across a 48-core machine and conclude that by modifying the kernel and applications, it is possible to remove many performance bottlenecks. However, their study focused on throughput, and not latency. With Chronos, we find that even for single-threaded processing the kernel introduces significant additional latency, even after accounting for these recent improvements. An analysis of latency in the end-host network stack was carried out by Larsen et al. [49].

2.3.3 Lock Contention

Lock contention has long been recognized as a key impediment to performance of shared memory and multi-threaded applications [78]. Replacing mutex locks with read/write locks may have little advantage [26]. Triplett et al. [81] propose a dynamic concurrent hash table with resizing using a *read-copy update* (RCU) mechanism. This mechanism works well in situations where the number of reads is significantly greater than writes. VoltDB [83] and H-Store [45] partition application state in memory across the CPU cores to achieve scalability. Here, incoming requests are partitioned at the application layer after arriving to the process. Our approach is different in that we rely on deep-packet capabilities of the NIC hardware to partition requests before they arrive to the OS or application.

2.3.4 Data center Networks & Applications

New transport protocols like QCN [5] and DCTCP [6] reduce in-network queueing and congestion, further reducing network latency. Recent proposals such as DeTail [89] and HULL [7] also focus on reducing latency by performing in-network traffic management.

2.4 Reconfigurable topologies

In section 2.1 we looked at sources of bursts in a data center end-host. One application of these bursts is that these bursts can be scheduled on short point to point link created between hosts by fast reconfigurable topologies. A range of new data center switch designs incorporating wireless or optical circuit technologies are enabling fast reconfiguration of the underlying topology. These “hybrid” networks propose to schedule appropriately large traffic demands via a high-rate circuit switch and handle any remaining traffic with a low-rate packet switch. Helios [34] and c-Through [85] are hybrid topologies that propose replacing electrical switch at the core layer of data centers with ‘MEMS’ based optical switch with electrical packet switch on the side. Similarly, several topologies have been proposed using wireless links [39, 47, 91].

2.4.1 Demand estimation

Demand estimation is the key to data-center traffic engineering. Existing Traffic engineering (TE) approaches collect network traffic matrix and runs an algorithm to identify elephant flows and then schedules them. The existing TE frameworks estimate network demand either based on heuristics e.g., Helios or uses buffer occupancy e.g., C-through. Helios collect port statistics from switches and uses this information as an indicator of future network traffic demand. Due to the nature of jobs running in data center demand may not be accurate e.g., if traffic lacks predictability. Helios demand estimator module takes few milliseconds to gather statistics and is not suited for rapidly switch reconfigurations like REACToR.

C-Through uses network statistics tool (netstat) to gather information about socket buffer occupancy. Netstat is a user space tool and launching and gathering this information takes milliseconds on our testbed. We use a similar approach but using a custom Linux kernel module. In our kernel module, we also restrict the size of hash-table that stores socket connections making it more efficient.

Fastpass demand estimator instruments send() and sendto() system calls and report the sizes of these calls to the controller.

2.5 Acknowledgements

This chapter in part, contains material as it appears in Kapoor, Rishi; Porter, George; Tewari, Malveeka; Voelker, Geoffrey M. ; Vahdat, Amin. “Chronos: Predictable Low Latency for Data Center Applications”, Proceedings of the ACM Symposium on Cloud Computing (SOCC), San Jose, CA, October 2012 The dissertation author was the primary investigator and author on this paper.

Kapoor, Rishi; Snoeren, Alex C.; Voelker, Geoffrey M. ; Porter, George. “Bullet Trains: A study of NIC burst behavior at microsecond timescales”, Proceedings of ACM CoNEXT, Santa Barbara, CA, December 2013. The dissertation a

Chapter 3

BulletTrains: A study of NIC burst behavior at microsecond timescales

In data centers designing efficient and performant packet processing devices, either in software or hardware, requires a deep understanding of the traffic that will transit them. Each of these devices has to carefully manage internal resources, including TCAM entries, flow entry caches, and CPU resources, all of which are affected by the arrival pattern of packets.

While numerous studies have examined the macro-level behavior of traffic in data center networks–overall flow sizes, destination variability, and TCP burstiness–little information is available on the behavior of data center traffic at packet-level timescales. Whereas one might assume that flows from different applications fairly share available link bandwidth, and that packets within a single flow are uniformly paced, the reality is more complex. To meet increasingly high link rates of 10 Gbps and beyond, batching is typically introduced across the network stack –at the application, middleware, OS, transport, and NIC layers. This batching results in short-term packet bursts, which have implications for the design and performance requirements of packet processing devices along the path, including middleboxes, SDN-enabled switches, and virtual machine hypervisors.

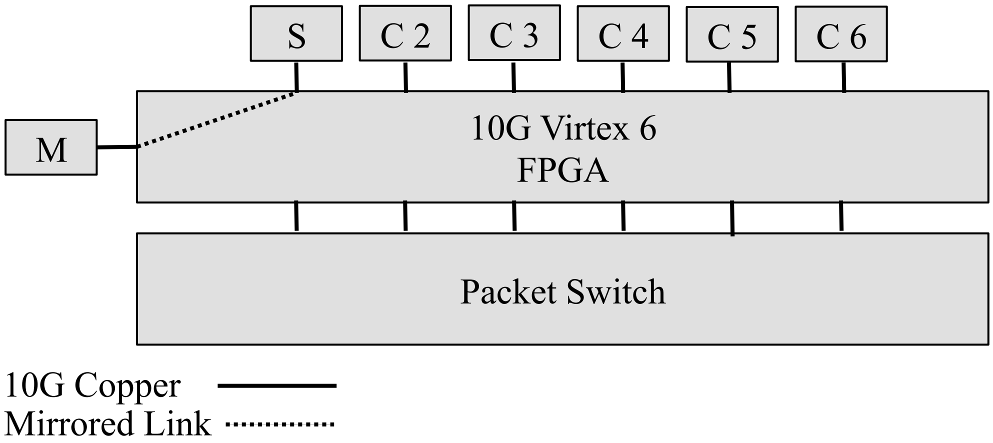


Figure 3.1. The data center testbed includes an Ethernet packet switch and an FPGA that timestamps packets at a 6.4-ns granularity.

In this chapter, we study the burst behavior of traffic emanating from a 10-Gbps end host across a variety of data center applications. We find that at 10–100 microsecond timescales, the traffic exhibits large bursts (i.e., 10s of packets in length). We further find that the level of this burstiness is largely outside of application control, and independent of the behavior of higher level applications. Through this study our aim is to better understand the network dynamics of modern data center servers and applications, and through that information, to enable more efficient packet processing across the network stack and support high bandwidth reconfigurable topologies.

3.1 Traffic measurements

To better understand the burst behavior of traffic emanating from hosts in data center environments, we have performed a traffic analysis of a set of testbed servers. Our evaluation seeks to:

1. **Measure the burstiness of a set of data center workloads.** For simple, long-lived communication patterns between a small number of nodes (e.g., a stride pattern), bursts can be quite large—up to about 100 packets. Even for workloads

with low destination stability, such as all-to-all patterns, bursts the length of a TSO segment (i.e., 64 KB) are seen in practice.

2. **Understand how each layer of the network stack contributes to burstiness.** We find that application behavior largely determines burstiness, yet even among bandwidth-constrained applications, configuration parameters strongly affect burstiness (for example, NFS configuration parameters or syscall parameters).

3. **Determine the effect of NIC and OS performance features (like GRO and TSO) on traffic emanating from the host.** For bandwidth-constrained workloads, we find that performance enhancing features strongly affect burstiness, especially TSO and LRO.

4. **Determine whether burstiness can be controlled through software changes.** Through modification of the TSO code in the kernel, it is possible to generate larger bursts.

We first describe our measurement methodology and then present our results.

3.1.1 Measurement methodology

In absence of packet-level traces and measurements on real data centers, we deployed a set of applications on a small, seven-node cluster. The applications we chose are the network file server (NFS), the Hadoop Distributed FileSystem (HDFS), a Hadoop MapReduce-based Terasort, and a set of microbenchmark applications generating synthetic traffic. Each of the applications we evaluate are bandwidth constrained.[1]

Hardware: We deployed the above applications on a set of HP DL360p servers, each with a pair of Intel E5-2630 six-core CPUs (2.3 GHz) running Debian Linux with kernel

[1] We also evaluated latency-sensitive applications like Memcached, but omit those results since they produced little to no traffic bursts.

version 3.6.6. Each server has 16 GB of memory and eight 146 GB 15K RPM hard drives (with an ext3 partition). Each server has an Intel 82599-based 10 Gbps NIC. The measurement server has a Myricom 10G-PCIE2-8B2-2S+E NIC. We used the Intel ixgbe driver (version 3.12.6) with default parameters and multiqueue disabled. With multiqueue enabled, we found that the traffic pattern was dependent on the specific NIC scheduling policy and hence we omitted multiqueue enabled results from our discussion.

OS configuration: The disk read-ahead buffer was configured to be 128 KB with a CFQ disk scheduler. At the start of our experiment, we drop caches to ensure that read requests go directly to the disk. The network interface was configured with a 1500-byte MTU. Unless otherwise noted, in all our experiments we enable both TSO and LRO on the NIC, though we increased the socket buffer size to be 32 MB to avoid bandwidth throttling due to receive window limitations. Linux 3.6.6's default setting of the TCP parameter `tcp_limit_output_bytes` was limiting the number of in-flight TCP packets, which reduced overall performance. Thus, we increased this parameter from 128 KB to 2 MB.

Network and packet capture: Each of the servers is directly connected to a 10 Gbps Xilinx Virtex 6 FPGA. The FPGA "passes through" each connection to ports on a 10 Gbps Fulcrum Monaco packet switch. The purpose of the FPGA is to, on demand, measure the traffic transiting to and from one of the servers. The FPGA generates a measurement record for each packet, including the packet's source and destination address, its size, and a timestamp of when the packet left or arrived to the port. The timing precision of the timestamp is 6.4 ns. The FPGA places per-packet measurements into packets destined to a dedicated measurement server, labeled *M*, as shown in Figure 3.1. For all the traces we ignore the TCP slow start behavior.

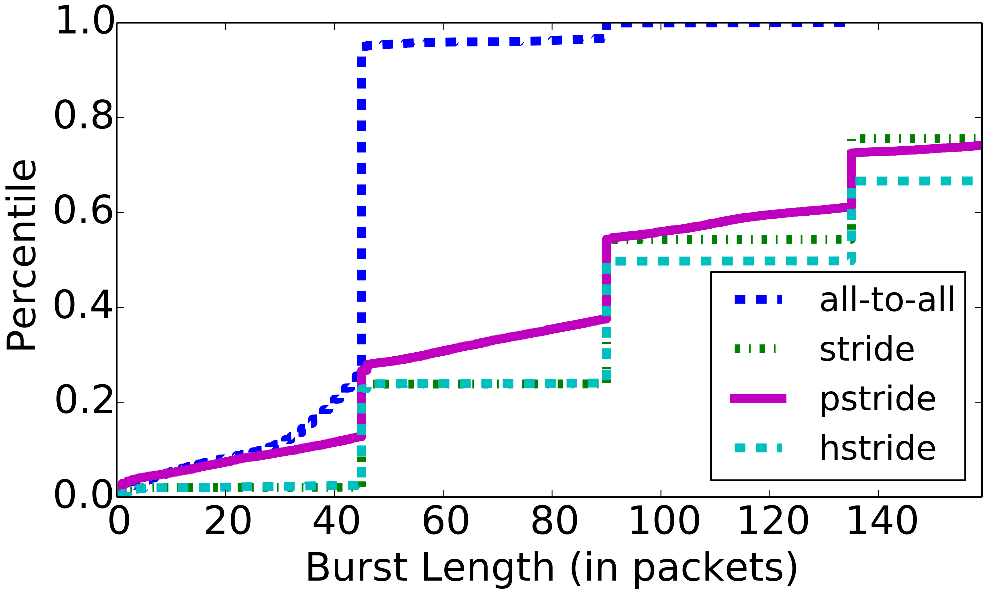


Figure 3.2. CDF of burst sizes with synthetic traffic patterns (with TSO and LRO enabled).

Application configuration: For the NFS experiments, we use NFS server version 3, and the NFS server exports a partition located on a single drive. The in-kernel NFS client reads a file by making multiple pipelined read calls to the server. The read size is specified while mounting the file system and the maximum read size supported by the NFS client is 1 MB. For the HDFS experiments, both the NameNode and the DataNode processes run on the same server, and the DataNode manages data on a single disk per machine. The HDFS data block size is 512 MB. Both the NFS and HDFS servers serve randomly generated files ranging in size from 25 MB to 5 GB. Each client, labeled *C1* through *C6* in Figure 3.1, copies data from the server to its local disk. We created identical copies of the same file on the server to allow parallel downloads from the clients. To ensure that all the clients start their downloads at the same time, we generate a coordination broadcast packet from a control host.

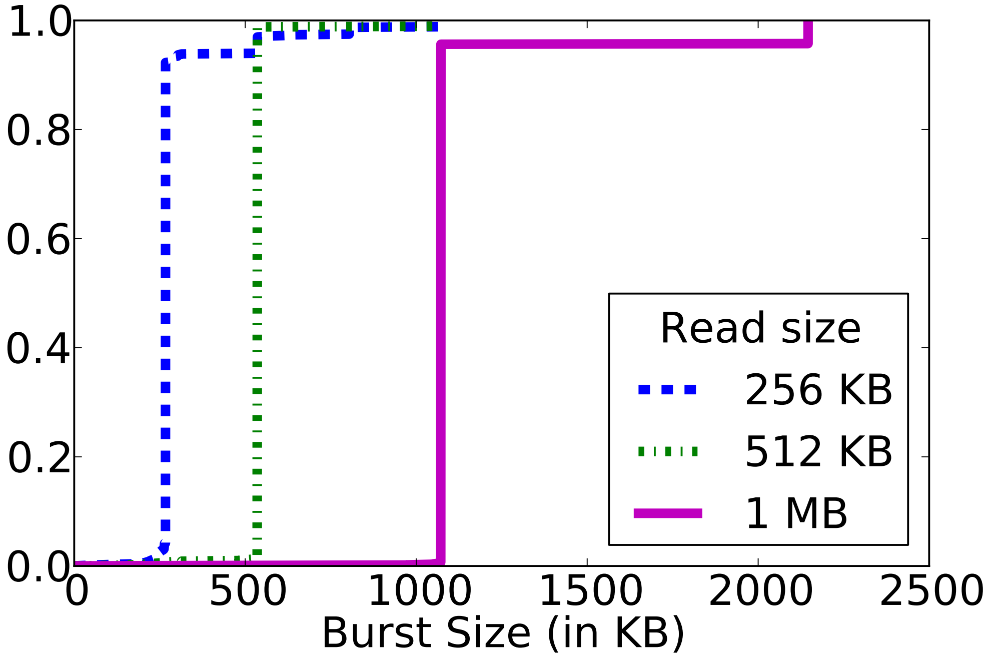


Figure 3.3. NFS server. Read syscall size large determine the burst size.

3.1.2 Microbenchmarks

To begin our evaluation, we measure a set of synthetic microbenchmarks, which are generated via simple memory-to-memory data transfers to eliminate any effects from the disks, application logic, and think time. The traffic patterns we consider are a *stride* pattern, in which a source host sends data to an intermediate host, which is in turn sending data to a destination host. This pattern differs from a simple transfer in that ACKs are intermixed with data packets. Next we consider two variants of the stride workload, both based on workloads used by Farrington et al. [34]. The *pstride* workload is similar to the stride workload, except that each host changes its destination in unison. In the *hstride* workload, each host opens many flows to each destination, and slowly changes the destination host at a flow level. Finally we consider an *all-to-all* workload where every host sends data to every other host.

Figure 3.2 shows burst lengths in packets. A *burst* is an uninterrupted sequential

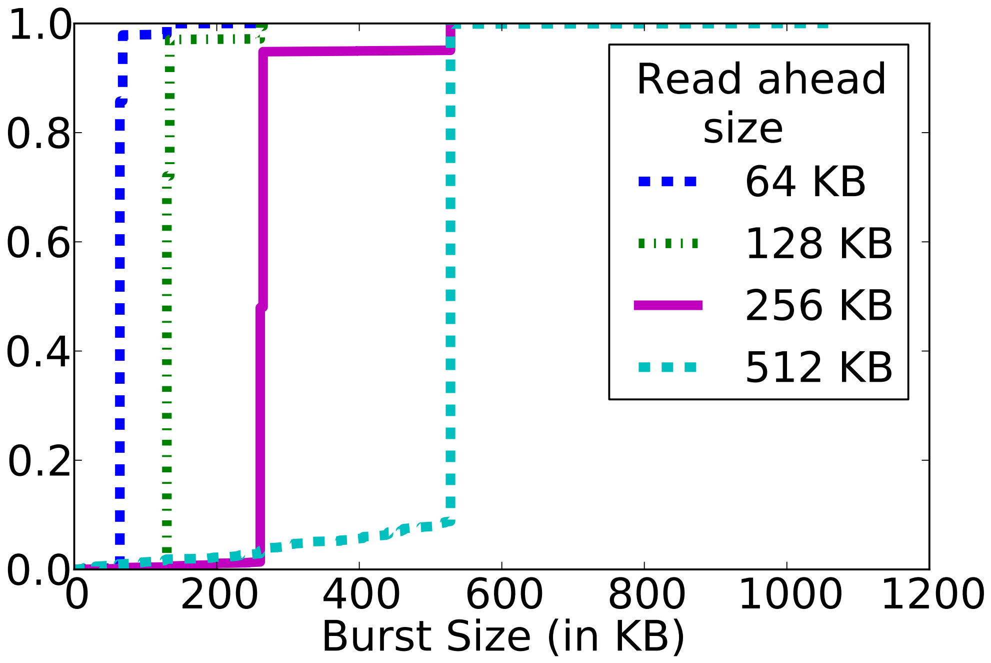


Figure 3.4. HDFS DataNode server. The read-ahead parameter determine the burst size.

stream of packets from one source to one destination with an inter-packet spacing of less than 40 ns. The burst length is the number of such closely spaced packets in the stream. We observe that the burst lengths for both the stride and hstride patterns are quite large—up to about 100 packets in the median case. Each of the jumps in the CDF fall at multiples of the TSO size (which in our testbed is 64 KB). This pattern corresponds to the NIC sending one or more TSO-sized amounts of data before switching destination flows. Most surprisingly, in the all-to-all pattern, the median burst length is 44 packets, corresponding to a TSO-sized amount of data. Even though there is no correlation at the application layer, the buffering done via TSO results in bursts of several dozen packets.

3.1.3 Effect of application behavior

We now analyze the burst behavior of three common, bandwidth-intensive applications: NFS, the Hadoop Distributed FileSystem (HDFS), and a MapReduce-based sorting program.

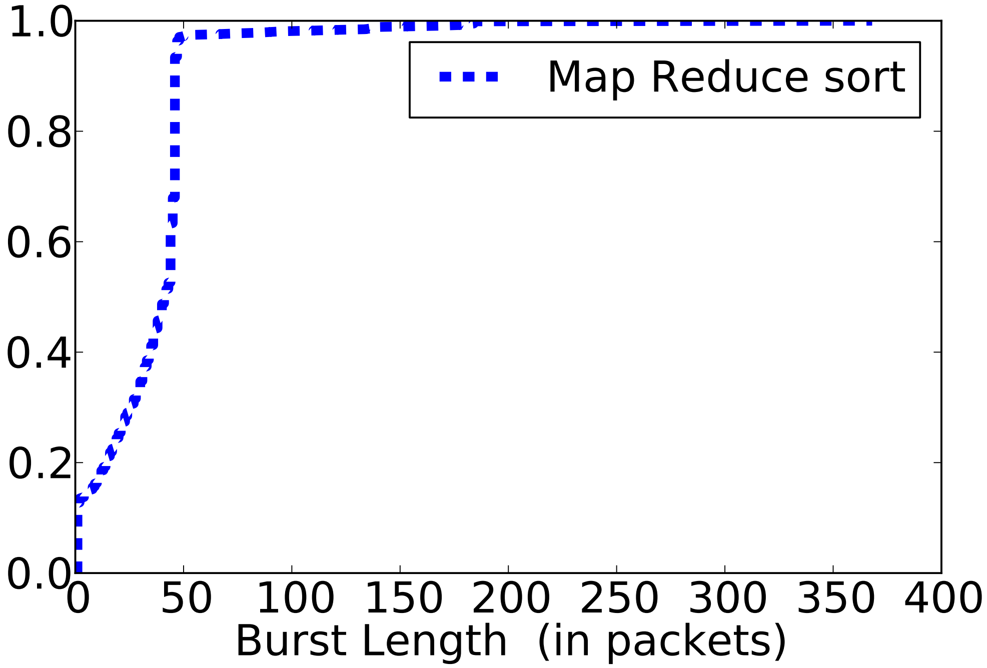


Figure 3.5. MapReduce Sort.For MapReduce workloads, intermediate data shuffling and various keep-alive messages reduce the burst length.

NFS: We configure six NFS clients to concurrently request a file from a single NFS server. Each client specifies the read size at volume mount time, and issues pipelined read requests to the server with a file offset and the read size. Figure 3.3 shows the resulting burst lengths. The length of the packet burst is highly correlated with the underlying read request size configured at the client. This correlation is due to a combination of mechanisms. The NFS server relies on the in-kernel `splice()` system call, which copies data from the disk in configurable batches (e.g., 128 KB by default). The default value of 128 KB can be tuned by changing the disk read ahead size via sysctl. Furthermore, the NFS server also performs its own batching as well, based on the read size that the client specified. As a result, the server reads the entire read size (e.g., 1 MB) off the disk into the memory buffers and then sends the buffered data to the networking stack. Our results confirm that application-layer batching can lead to highly bursty behavior.

HDFS: We next configured six HDFS clients to concurrently request a file from a single HDFS DataNode server. Figure 3.4 shows the resulting burst lengths. As in the NFS application, HDFS-based packet burst sizes are highly correlated with the read-ahead size, since HDFS indirectly uses the `sendfile()` call (via Java's `transferTo()` method). The `sendfile()` call copies data from the disk into the network buffers in batches of disk read-ahead size. Unlike the NFS server, no additional buffering or batching is done by the server process, and so the observed burst lengths correspond more directly to the implementation of `sendfile()`.

MapReduce Sort: To evaluate an all-to-all application we use Hadoop's Terasort. We installed both the NameNode and TaskTracker on node C6, and generated 120 GB of data spread across six remaining nodes (S and C1–C5). The resulting burst behavior is shown in Figure 3.5. The median burst length is approximately 64 packets, which is lower than the file transfer workloads above. Although still large, this lower burst length is due in part to a mixture of flows on each server, since each node exchanges intermediate data, status updates, keep-alive messages, and file transfer requests.

3.1.4 Effect of NIC hardware

We now examine several recent mechanisms designed to improve network performance and lower CPU utilization. Specifically, we examine LRO and TSO, deployed in the NIC, and GRO, which runs in the kernel. We repeated the microbenchmark experiments from Section 3.1.2, enabling or disabling these NIC parameters. Figure 3.6 shows the results of the all-to-all workload.

We find that enabling TSO affects the burst length directly, whereas LRO, by coalescing several packets on the receive side into larger acknowledged segments, indirectly causes burstiness. Thus, disabling TSO has a more prominent effect on shrinking the

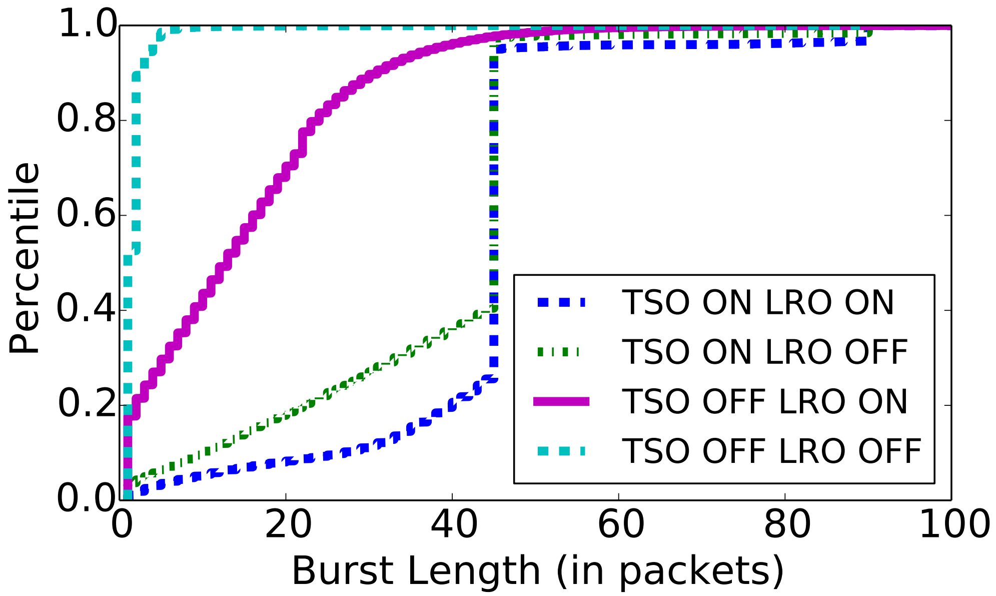


Figure 3.6. The TSO NIC mechanism directly increases the burst length, while LRO acts indirectly with a smaller effect.

Table 3.1. Average throughput with different NIC settings.

TSO	LRO	Average Throughput
ON	ON	8.7 Gbps
ON	OFF	7.1 Gbps
OFF	ON	5.5 Gbps
OFF	OFF	2.4 Gbps

burst size, as compared to disabling LRO. However, both strongly affect CPU utilization, as shown in Table 3.1. Disabling both LRO and TSO drops throughput to 2.4 Gbps, since the kernel cannot keep up with the rate of packets necessary to saturate the link. Combinations of LRO and TSO result in intermediate throughputs, and both enabled produce the highest throughput, as expected.

To understand the effect of LRO on burst size, we collected measurements of an *iperf* session between two servers with LRO either on or off, and calculated the throughput, CPU utilization, and ACK ratio for each case. The ACK ratio is the average number of data bytes acknowledged by a single ACK, one indication of burstiness.

Table 3.2. The effect of LRO on CPU utilization and throughput. The ACK ratio represents the amount of data acknowledged by a single ACK packet, and the "pinned" case refers to a configuration in which the application and interrupts are pinned to separate cores.

LRO Setting	CPU Util (%)	Throughput (Gbps)	ACK Ratio (KB)
ON	53	9.49	44
OFF	100	7.3	3
OFF (pinned)	95	9.25	3

Table 3.2 shows these results. Without LRO an ACK is generated for every other segment, whereas with LRO enabled up to 44 KB are acknowledged at a time, resulting in increased burstiness.

Noting that link-level burst lengths are highly correlated with the TSO sizes configured on our NICs, we next seek to understand whether the OS and applications would be able to send larger bursts if the TSO size were increased. Linux is limited to TSO sizes of 64 KB, and the NIC we used supports up to 256 KB. We were able to modify Linux to support a TSO size of up to 148 KB. Figure 3.7 shows the resulting burst lengths of an one-to-all workload, and indeed the OS and applications were able to send bursts up to this new maximum. Thus traffic leaving a server could be even more bursty by modifying the TSO mechanism.

3.2 Implications

We now discuss the implications—both positive and negative—of such bursts.

Negative effects of bursts: Blanton and Allman [19] analyze packet traces from three different networks and find that, for large burst lengths (e.g., those greater than 10 segments), the probability of dropping packets increases. In fact, with very large bursts

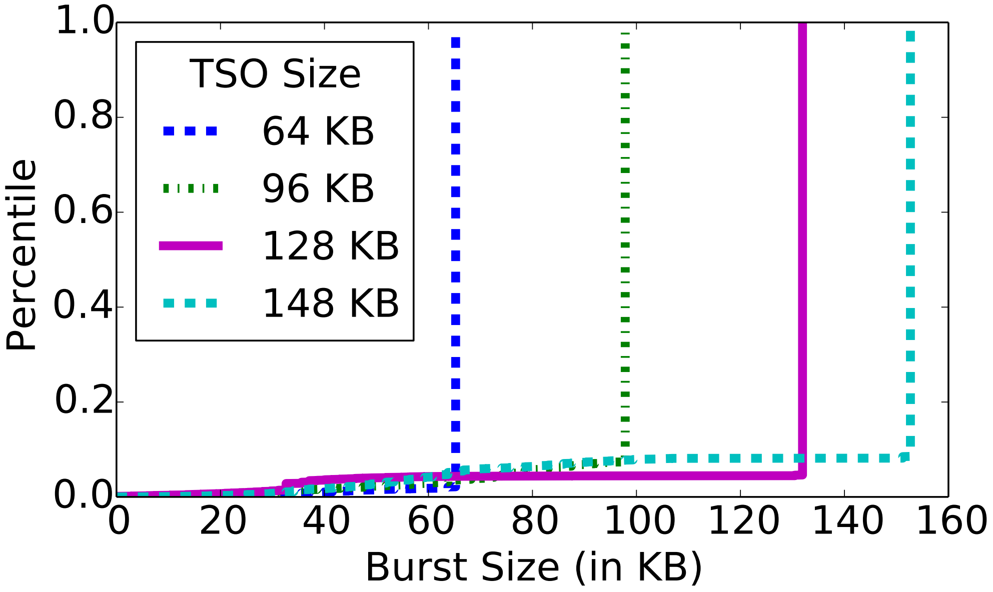


Figure 3.7. Increasing the TSO size beyond the default maximum of 64 KB results in larger bursts.

(i.e., 60 segments or more), the probability of dropping a single segment increases to 100%. This connection between packet bursts and packet loss is not confined to these examples. For example, in the YouTube network, bursty traffic is responsible for 40% of total packet losses [67]. Jiang et al. [43] show that flow-level bursts can result in increased queuing delay in the network. Alizadeh et al. [7] show that bursty traffic causes temporary increases in network buffer occupancy, which results in variable latency and higher packet losses within a data center. To reduce burstiness, Blanton and Allman have proposed placing a limit on the sending of new segments in response to an ACK, called *MaxBurst* [9]. Several efforts [7, 90, 3] argue for pacing packets to reduce TCP burstiness.

Positive effects of bursts: Designing efficient and performant packet processing devices (e.g., middleboxes, switches, SDN controller) relies on having an understanding of the traffic that will transit them. Each of these devices has to carefully manage internal resources including TCAM entries, flow entry caches, and CPU resources, which are affected by the arrival pattern of packets. The presence of bursts show that packets to a destination have temporal locality. Given a packet to a destination, we can predict with some probability that the next packet will be for the same destination. This temporal effect has interesting implications for data center traffic rule and policy management and middle-boxes. Routhier [42] argue that the presence of trains enable certain optimizations in the network, such as amortizing classification operations across a large number of consecutive packets to enable *path caching*. FastTrak [57] and vCRIB [54] exploit locality in flows to offload rules from the hypervisor to the limited memory on hardware NICs and switches. Similarly, software middleboxes (e.g., CoMb [74]) can offload packet classification and policy rules on to the NIC. A CoMb middlebox may also run different applications on the same hardware platform. A CoMb server might exploit burstiness by carefully scheduling applications for better CPU and cache utilization, for example, by running an individual application on an entire burst of packets, followed by the next application, instead of context-switching applications on per-packet basis. Jain and More, and recently, Sinha et al. [75] exploit the burstiness of TCP to schedule "flowlets" on multiple paths through the network. Wischik finds that traffic bursts can potentially inform the sizing of buffers in routers and gateways along the path [86]. Recently, Vattikonda et al. [82] proposed overlaying a data center network with a TDMA-based link arbitration scheme, which replaces Ethernet's shared medium model. With TDMA support, each host would periodically have exclusive access to the link. To maximally utilize that resource, each host would ideally send a burst of data. Porter et al. [63] rely on a similar TDMA scheme to implement an inter-ToR optical circuit switched interconnect.

3.3 Summary

Thus far, we have shown that traffic exhibits large bursts at sub-100 microsecond timescales, and these bursts are highly correlated with the size of TSO segments, disk read-ahead settings, and application send sizes. Our results indicate that irrespective of higher layer application behavior, packets come out of a 10-Gbps server in bursts due to batching. The bursty traffic has implications towards designing cost efficient packet processing devices. Moreover, these short-term bursts is an important result for REACToR, a hybrid electrical and circuit ToR with rapidly reconfiguring short circuit connections (chapter 7), because if a circuit is allocated an interval at a time, any instant when the instantaneous demand or burst does not fully saturate the link rate implies that circuit bandwidth is wasted. Presence of these bursts, means that, given a circuit-switch configuration the entire link bandwidth would be dedicated to servicing a single burst of traffic.

3.4 Acknowledgments

This work was supported in part by grants from the National Science Foundation (CNS-1314921 and CNS-1018808) and by generous research, operational and/or in-kind support from Microsoft, Google, and the UCSD Center for Networked Systems (CNS).

This chapter contains material as it appears in Kapoor, Rishi; Snoeren, Alex C.; Voelker, Geoffrey M. ; Porter, George. “Bullet Trains: A study of NIC burst behavior at microsecond timescales”, Proceedings of ACM CoNEXT, Santa Barbara, CA, December 2013. The dissertation author was the primary investigator and author on this paper.

Chapter 4

Importance of tail latency in data centers

In chapter 1 we looked at traffic pattern of bandwidth intensive applications at microsecond scales. Another class of applications that run inside data centers is latency sensitive applications such as Web search and Memcached. These applications or services generate a user response by accessing data across thousands of servers. For example, an end-user response may be built incrementally from dependent, sequential requests to networked services such as caches or key-value stores. Or, a set of requests can be issued in parallel to a large number of servers (e.g., web search indices) to locate and retrieve individual data items spread across thousands of machines Hence, the 99^{th} percentile of latency typically defines service level objectives (SLOs): when hundreds or thousands of individual remote operations are involved, the tail of the performance distribution, rather than the average, determines service performance. Being driven by the tail increases development complexity and reduces application quality [71].

In this chapter we present a systematic measurement study to understand the tail latency in the data center context. We discuss the effect of latency and high latency variation on two data center workload patterns — (1) Partition/Aggregate, (2) Dependent/Sequential traffic pattern and how high latency variation impacts the end-to-end performance and operation of data center applications. We use Memcached as an example of each of these communication patterns. Memcached is a popular, in-memory Key-Value (KV) store, deployed at Facebook, Zynga, and Twitter [17, 53]. Its simple API consists of operations that get, set, delete, and manipulate KV pairs. For high throughput, Memchached requests are typically issued using UDP [73].

4.1 The partition/aggregate pattern

In the Partition/Aggregate communication pattern, data is retrieved from a large number of servers in parallel prior to being combined into a response for the requesting service. An example of this pattern is a horizontally scaled Web search query that must access state from hundreds to thousands of inverted indices to generate the final response. The achievable service-level objective of an application relying on this pattern is limited by the slowest response generated, since all requests must complete before a response can be sent back to the user. In practice, this means that the latency seen by the end user approaches the tail latency of the underlying services. Here, the key insight is that increasing the number of servers increases the probability of hitting the tail latency more often, and hence increases the overall latency seen by the end user. We now show this straggler behavior both theoretically and experimentally.

Analysis: We first consider a client issuing a single request to each of S service instances in parallel. For simplicity, we assume the service time is an independent and identically distributed (i.i.d.) random variable with a normal distribution. Consider an S-length vector of the form:

$$\vec{v} =< N(\mu,\sigma), N(\mu,\sigma), ..., N(\mu,\sigma) > \tag{4.1}$$

where $N()$ is the normal distribution, and $\mu = 90\mu s$ and $\sigma = 50\mu s$ (these values are based on our observations of Memcached's latency, described in Section 5). We estimate service time by computing values of sets of i random variables, where i ranges from 1 to 100. For each set we compute the maximum over the values of the variables in the set, repeating each measurement five times to determine the latency and variance. Figure 4.1 shows the result.

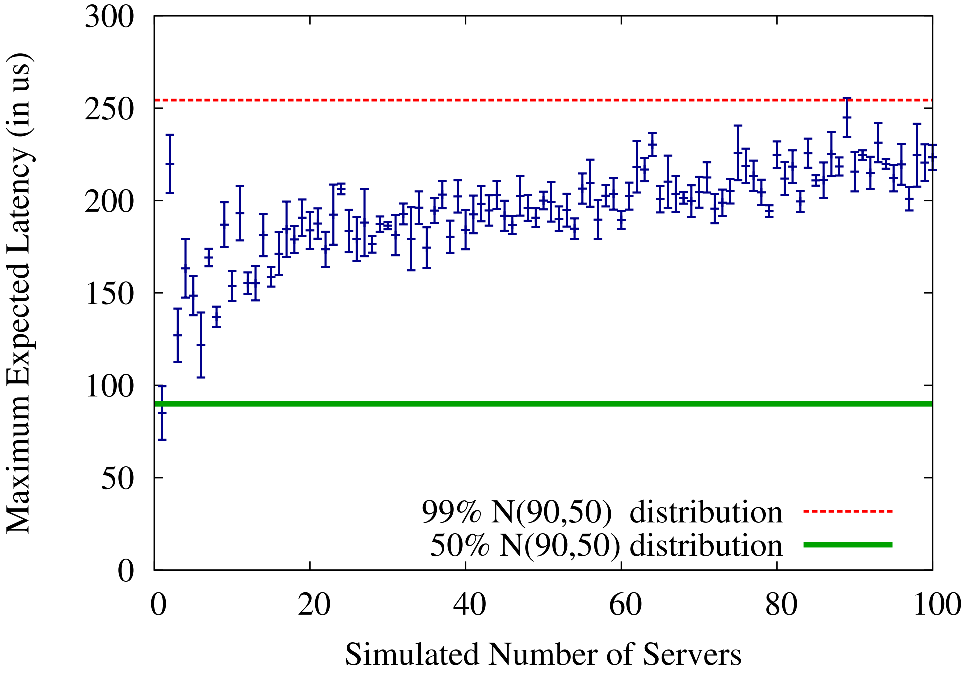


Figure 4.1. Predicted by probabilistic analysis. As the scale of the Partition/Aggregate communication pattern increases, latency increases due to stragglers.

As the number of servers increases, the maximum observed value in $\vec{v}$ increases as well. We also plot the 50^{th} and 99^{th} percentiles of the underlying $N(90,50)$ distribution. In this simulation, when the number of servers is small, the maximum expected latency is close to the mean of 100 μs (the 50^{th} percentile of the random variable). However, as S grows the maximum observed value approaches the 99^{th} percentile value of $254.25\mu s$. In this way, the end-to-end latency of the Partition/Aggregate communication pattern is driven by the tail-latency of nodes at scale.

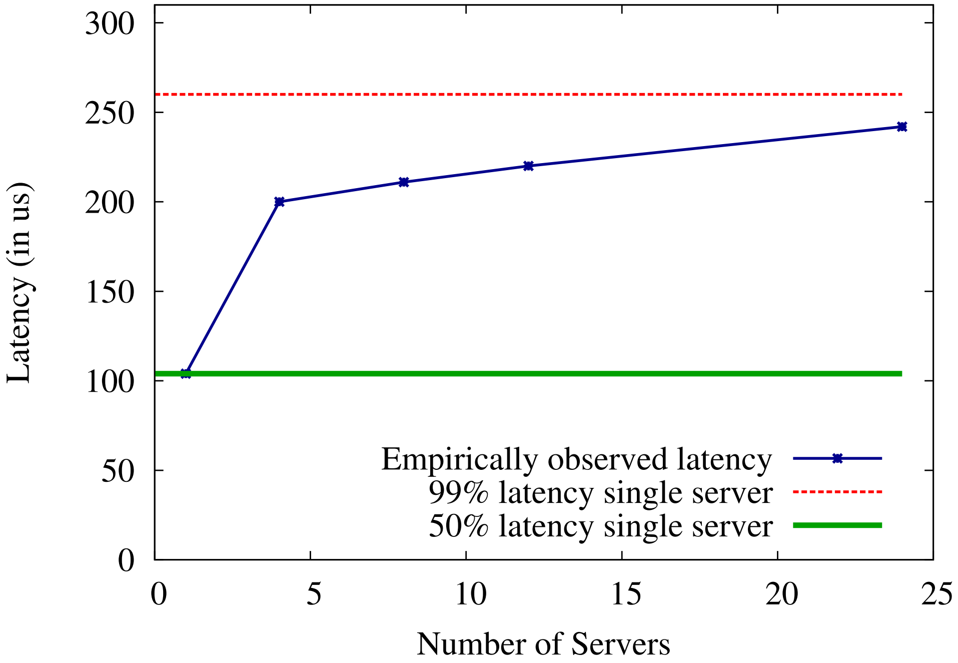


Figure 4.2. Empirically observed. As the scale of the Partition/Aggregate communication pattern increases, latency increases due to stragglers.

Experimental validation: To validate the above probabilistic analysis, we perform the following experiment on our testbed. We set up six Memcached clients, each on different machines, and measured the latency seen by one of these clients. Each client issues *S* parallel *get* requests to a set of *S* server instances (where *S* ranges from 1 to 24). Clients waits for response from all the servers before generating next set of requests. Each server instance runs on its own machine. In addition, we used the *memslap* load generator included with Memcached to generate requests uniformly distributed across the key-space at a low request rate, so as not to induce significant load on the servers.

Figure 4.2 shows the results of experiments and observed single-server median latency (approximately $100\mu s$) and the 99^{th} percentile of latency (approximately $255\mu s$). As expected, when issuing a single request to a single server the observed latency is nearly the 50^{th} percentile of service time. However, as *S* increases, the observed latency of the set of requests quickly approaches the long tail of latency, in this case just below the 99th percentile.

4.2 The dependent/sequential pattern

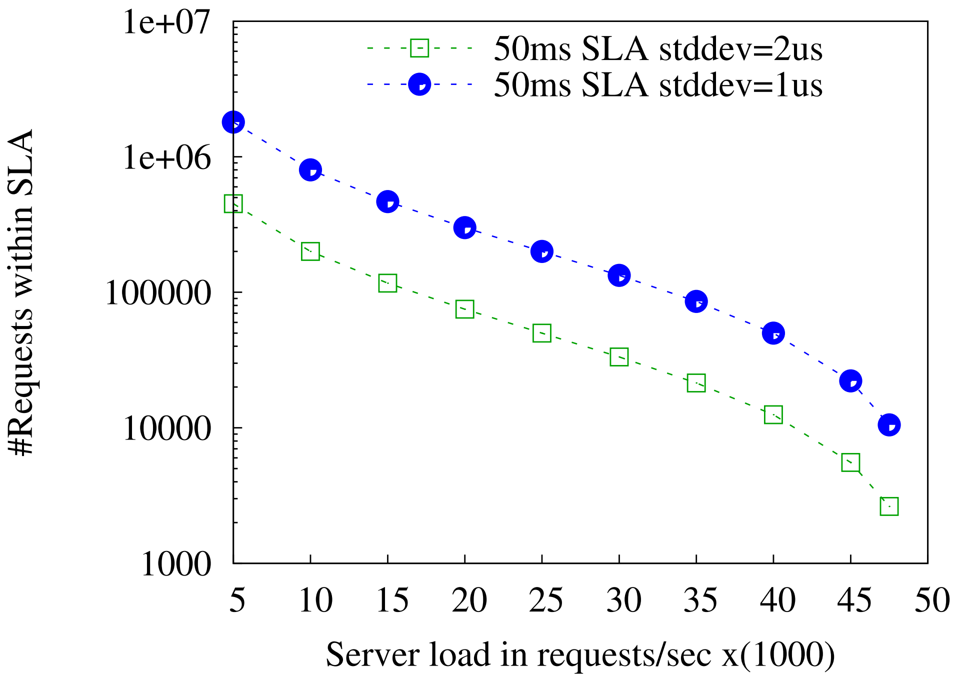


Figure 4.3. Predicted by queueing analysis. For the Dependent/Sequential communication pattern, the number of subservice invocations permitted by the developer to meet end-to-end latency SLAs depends on the variance of subservice latency.

Another communication pattern in data centers is the dependent/sequential workflow pattern, where applications issue requests one after another such that a subsequent request is dependent on the results of previous requests. Dependent/sequential patterns, for example, force Facebook to limit the number of requests that can be issued to build a user's page to between 100 and 150 [71]. The reason for this limit is to control latency, since a large number of sequential requests can add up to a large aggregate latency. With a large number of sequential requests the number of requests hitting the tail latency will

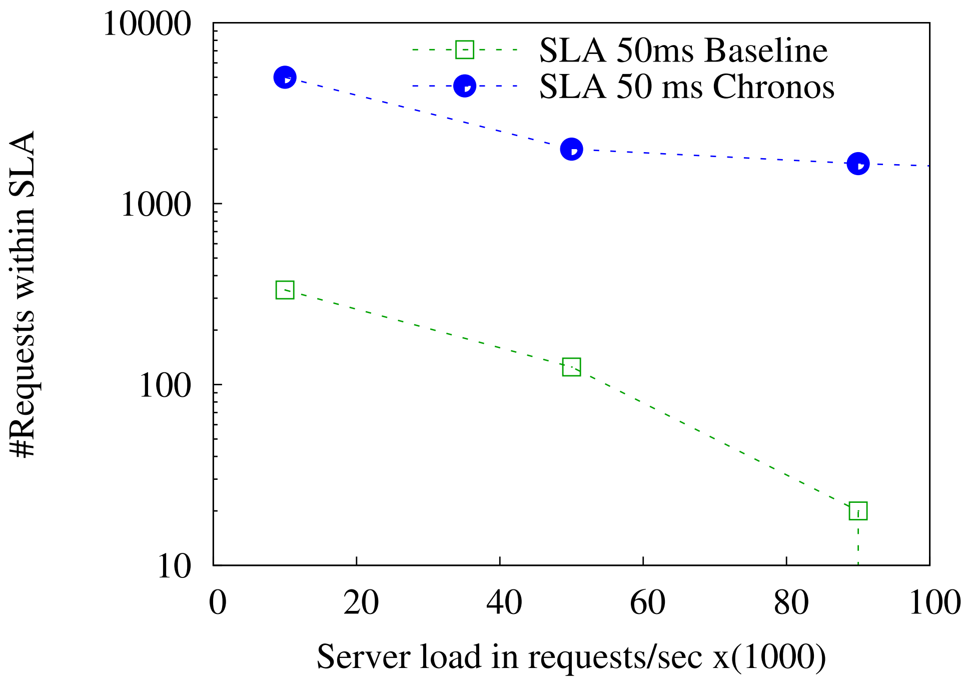


Figure 4.4. Empirically-observed. For the Dependent/Sequential communication pattern, the number of subservice invocations permitted by the developer to meet end-to-end latency SLAs depends on the variance of subservice latency.

also increase, thus lowering the number of otherwise possible sequential invocations. Another example of this pattern is search queries that are iteratively refined based on previous results.

In both cases, increasing the load on the subservices results in increased service time, lowering the number of operations allowed during a particular time budget. This observation is widely known, and in this subsection we show how it can be validated both through a queueing analysis as well as a simple microbenchmark.

Consider a simple model of a single-threaded server where clients send requests to the server according to a Poisson process at a rate λ. The server processes requests one at a time with an average service time of μ. Since the service time is variable, we model the system as an M/G/1 queue. Using the Pollaczek-Khinchine transformation [8], we compute the expected wait time as a function of the variance of the service time using

$$W = \frac{\rho + \lambda \mu Var(S)}{2(\mu - \lambda)} \tag{4.2}$$

where $\rho = \lambda / \mu$.

Based on this model, we can predict the service latency as a function of service load, mean latency, and the standard deviation of variance. To observe the effect of latency variation, we evaluated the model against $\sigma = 1$ (based on our observations of Memcached), and $\sigma = 2$ (representing a higher variance service). For each σ value, we use the model to compute the latency, and from that, we compute the number of service invocations that a developer can issue while fitting into a specified end-to-end latency budget, and plot the results in Figure 4.3. As expected, that budget is significantly reduced in the presence of increased latency variance.

To validate this model, we compare the predicted number of permitted service invocations to the actual number as measured with Memcached deployment in our testbed, shown in Figure 4.4. The experimental setup and experiments are described in detail in Section 6.2.2. Here, we measure the 99^{th} percentile of latency for both baseline Memcached as well as Memcached implemented on Chronos (CH) with uniform inter-arrival time and access pattern for requests. Each point in figure represents the number of service invocations permitted with the specified SLA, as a function of the server load, in requests per second.

4.3 Summary

The overall trends in these simple studies confirm the intuition that delivering predictable, low latency response requires not just a low mean latency, but also a small variation from the mean. We have shown that the end-to-end latency for the Partition/Aggregate communication pattern is driven by the tail-latency at scale. In the case of the Dependent/Sequential pattern, the tail latency determines the number of service invocations allowed within the SLO. Thus, it is important to reduce the variance in service latency in addition to bringing down overall latency.

4.4 Acknowledgements

This chapter in part, contains material as it appears in Kapoor, Rishi; Porter, George; Tewari, Malveeka; Voelker, Geoffrey M. ; Vahdat, Amin. “Chronos: Predictable Low Latency for Data Center Applications”, Proceedings of the ACM Symposium on Cloud Computing (SOCC), San Jose, CA, October 2012 The dissertation author was the primary investigator and author on this paper.

Chapter 5

Data Center Latency Characterization

In previous chapter we have shown that the end-to-end latency for different application communication pattern is driven by the tail-latency at scale. In this chapter, we give a detailed analysis of the main components contributing to the end-to-end latency in the data center applications. Understanding and analyzing sources of latency and latency variation is important to build performant and efficient applications.

We summarize the results in Table 5.1 and report the contribution of each component in the end-to-end latency. This includes one-way network latency for a request to reach from the client to the server, the latency at end-host server to deliver the request to the application and application latency for processing the request and sending the out the response from the server. As a concrete example, we further analyze the impact of server load and lock contentions due to concurrent requests on the Memcached server latency.

5.1 Sources of end-to-end application latency

Data center Fabric: The data center fabric latency is the amount of time it takes a packet to traverse the network between the client and the server. This latency can be further decomposed into propagation delay and in-switch delay. Within a data center, speed of light propagation delay is approximately 1 μs. Within each switch, the

Table 5.1. Latency sources in data center applications. Underlying operating system is the Debian Linux 2.6.28 kernel. †The network fabric latency assumes six switch hops per path and at most 2-3 switches congested along the path. Switch latency is calculated assuming 32 port switch with a 2MB shared buffer (i.e., 64KB may be allocated to each port). *Application latency is based on Memcached latency.

Component	Description	Mean latency (μs)	99 %ile latency (μs)	Overall share
DC Fabric	Propagation delay	<1	-	-
	Single Switch	1-4	40-60	1%
	Network Path†	6	150	7 %
End-host	Net. serialization	1.3	1.3	1.4 %
	DMA	2.6	2.6	3 %
	Kernel (incl. lock contention)	**76**	**1200-2500**	**86-95** %
Application	Application*	2	3	2 %
	Total latency	88	1356-2656	100 %

switching delay is approximately 1–4 μs. Low-latency, cut-through switches further reduce this packet forwarding latency to below one microsecond. A packet from client to server typically traverses 5–6 switches [4]. A packet can also suffer queueing delay based on prevailing network congestion. We calculate the queueing delay by measuring the additional time a packet waits in switch buffers. Typical commodity silicon might have between 1–10MB buffers today for 10Gbps switches. However, this memory is shared among all ports. So for a 32-port switch with relatively even load across ports and with 2MB of combined buffering, approximately 64KB would be allocated to each port. During periods of congestion, this equates to an incoming packet having to wait for up to 50 μs (42 1500-byte packets) before it can leave the switch. If all buffers along the six hops between the source and destination are fully congested, then this delay can become significant. Several efforts described in Section 2.3.4 aim to minimize congestion and thus latency. We expect that, in the common case, the networks paths will be largely uncongested. While in network bottlenecks such as delay in data center fabric are outside the scope of this effort, the value of Chronos is that it addresses the key latency bottlenecks in the end-host to deliver low-latency services.

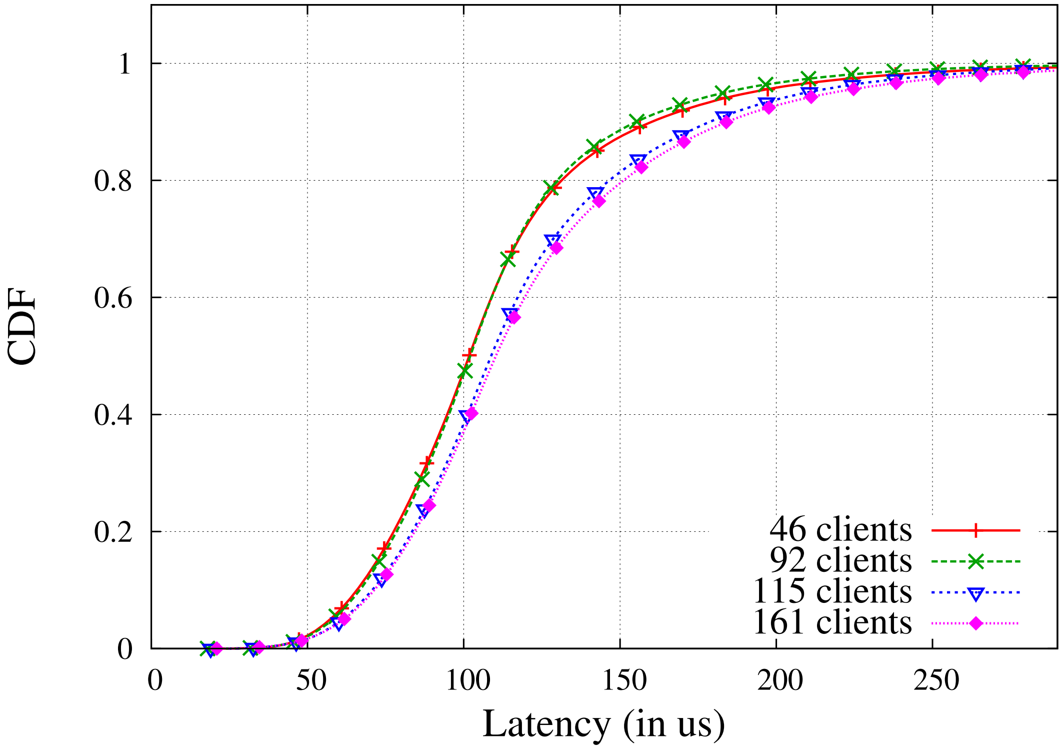


Figure 5.1. Memcached latency distribution at 30% (low) utilization.

End-host: End-host latency includes the time required to receive and send packets to and from the server NIC, as well as delivering them to the application. This time includes the latency incurred due to network serialization, DMA the packet from the NIC buffer to an OS buffer, and traversing the OS network stack to move the packet to its destination user-space process.

To understand the constituent sources of end-host latency under load, we profile a typical Memcached request. We issued 20,000 requests/second to the server, which is approximately 2% network utilization in our testbed. We instrumented Memcached 1.6 beta and collected timestamps during request processing. To measure the server response

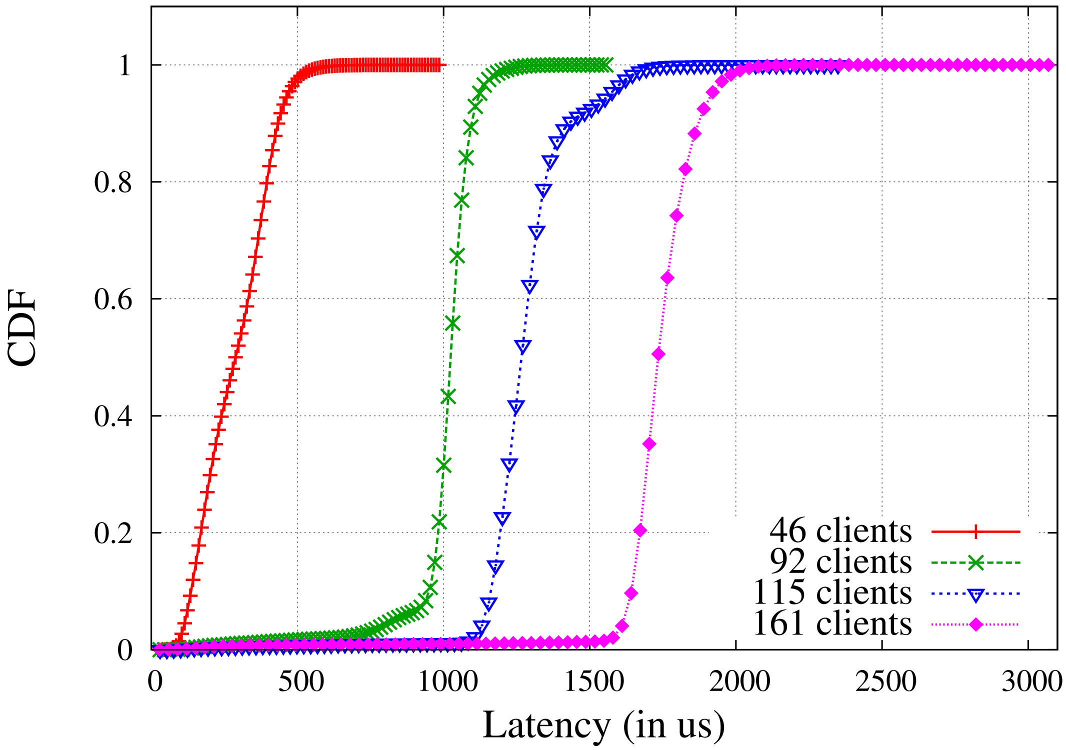


Figure 5.2. Memcached latency distribution at 70% (high) utilization.

time, we installed a packet mirroring rule into our switch to copy packets to and from our server to a second measurement server running Myricom's Sniffer10G stack, delivering precise timestamps for a 10Gbps packet capture (at approx. 20ns resolution). Section 6.2 presents full details on the testbed setup.

A median request took $82\mu s$ to complete at low utilization, with that time divided across the categories shown in Table 5.1. *Network Serialization latency* is based on a 100B request packet and a 1500B response at 10Gbps. *DMA latency* is the transfer time of a 1600B (request and response) calculated assuming a DMA engine running at 5GHz.

Application: This is the time required to process a message or request, perform the application logic, and generate a response. In the case of Memcached, this includes the time to parse the request, look up a key in the hash table, determine the location of value in memory and generate a response for the client. We measured the Memcached application latency by wrapping timer calls around the application. We record the start time of this measurement immediately after the socket *recv* call is issued; the end time is measured just before the application issues the socket *send* call. The application latency in Memcached is $2\mu s$. In Section 5.2 we discuss other factors that contribute to application latency, including application thread lock contention.

The remainder of the time between the observed request latency and the above components includes the kernel networking stack, context switch overhead, lock contention, kernel scheduling overhead, and other in-kernel, non-application end-host activity. The contribution of kernel overhead alone accounts for more than 90% of the end-host latency and approximately 85% of end-to-end latency. In the next section, and in rest of the paper, we focus our efforts on understanding the effect of kernel latency on the end-host application performance, aiming to reduce this important and significant component of latency.

5.2 End-to-end latency in Memcached

In this section we further analyze Memcached latency. We show how increasing the load at the server results in queueing of pending requests in the kernel which significantly increases the tail latency. We further show that lock contention for processing concurrent requests also results in significant latency variation.

Effect of server load: To measure Memcached performance, we use a configurable number of Memslap clients [1], which are closed-loop (i.e., each client sends a new request only after receiving the response from the previous request) load generators included with the Memcached distribution to send requests to a Memcached server with four threads. Each client is deployed on its own core to lower measurement variability. We observe that Memcached can support up to 120,000 requests/second with sub millisecond tail latency. We next subject the Memcached server to a fixed request load, and observe the distribution of latency. We evaluated the server at a low request load of 40,000 requests per second, which is approximately 30% of the server's maximum throughput, and also at a high load of 90,000 requests per second, or about 70% of its maximum throughput. On each of the 23 client machines, we reserve one CPU core for Linux, leaving seven for client instances, which means we can support up to 161 clients.

At low server utilization (30%), increasing the number of clients had little effect on distribution of latency as shown in Figure 5.1. By increasing the number of clients we increase the number of concurrent requests at the server, even though load offered by each client drops. Most responses completed in under 150 μs, with the tail continuing up to approximately 300 μs, shown in Figure 5.1. This corresponds to lower levels of load at which developers run their services to ensure low tail-latency. However, at high server utilization (70%), increasing the number of clients had a pronounced effect on observed latency. High load resulted in a significant latency increase as the number of clients increased, reaching a maximum at about 2,000μs, shown in Figure 5.2. These measurements aid our understanding of current practices of running services at low levels of utilization. Operating these services at higher utilization necessitates reining in the latency outliers.

Request queueing in the application plays a significant role in the latency increase. Two sources of this queueing are variance in kernel service time and an increase in lock contention within the application due to an increase in concurrent requests. Profiling CPU cycles spent during the experiment shows that the bulk of the time is spent copying data and context switching.

Lock contention: We used the mutrace [55] tool during runtime to validate this last point and saw a significant amount of lock contention. We evaluated a Memcached instance with concurrent requests from 20 memslap clients and found that more than

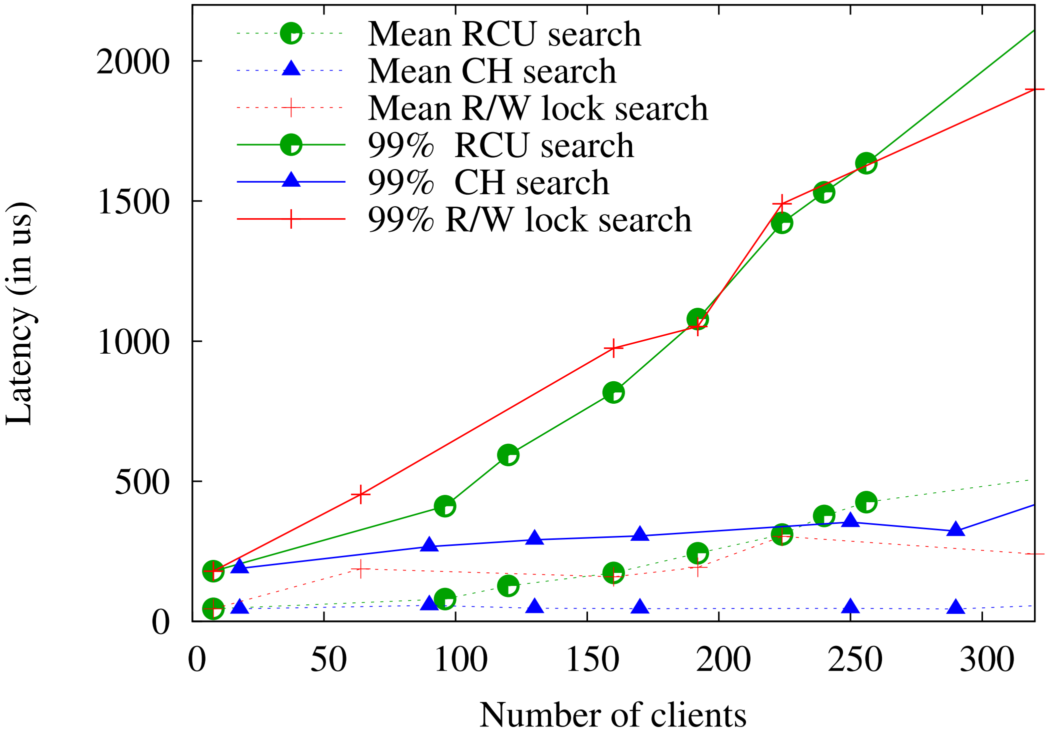


Figure 5.3. Web search latency of single Index server.

50% of lock requests were contested, with that contested time accounting for about a third of the overall experiment duration. We found that the source of this lock contention in Memcached was a shared hash table protected by a pthread lock. This lock must be acquired both for update as well as lookup operations on the table. With pthread locks (used by Memcached), contention not only induces serialization, but must also be resolved in the kernel, adding further delay and variance to observed latency.

To quantify lock overhead, we modified a Memchached based Web search application to use two different synchronization primitives, (1) read/write locks and (2) an RCU (read copy update) mechanism [81] in place of the conventional pthread system locks in Memcached. These synchronization primitives are more efficient for the read-dominant workloads that are common in applications like key-value stores (where the number of get requests is much larger than set requests) and search (where index-update is less frequent than index-lookup).

In addition to using the new locking primitives, we also modify the applications to use user-level networking APIs to bypass the kernel and eliminate kernel overheads in latency. We describe the user-level APIs in more detail in Section 6.1, but for illustration we can assume that use of these APIs removes kernel overhead completely. Bypassing the kernel with user-level APIs allows us to quantify the overhead caused due to application lock contention alone. For evaluation, we vary the number of memslap clients that send requests to the modified Memcached instances. We used 10-byte keys and 1400-byte values with a get/set requests ratio equal to 9:1 as suggested in [28]. Figure 5.3 shows the results of this experiment (the 99% and Mean CH search curves correspond to the Chronos results and can be ignored for now). Here, we see that even with an implementation based on read/write locks and RCU, latency remains high. Read/write locks do not scale because the state associated with them (number of readers) still needs to be updated atomically. For a small number of concurrent clients RCU performs well but as load increases there is significant variation in latency. Note that the performance of these synchronization primitives would be reduced if the workload pattern shifted towards a more write-heavy demand.

5.3 Summary

Our observations, summarized in Table 5.1, indicate that the operating system accounts for more than 90% of end-host latency at low and high requests loads. To eliminate this kernel and network stack overhead, we studied user-level, kernel bypass, zero-copy network functionality. These APIs are known to minimize latency by eliminating the kernel from the critical message path, and thus avoiding overheads due to multiple copies and protection domain crossings [20, 25, 65, 84]. While user-level networking removes kernel overhead, its usage alone is not sufficient to achieve substantial performance improvements. In fact, use of these APIs exposes two new bottlenecks in the system: lock contention and high load hotspots, both of which limit application performance.

5.4 Acknowledgements

This chapter in part, contains material as it appears in Kapoor, Rishi; Porter, George; Tewari, Malveeka; Voelker, Geoffrey M. ; Vahdat, Amin. “Chronos: Predictable Low Latency for Data Center Applications”, Proceedings of the ACM Symposium on Cloud Computing (SOCC), San Jose, CA, October 2012 The dissertation author was the primary investigator and author on this paper.

Chapter 6

Chronos: Predictable Low Latency for Data Center Applications

Based on our observations in chapter 4 and chapter 5, we propose Chronos, a communication framework that leverages both kernel bypass and NIC-level request dispatch to deliver predictable low latency for data center applications. Chronos directs incoming requests to concurrent application threads in a way that drastically reduces, and in some cases eliminates, application lock contention. Chronos also provides an extensible load balancer that spreads incoming requests across available processing cores to handle skewed request patterns while still delivering low-latency response time.

Our evaluation shows that Chronos substantially improves data center application throughput and latency. We show that Memcached implemented on Chronos, can support 200,000 requests per second with a mean operation latency of $10\mu s$ with a 99^{th} percentile latency of only 30 μs, a factor of 20 lower than unmodified Memcached. We find similar benefits for Web search and the OpenFlow controller.

6.1 Design Goals

Our goal is to build an architecture with these features:

1. **Low mean and tail latency:** Achieve low predictable latency by reducing the

overhead of handling sockets and communication in the kernel. Reducing the application tail latency improves the latency predictability and the application performance.

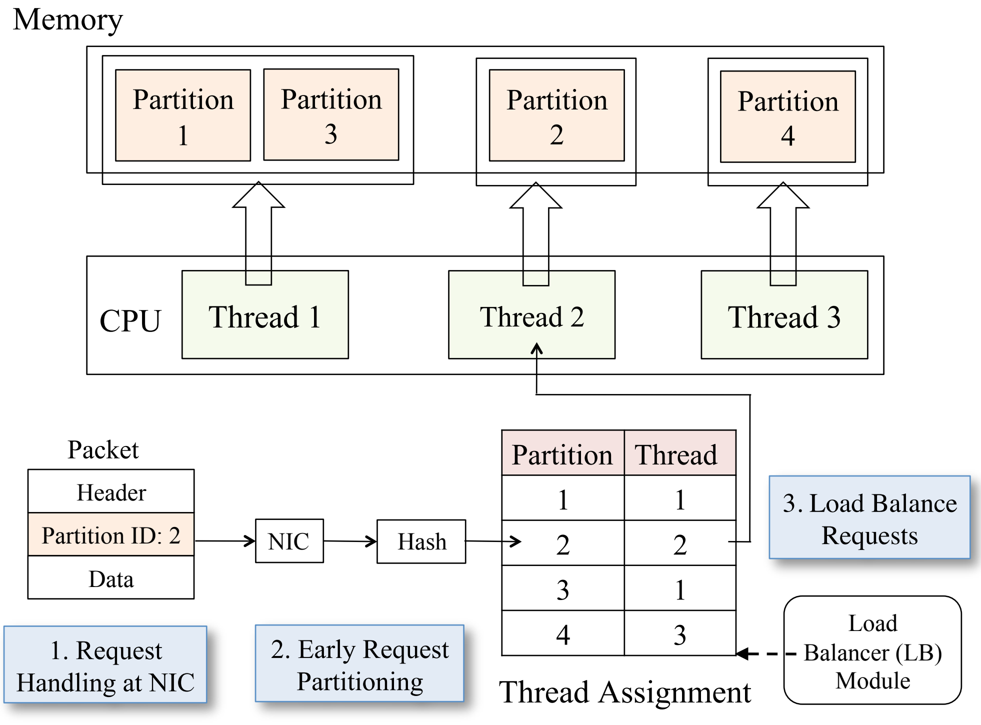
2. **Support high levels of concurrency with reduced or no lock contention:** Reduce or eliminate application lock contention by partitioning requests across application threads within a single end-host.

3. **Early request assignment and introspection:** Partition incoming client requests as early as possible to avoid application-level queue build-up due to request skew and application lock contention.

4. **Self tuning:** Dynamically load balance requests within a single node across internal resources to avoid hotspots and application-level queueing, without assuming apriori knowledge of the incoming request pattern and application behavior.

6.1.1 Design and implementation

We now describe the design of Chronos (shown in Figure 6.1). Chronos partitions application data to allow concurrent access by the application threads (described in Section 6.1.1). It maintains a dynamic mapping between application threads and data partitions in a lookup table, and when a packet arrives at the server, Chronos examines the partition ID in the application header and consults the lookup table to steer the request to the proper application thread. The Chronos load balancer periodically updates the mapping between partitions and application threads to balance the load on each thread such that the request response time is minimized. In Chronos, requests are demultiplexed between application threads early, in the NIC, to avoid lock contention and multiple copies. At a high level, the Chronos request servicing pipeline is carried out in three stages: (1) request handling, (2) request partitioning and (3) load balancing.



Partition	Thread
1	1
2	2
3	1
4	3

Figure 6.1. Chronos system overview.

Request handling

As described in Section 5, a major source of latency in end host applications is the operating system stack. Chronos eliminates the latency overhead introduced due to kernel processing by moving request handling out of the kernel to user-space by using zero-copy, kernel-bypass network APIs. These APIs are available from several vendors and can be used with commodity server NICs [56, 76, 77].

We now explain one possible way of implementing user-level networking. When the NIC driver is loaded, it allocates a region of main memory dedicated for storing incoming packets. This memory is managed as send and receive ring buffers, called NIC queues. To bypass the kernel, an application can request an exclusive handle to one or more receive ring buffers for its different threads. The receive ring buffers are mapped to addresses in the application address space. Outgoing packets from the application are enqueued into a selected ring, and are sent on the wire by the NIC. The incoming packets at the NIC are classified to a receive ring based on the output of a hash function. This classifying function can be implemented in the hardware or in software. Though in-NIC request classification will be most efficient, it is less flexible than a software classifier. Chronos is not tied to any specific NIC implementation for user-level networking as long as it can correctly classify the incoming packets and assign them to the right application thread. For hardware classification, one could extend the receive-side scaling (RSS) feature in the NICs such that it hashes packets into rings based on a custom offset instead of hashing on the fixed 5-tuple in the packet header.

For prototyping Chronos, we use a custom hash function implemented in user space for request classification. The custom hash function enables deep packet inspection and arbitrary processing over the packet contents by executing the hash function on any one of the CPU cores. This function works by registering a C function with the NIC API, and then when a new packet arrives, the NIC will call the function, passing it a pointer to the packet and the packet length. This function returns the receive ring buffer id which the packet should be classified to. Note that there is no additional copying involved. However, software hashing has performance cost as it may cause cache misses. This is because the custom hash function would read the packet header first and then assign it logically to a ring buffer. The packet may then be processed by an application thread on a different CPU core, which may not share an L2 cache with the classifying core. For our implementation, the performance penalty due to user space processing was outweighed by the latency incurred in the kernel. For the simple custom hash functions we implemented the execution overhead is in nanoseconds, less than the packet inter-arrival times for 10 Gbps links.

Finally, note that the application is not interrupted as the packets arrive at the server. Instead, it must poll the receive ring buffer for new packets using *receive()*. For

Chronos, we have a dedicated thread monitoring the NIC queues that registers packet reception events with the applications.

Request partitioning

Bypassing the kernel significantly reduces the latency, since a request can now be delivered from the NIC to the application in as low as 1-4 μs. However, this reduction in packet transfer latency exposes new application bottlenecks namely lock contention, core overloading or processing hot-spots due to skewed requests. These bottlenecks are responsible for significant variation in latency causing unpredictability. A classic approach to reducing lock contention is to separate requests that manipulate disjoint application state as early as possible. Chronos uses this approach and minimizes shared state with static division of the state into disjoint partitions that can be processed concurrently. For instance, in case of Memcached, we replace a single centralized hash table with the entire keyspace and associated slab class lists with *N* hash tables and slab class lists with smaller regions of the keyspace. Each of these *N* hash tables represents a partition and can now be assigned to a hardware thread for concurrent processing. A single thread can handle multiple data partitions.

With partitioned data, we now need to send each request to the thread handling that partition. Chronos uses a classifying function (described in Section 6.1.1) to examine the application header for the partition ID and steering the request to the receive ring buffer of the thread which handles the data partition for the request. While it is possible to add a new field (partition ID) to the application header to steer requests to the appropriate application threads, we choose instead to overload an existing field. In case of Memcached, we rely on the *virtual bucket*, or *vBucket* field, which denotes a partition of keyspace. For the search application we use the search term itself, and for the OpenFlow controller we use the switch ID.

The partitionable data assumption fits well for classes of applications like key-value stores, search, and OpenFlow. Handling requests for data from multiple partitions is an active area of research [45], and one we hope to study in future work.

Extensible load balancing

The end-host should be able to handle large spikes of load, with multiple concurrent requests, while running the underlying system at high levels of utilization. While static request partitioning helps in reducing lock contention, it could still lead to hot-spots where a single thread has to serve a large number of requests. To this end, we present a novel load balancing algorithm that dynamically updates the mapping between threads and partitions such that the incoming requests are equally distributed across the threads.

We now describe the load balancing mechanism. Chronos uses a classifier based on the partition ID field in the application header, and a soft-state table to map the partition ID field to an application thread. To reduce lock contention, the partition-to-thread mapping should ensure that each partition is exclusively mapped to a single thread. The load balancing module periodically updates the table based on the offered load and popular keys. For simplicity, assume that the Chronos load balancer measures the load on a data partition as a function of the number of incoming requests for that partition. This is true for key-value stores when each request is identical in terms of time required for processing the request (table lookup) but not for applications like in-memory databases. In general, the load on a partition is representative of the expected time taken to process the assigned requests. The number of requests served for each partition is maintained in user space for each ring buffer. A counter is updated by the classifying function while handling requests, and the load balancer could optionally be extended to measure the load in other ways as well. The load on a thread is the total load on all partitions assigned

Algorithm 1. *Chronos* Load Balancer updates partitionID to thread mapping based on load offered in last epoch.

```
$IdealLoad = totalEpochLoad/totalThreads$
for all $k \in \{totalThreads\}$ do
    $threadLoadMap[k] = 0$
end for
for all $v \in partitionID$ do
    $t = epochMap.getThread(v)$
    if $threadLoadMap[t] \leq IdealLoad$ then
        currentEpochMap.assign(v, t)
        $threadLoadMap[t].add(v.load)$
    else
        for all $k \in \{totalThreads - \{t\}\}$ do
            if $threadLoadMap[k] \leq IdealLoad$ then
                currentEpochMap.assign(v, k)
                $threadLoadMap[k].add(v.load)$
                break
            end if
        end for
    end if
end for
$epochMap = currentEpochMap$
```

to a thread.

The Chronos load-balancing algorithm divides time into epochs, where each epoch is of maximum configurable duration T. The load balancer maintains a mapping of each partition to an application thread in the epoch, *epochMap*, along with per-partition load information. The load balancer also maintains a separate map for measuring thread load, *threadLoadMap* which indicates the number of requests served by an application thread in the current epoch.

The load balancing algorithm greedily tries to assign partitions to the least loaded thread only if the thread to which partition is already assigned is overloaded with requests. This is to avoid unnecessary movement of partitions across threads. When the application starts, the Chronos load balancer initializes the table with a random mapping of partition

IDs to threads. Algorithm 1 shows psuedocode for the *Chronos* load-balancer module. A new epoch is triggered when the duration T elapses. At the start of a new epoch, the load balancer computes the new mapping as described in Algorithm 1. The load balancer computes the total load in the last epoch and divides that by the number of threads to obtain the ideal load each thread should serve in the next epoch, under the assumption that load distribution will remain the same. In each epoch, it initializes the load for each thread to be zero. It then iterates through all partitions, checking if the thread it is currently assigned to can accommodate the partition load or not. If not, the algorithm assigns the partition to the first lightly loaded thread.

For the proposed algorithm to work effectively, the number of partitions should be at least the number of cores available across all of the application instances. Note that Chronos load balancing does not add to cache pollution that might happen due to sharing of partitions among threads. In fact, the baseline application will have lower cache locality given that all of its threads access a centralized hash table. While the proposed load balancing algorithm tries to distribute the load uniformly on all threads, Chronos can also be used with other load balancing algorithms which optimize for different objectives.

Note that concurrent access to the partitioned data is still protected by a mutex to ensure program correctness, however the partitioning function ensures that there is a serialized set of operations for a given partition. The only time that two application threads might try to access the same partition is during the small windows where the load balancing algorithm updates its mapping. This remapping can cause some requests to follow the new mapping, while other requests are still being processed under the previous mapping. We will show in the evaluation that this is a relatively rare event, and for reasonable update rates of the load balancer, would not affect the 99^{th} percentile of latency.

6.1.2 Application case studies

Chronos does not require rewriting the application to take full advantage of its framework. Chronos requires only minor modifications to the application code for using the user-level networking API. To demonstrate the ease of deploying Chronos, we port the following three data center applications to use Chronos and evaluate the improvement in their performance.

Memcached: Rather than building a new key-value store, we base Chronos-Memcached (Chronos-MC) on the original Memcached codebase. Chronos-MC is a drop-in implementation of Memcached that modifies only 48 lines of the original Memcached code base, and adds 350 lines. These modifications include support for user-level network APIs, for the in-NIC load balancer, and for adding support for multiple partitions.

Web Search: Another application we consider is Web search, a well-studied problem with numerous scalable implementations [24, 35]. We choose Web search since it is a good example of a horizontally-scalable data center application. Web search query evaluation is composed of two components. The first looks up the query term in an inverted-index server to retrieve the list of documents matching that term. The second retrieves the documents from the document server. The inverted index is partitioned across thousands of servers based on either document identifier or term. For Chronos-WebSearch (Chronos-WS), we implement term-based partitioning. We wrote our own implementation of Web search based on Memcached.

It is important that Web search index tables are kept updated, and so modifications to them are periodically necessary. One approach is to create a completely new copy of the in-memory index and to then atomically flip to the new version. This would impose a factor of two memory overhead. Another option is to update portions of the index in place, which requires sufficient locking to protect the data structures. We implemented an index server using read/write locks and UNet APIs. The index server maintains the index-table as search term and associated documents IDs, as well as word frequency and other related information. We also implemented a version of the index server with an RCU mechanism from an open-source code base provided by the RCU authors [81]. We modified it to work with the UNet APIs. Chronos-WS further divides the index server table into several partitions based on terms for efficient load balancing.

OpenFlow Controller: We also implemented an OpenFlow controller application on Chronos (Chronos-OF) using code provided by [59]. This application is different from the Memcached and Web search applications since it is typically not horizontally scaled in the same way as these other applications. However, given that the OpenFlow controller can be on the critical path for new flows to be admitted into the network, its performance is critical, even if the entire application is only deployed on a single server. This application receives requests from multiple switches and responds with forwarding rules to be inserted in the switch table.

6.2 Evaluation

In this section we evaluate the Chronos-based Memcached, Web server and OpenFlow controller using micro and macro-benchmarks. Overall, our results show that:

- Even with Memcached running on the MOSBENCH [22] kernel with an efficient network stack, the tail latency is still high. This justifies the use of kernel bypass networking APIs to deliver predictable low latency.

- Chronos-MC exhibits up to 20x lower mean latency compared to stock Memcached for a uniform request arrival rate of 200,000 requests/sec. For bursty workloads, it reduces the tail latency by 50x for a request rate of 120,000 requests/sec. Reduced tail latency improves the latency predictability and application performance.

- Chronos-MC can effectively scale up to 1 million requests/sec taking advantage of load balancing across concurrent threads.

- Chronos-WS achieves an improvement of 2.5x in mean latency as compared to baseline Web Server application that uses Read/Write locks.

- Chronos-OF achieves an improvement of 12x in mean latency as compared to baseline OpenFlow application.

We now describe our experiment setup, the workloads we use, and performance metrics we measure.

Testbed: We deployed Chronos on 50 HP DL380G6 servers, each with two Intel E5520 four-core CPUs (2.26GHz) running Debian Linux with kernel version 2.6.28. Each machine has 24 GB of DRAM (1066 MHz) divided into two banks of 12 GB each. All of our servers are plugged into a single Cisco Nexus 5596UP 96-port 10 Gbps switch running NX-OS 5.0(3)N1(1a). This switch configuration approximates the ideal condition of nonblocking bandwidth on a single switch. We do not focus on network sources of latency variability in this evaluation. Each server is equipped with a Myricom 10 Gbps 10G-PCIE2-8B2-2S+E dual-port NIC connected to a PCI-Express Gen 2 bus. Each NIC is connected to the switch with a 10 Gbps copper direct-attach cable. When testing against kernel sockets, we use the myri10ge network driver version 1.4.3-1.378 with interrupt coalescing turned off. For user-level, kernel-bypass experiments we use the Sniffer10G driver and firmware version 2.0 beta. We run Memcached version 1.6 beta, configured to use UDP as the transport layer protocol, along with support for binary protocol for efficient request parsing and virtual buckets for enabling load balancing.

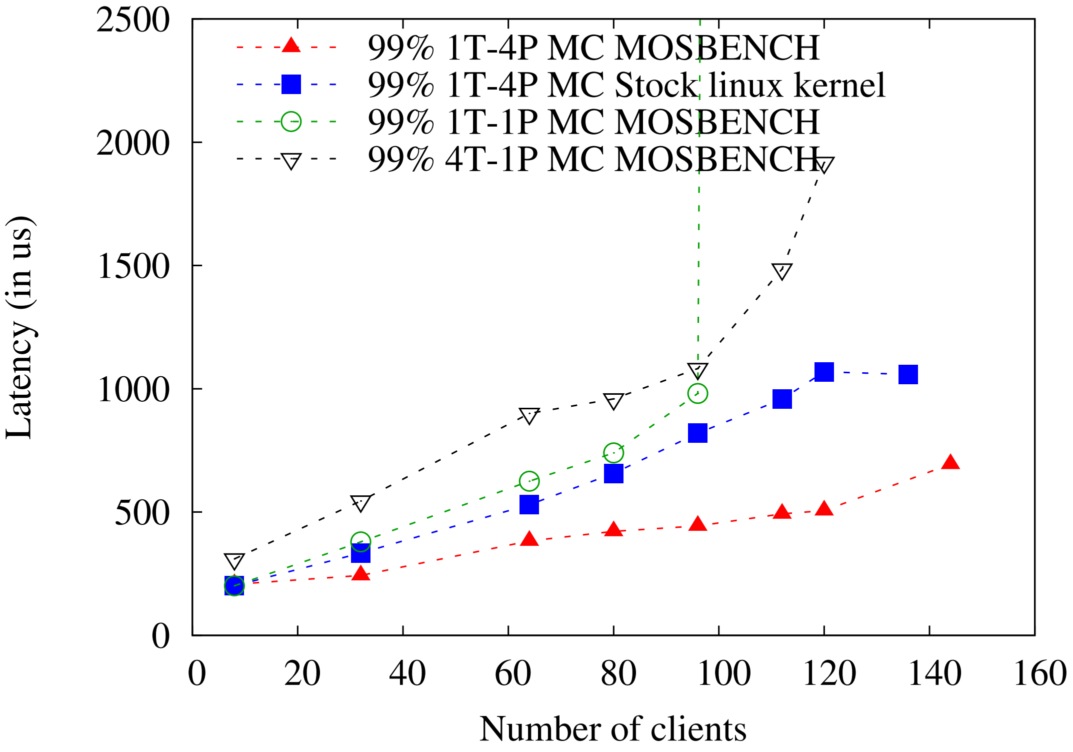


Figure 6.2. Legend: *n*T-*m*P stands for *n* thread *m* processes of Memcached(MC). Shown is the tail latency for one and four threads (1T and 4T) running in either one process or four processes (1P or 4P).

Metrics and Workloads: Like any complex system, the performance observed from Memcached and Chronos is heavily dependent on the workload, which we define using the following metrics: 1) request rate, 2) request concurrency, 3) key distribution, and 4) number of clients. The metrics of performance we study for both systems are the 1) number of requests per second served, 2) mean latency distribution, and 3) 99^{th} percentile latency distributions. To evaluate baseline Memcached and Chronos under realistic conditions, we use two load generators. The first, Memslap [1], is a closed-loop benchmark tool distributed with Memcached that uses the standard Linux network stack. It generates a series of *get* and *put* operations using randomly generated data. We configure it to issue 90% get and 10% put operations for 64-byte keys and 1024-byte values since these values are representative of read-heavy data center workloads [28]. For the results that follow, we found that varying the key size had a minimal effect on the relative performance between Chronos and baseline Memcached. The second load generator is an open-loop load program (i.e client generates requests at a fixed rate irrespective of pending previous requests) we built in-house using low-latency, user-level network APIs to reduce measurement variability. Each instance of this second load generator issues requests at a configurable rate, up to 10Gbps per instance, with either uniform or exponential inter-arrival times. The KV-pair distribution used by the tool is patterned on YCSB [28]. Note that the latency numbers reported in figures generated by the closed-loop clients are higher by 50–70 μs compared to open loop clients since closed loop clients also report the kernel and network stack latency. For Chronos, we run the load-balancer every $50\mu s$, unless specified otherwise.

6.2.1 Memcached on an optimized kernel

We examine the latency of different configurations of Memcached instances – i) one single threaded, ii) one multi threaded (with four threads) and iii) multiple single threaded processes (four processes each running on its own core) – using the MOSBENCH kernel (pk branch) with an efficient network stack. The multi threaded Memcached incurs intra-thread lock contention, while the single threaded and multi-process configurations are free of intra thread lock contention. However, multiple single threaded Memcached processes can support more clients as compared to single threaded instances.

To measure the performance of these different configurations we use a configurable number of Memslap clients, each deployed on its own core to lower the measurement variability. A Memslap client opens a socket connection to one of the four Memcached process. While running in single threaded mode, and thus free of intra-thread

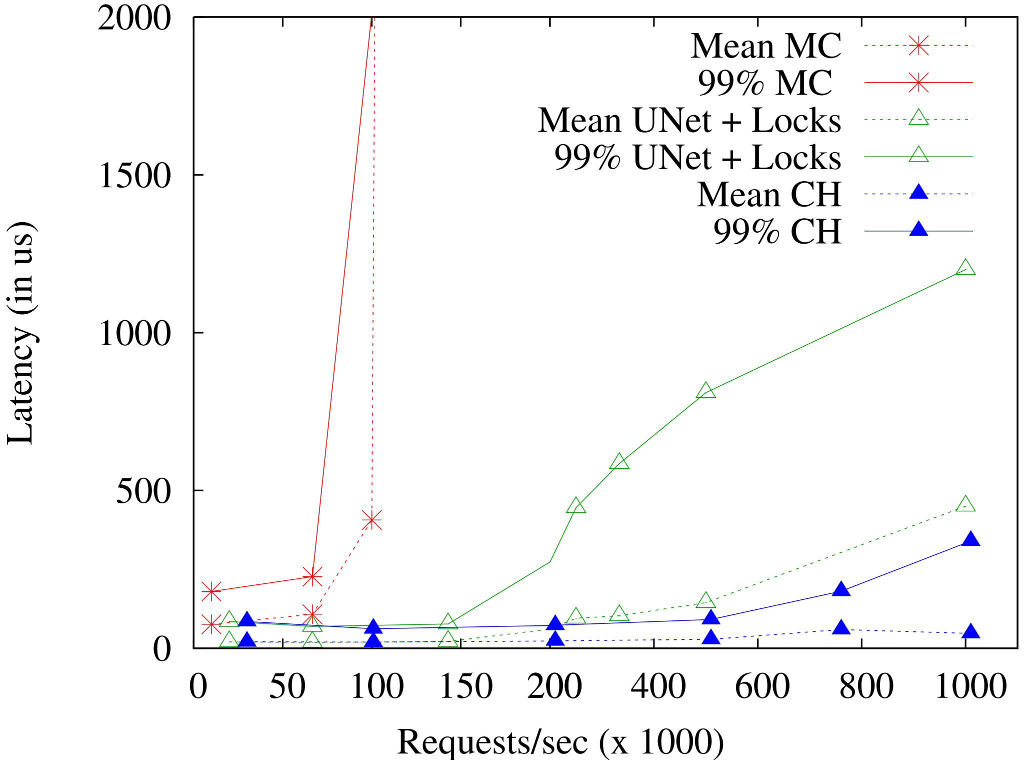


Figure 6.3. Latency of baseline Memcached (MC), Memcached with user-level network APIs (UNet locks), and Chronos (CH) with 10 open loop clients.

resource contention, we expect the single threaded, multiple process Memcached latency and variance to be lower than multi-threaded instance on MOSBENCH. Figure 6.2 shows our results. For comparison, we also plot the performance of Memcached with the stock linux kernel. Our results show that even with the optimized MOSBENCH kernel, the 99^{th} percentile latency for four single threaded multi-process configuration is as high as 810 μs with 140 clients (35 clients/process), indicating that the kernel's contribution to the tail latency is significant despite kernel optimizations and a lack of application lock contention.

6.2.2 Uniform request workload

In this subsection we show that Chronos-MC reduces the mean application latency by a factor of 20x as compared to baseline Memcached for a workload with uniform inter-arrival time and access pattern for requests. Chronos-MC also outperformed a Memcached implementation that only leveraged user-level networking but no other Chronos feature (request partition or load balancing). We started instances of the three different Memcached implementations with four threads each. We also instantiated 10 client machines running our custom open-loop load generator utilizing user-level network APIs. Each client issues requests at a configurable rate, measuring the response time as perceived by the client as well as any lost responses. The server is pre-installed with 4GB of random data, and clients issue requests from this set of keys using a uniform distribution with uniform inter-request times. We use 1KB values and 64 byte keys in a 9:1 ratio of gets to sets. To avoid overloading the server beyond its capacity, each client terminates when the observed request drop rate exceeds 1%.

Figure 6.3 shows the results for this experiment. While baseline Memcached supports up to approximately 120,000 requests per second before dropping a significant number of requests, Chronos supports a mean latency of about 25 μs up through 500,000 requests per second and rises just above 50μs at 1M requests per second. The Memcached instance with just the socket API replaced with the user-level kernel API not only has higher mean latency, but the variation of latency is significantly higher, as shown by the 99th percentile, indicating that reducing variability in the network stack, operating system, and application are all important to reduce tail latency.

We also evaluate the performance of Chronos-MC with a larger number of closed loop clients. We instantiated eight client Memslap processes on each physical client machine, and scaled up to 50 client machines. As shown in Figure 6.4, we see that

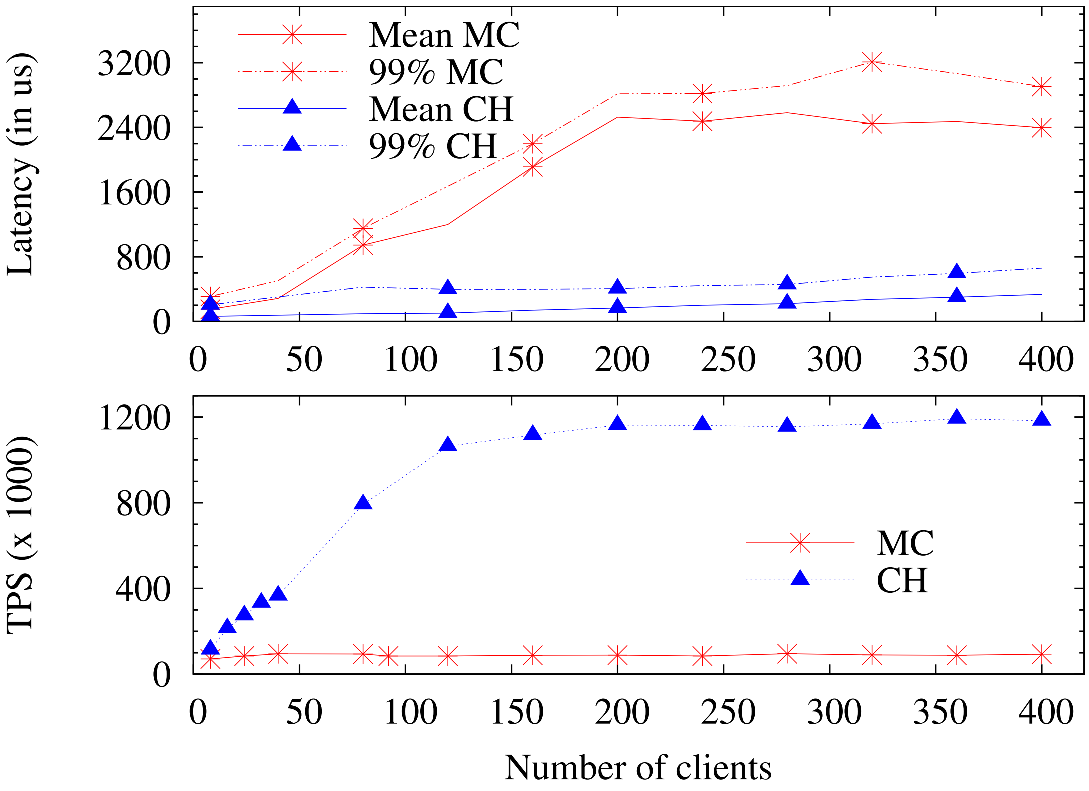


Figure 6.4. Latency as a function of the number of clients with the Memslap benchmark (closed loop).

Chronos-MC supports over 1 million transactions per second (TPS), limited only by the NIC's throughput limit of 10Gbps. With 120 clients, the number of requests served levels out, causing a small amount of additional latency as requests wait to be transmitted at the client. In contrast, baseline Memcached serves fewer request/sec with high latency.

6.2.3 Skew in request inter-arrival times

In this subsection, we show that the techniques used in Chronos deliver predictable low latency even with skewed request inter arrival times. With the skewed workload Chronos achieves 50x improvement relative to baseline Memcached while serving 10,000 requests per second.

The presence of skewed request inter-arrival times means that, although the

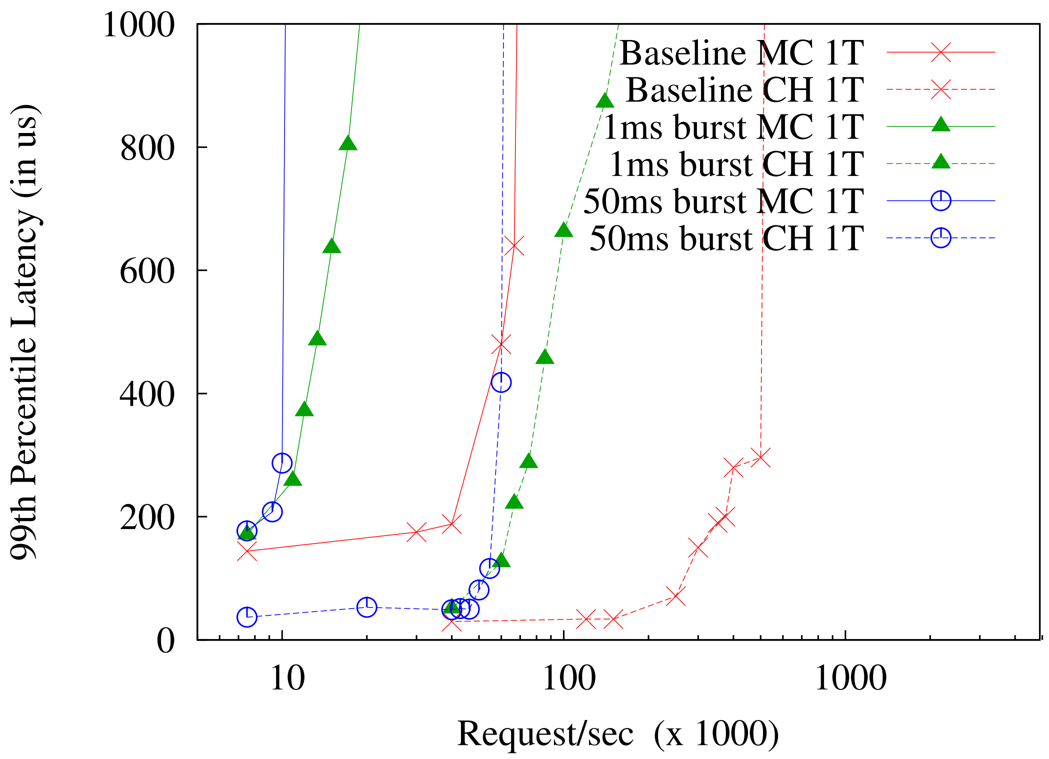


Figure 6.5. The effect of skewed request inter-arrival times on tail latency. X-axis in logscale.

average request load might be manageable, there are short periods of request overload. Depending on how skewed the request pattern is, there might be several back-to-back requests followed by a gap in requests. From the server point of view, skewed workload induces a momentary state of overload, which results in application-layer queueing. To study this behavior, we use the methodology described by Banga and Druschel [15], originally presented in the context of Web server evaluation. Here, multiple clients generate traffic at a fixed rate, punctuated with synchronized short bursty periods. These bursty periods are characterized by two parameters: 1) the ratio of the maximum request rate in the burst and the overall average request rate, and 2) the duration of bursts. We fix the maximum-to-average request ratio to be 10, and limit the burst duration such that each burst has 10% of the total requests sent. Lastly, we ensure that the number of

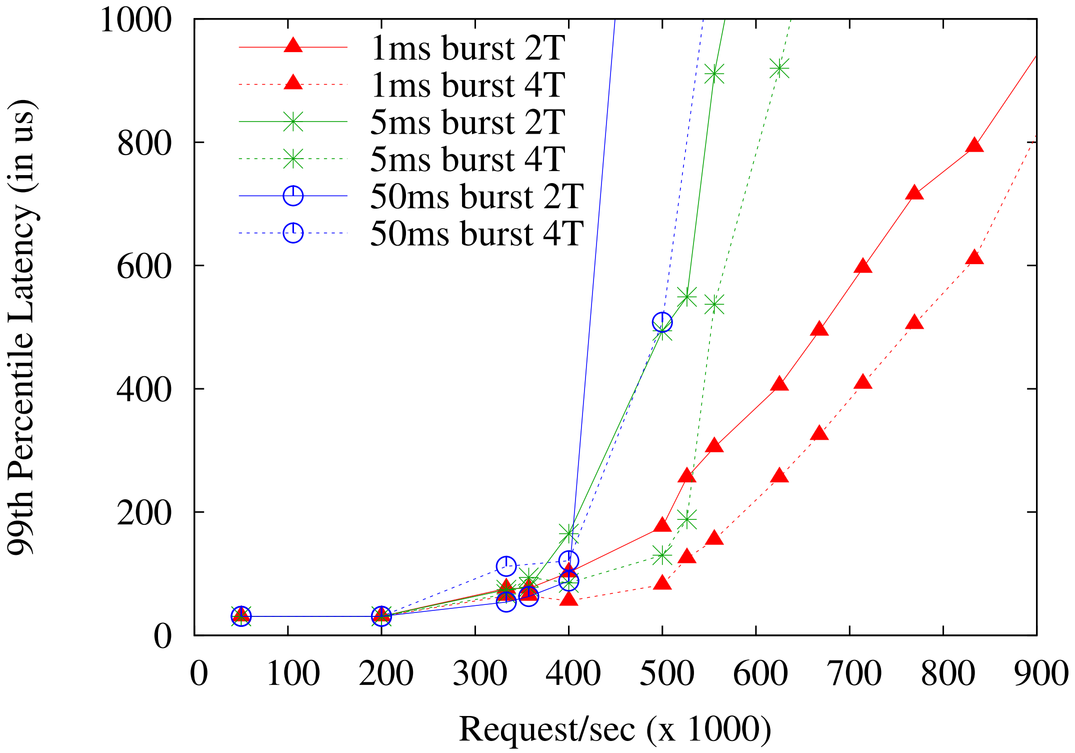


Figure 6.6. The latency with two threaded (2T) and four threaded (4T) instances of Chronos-MC under skewed request arrivals.

requests in a burst are fixed across the experiments.

Figure 6.5 shows the 99^{th} percentile of latency for baseline Memcached (MC) as well as Chronos-MC (CH) across a range of burst periods. We see that in the baseline even short burst durations of 1ms impose significant levels of application queueing at 10,000 requests per second, driving latency up to over a millisecond. Note that without request inter-arrival time skew, baseline Memcached supported up to 120,000 requests per second with sub $500\mu s$ latency. (Figure 6.3). For Chronos-MC under a uniform request inter-arrival rate, latency stays largely flat up through 500,000 requests per second (Figure 6.3). However, just as in the baseline Memcached case, inducing request bursts drives up latency significantly while reducing the throughput of the system. For 1ms bursts, the request rate is reduced to 40,000 requests per second for keeping the latency

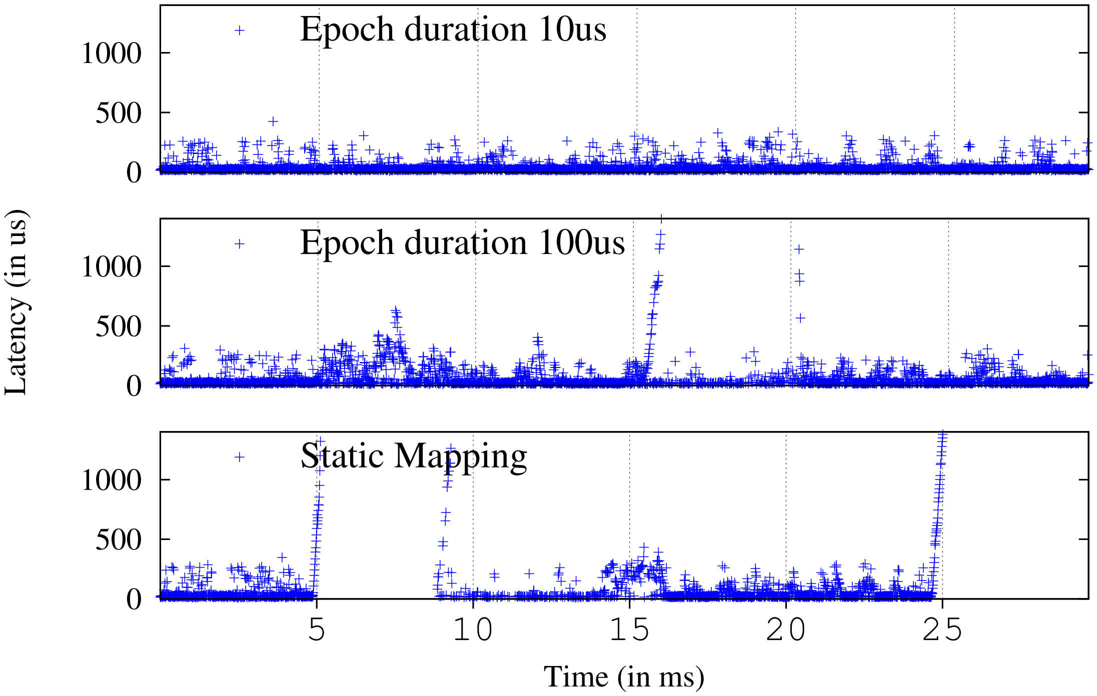


Figure 6.7. An evaluation of the responsiveness of the Chronos load balancer module across two time epochs (10μs on top and 100μs in the middle) and the static mapping strategy (on the bottom).

under 30μs, with an observed latency of up to 1ms at over 150,000 requests per second. For longer burst durations, this effect is more pronounced.

Figure 6.6 shows how load balancing with more threads improves the performance of Chronos-MC. We consider request loads up to 1M requests per second forwarded to Chronos instances with either two or four application threads, each running on its own CPU core. As in the single-thread case, bursts in request rates arriving faster than the effective service time of the application induce application queueing, and thus increases in delay. This effect is more pronounced at higher loads, given that there is less time between arriving requests. Adding additional cores mitigates the effect of bursts, but for sufficient burst lengths queueing will still build up with any fixed number of CPU cores.

6.2.4 Skew in request access pattern

In this section, we show how loadbalancing with Chronos at fine grained time scales significantly reduces the latency variation with skewed request access patterns. Results are shown in Figure 6.7. The Chronos load balancing module periodically reapportions requests across application threads to evenly balance the load. As described in Section 6.1.1, the load balancer works in concert with the NIC-level hash function to ensure that requests are sent to application threads in such a way as to minimize or eliminate lock contention. Thus, with Chronos-MC, it is expected that the load balancer assigns requests across application threads such that each thread sees a strict partition of vBuckets.

We run the following experiment, to evaluate the responsiveness of Chronos-MC to request access skew. We set up a Chronos-MC instance with four threads and configure the load balancing module with an epoch time of $10\mu s$ and $100\mu s$. A single open-loop client sends requests at a rate of 1 million requests/sec. Keys are chosen at random at the start of each client epoch such that three keys receive 99% of the requests. This pattern is motivated by the desire to have three of the four cores handling the hot/popular keys, and have the remaining core receive all of the cold/unpopular keys. We know by construction that without an adaptive load balancing module, each time the client epoch changes overload would occur since two or more popular keys would be handled by a single application thread, and the rate of requests is sufficiently high as to induce overload in that case. Note the client and the server epochs are not synchronized. We repeat the same experiment for a Chronos instance with static mapping of keys to threads. Figure 6.7 shows the latency distribution for Chronos at $10\mu s$ (top), $100\mu s$ (middle), and for the static mapping (bottom). At the start of each epoch, we see occasional long spikes in the 100 μs case before it is able to adapt to shifts in workload. The static mapping approach fails when two or more popular keys are served from the same application since these types of co-located request hotspots cannot be migrated to other cores. Unlike previous figures which show only 99^{th} percentile latency number, Figure 6.7 shows all data-points including few outliers.

Discussion: Due to our reliance on partitioning to spread load across cores, there are certain cases that will cause the load balancing element in Chronos to perform poorly. When a single key in a partition, or the partition itself, becomes hugely popular, the rate of requests to that partition can overload a single thread. This happens when the request load approaches 500,000 requests/sec (which is greater than 5 Gbps of traffic). When a single key becomes that popular, we are limited in our response, and would suggest that the application itself be re-architected, since such a high get/set load on a single key would not be practical at scale. However, it is more likely that several keys in the same partition might together induce such a high load. We can alleviate this condition by moving those common keys to separate vBuckets, or by modifying the request handling logic in Chronos to allow the server to split and join buckets based on load demands. We have not yet evaluated these possible features.

6.2.5 Chronos Web Search

As described in Section 6.1.2, the Web search application maintains a hash table to store the term and associated document, protected by read/write locks. In Chronos-WS, we further divide this index into twelve partitions based on the term, and store them in separate tables protected by a mutex. We evaluate Chronos in comparison to an RCU lock-based implementation of the hash table that was provided by Triplett et al [81]. Additionally, we modified this implementation to work with the same user-level networking API used in Chronos to provide a direct comparison. For search we used 10-byte keys and 1400-byte values in the inverted index list, with a get/set requests ratio

Table 6.1. Latency of the OpenFlow Controller.

Component	# Switches	Mean latency (μs)	99 %ile latency (μs)
OpenFlow	1	65	140
OpenFlow	16	120	250
Chronos-OF	1	8	50
Chronos-OF	16	10	51

equal to 9:1. Figure 5.3 shows the results of our evaluation. Even with an implementation based on read/write locks and RCU, we see higher latency compared to Chronos-WS with large number of clients. The performance improvement of Chronos-WS would be higher if the workload shifted towards a more write-heavy mixture. The reason for this is that these primitives are optimized for read-heavy workloads and Chronos makes no such assumption about workload type. The RCU implementation based on user-level APIs scales up to 550K requests/sec, while the Chronos implementation scales up to 1M requests/sec. At low request rates and low levels of concurrency, the RCU implementation has similar performance as Chronos-WS. But as we increase the number of clients, and thus load on the server, the application latency increases from 2-3 microseconds to 6-11 microseconds for RCU-WS. This small variation in application latency results in a large end-to-end latency at high loads due to increased queuing delay.

6.2.6 Chronos OpenFlow controller

Finally, we show that the Chronos based implementation of the OpenFlow controller (Chronos-OF), which uses TCP for handling requests, reduces the mean latency for request processing by a factor of 12x as compared to baseline.

For this experiment, we replaced the default kernel TCP network implementation in the controller with the user-level TCP implementation provided by our NIC vendor in our evaluation testbed. The controller software itself is single-threaded. For generating load, we used the *Cbench* benchmark [27]. Cbench emulates switches that send *packet-in* messages to the controller, and waits for flow modification rules to be inserted in the switch forwarding tables in response. The controller implements a learning switch application, which generates appropriate forwarding rules in response to packet-in events. We simulated 16 switches supporting 1M MAC entries as suggested in [27]. To measure the controller latency, we installed a packet mirroring rule described in Chapter 5.

Table 6.1 shows the results of this experiment. We see that removing the kernel has the predictable effect of reducing average latency. However, the effect on the 99^{th} percentile of latency is that the difference between one emulated switch and sixteen emulated switches is only a single microsecond, as compared to 110 microseconds in the baseline case. We expect Chronos-OF controller performance to improve further by enabling load balancing for a multi-threaded implementation.

6.3 Discussion

To achieve high efficiency, data center networks often rely on multi-tenancy and server virtualization to maximize resource usage. The feasibility of Chronos depends on being able to support these techniques in in a variety of different data center environments.

In a large, multi-tenancy data center, latency sensitive applications share the same end-host with other jobs. A key question for Chronos is what impact this sharing has on latency, and in particular tail latency. To gain some insight into this question, we setup an experiment to test this condition. We first set up a Memcached server, and started a background job that receives traffic from six clients in parallel. Each client sends traffic at rate of 440Mb/s to this background job. We instantiated 21 Memslap clients, and measured the latency of both a stock Memcached server, and Chronos, with and without the presence of the background traffic. These particular rates and numbers of clients were chosen to induce sufficient load on the system to evaluate this question. In the case of baseline Memcached, the presence of background traffic resulted in more than a 60% increase in tail latency, while Chronos-MC's performance was not affected by the presence of the background traffic. This initial result indicates that Chronos can provide low latency in the presence of multi-tenancy, and we seek to further evaluate this in more depth in future work.

Supporting virtualization in the data center and consolidating multiple VMs on a single end-host have become common place today. NIC hardware has been augmented to support SR-IOV, or Single Root I/O Virtualization. SR-IOV allows a guest OS to directly configure access to virtualized instances of the NIC without going through the hypervisor. Although not implemented in this work, we expect Chronos to leverage these features to provide predictable latency in a virtualized setting.

6.3.1 Effect of NUMA-awareness on latency

Modern processor architectures employ non-uniform memory access (NUMA) architectures, in which memory is partitioned across two or more banks, or domains. The access time to a core-local domain is lower than that of a remote domain, and so it is advantageous to organize memory to be as domain-local as possible. To evaluate the effect of NUMA on Chronos, we setup an experiment as follows. We choose a Chronos-based Memcached instance with four threads, of which two are in one NUMA domain, and two are in the other. We then adjust the memory allocator to allocate domain-local memory for each thread. We compared the observed latency of this with a second Chronos-based Memcached instance in which the allocator selects entirely domain-remote memory for each thread.

Figure 6.8 show the latency in these two cases. At low to medium rates of requests, there is little difference between the two policies. As the request rate exceed 1

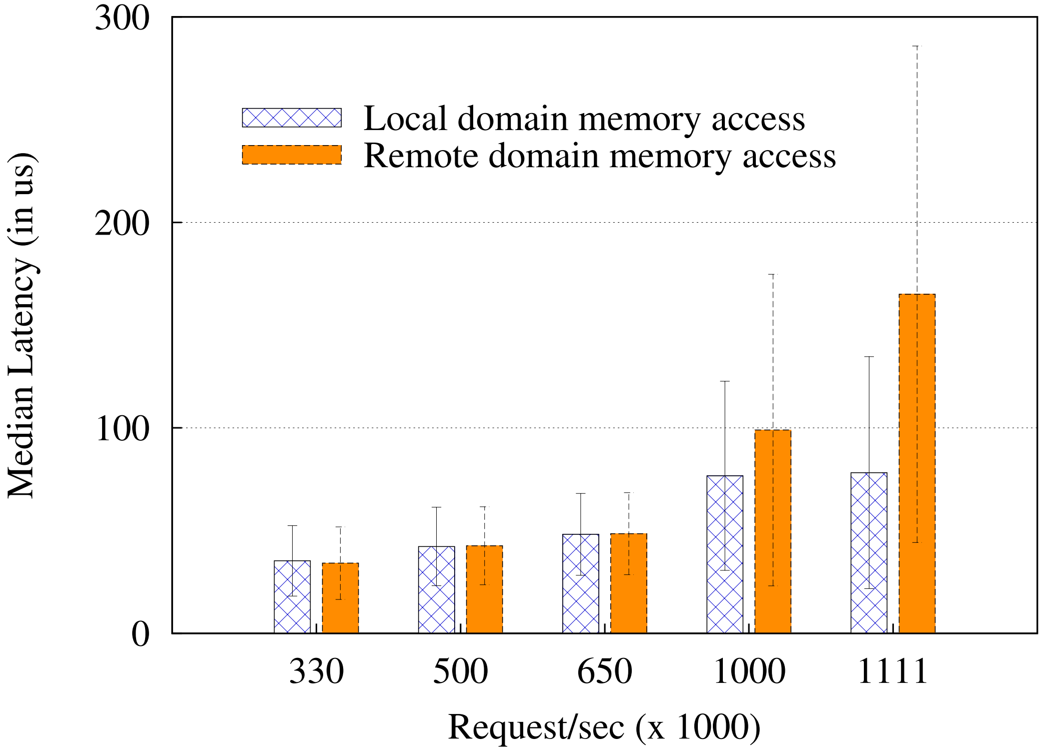


Figure 6.8. The effect of NUMA-awareness on the Chronos-Memcached load balancer. There is little difference at lower levels of utilization, and an approximate doubling of latency (and latency variation) at the highest levels of utilization.

million requests per second, there is a divergence in which the NUMA-remote instance imposes almost double the latency of the NUMA-local instance, with significantly high latency variation.

In our testbed, each NUMA domain contained four cores, which alone were enough to saturate the 10 Gbps NIC. Thus, it is not necessary to load balance requests across NUMA domains to meet throughput requirements. So ensuring that the load balancer restricts requests to NUMA-local cores is adequate for current link speeds. Furthermore, when running the server in low or moderate request loads, the effect is minimal in either case. Thus NUMA effects are not significant to the efficiency of Chronos, however their effect might become more pronounced in environments utilizing virtual machines. The specific issue arises when cores from different NUMA domains are assigned to the same virtual machine, causing high memory latencies and increasing tail latency.

6.4 Summary

The scale of modern data centers enables developers to deploy applications across thousands of servers. However, that same scale imposes high monetary, energy, and management costs, placing increased importance on efficiency. To meet strict SLA demands, developers typically run services at low utilization to rein in latency outliers, which decreases efficiency. In this work, we present Chronos, an architecture to reduce data center application latency especially at the tail. Chronos removes significant sources of application latency by removing the kernel and network stack from the critical path of communication by partitioning requests based on application-level packet header fields in the NIC itself, and by load balancing requests across application instances via an in-NIC load balancing module. Through an evaluation of Memcached, OpenFlow, and a Web search application implemented on Chronos, we show that we can reduce latency by up to a factor of twenty, while significantly reining in latency outliers.

6.5 Acknowledgments

We would like to thank Abhijeet Bhorkar and Mohammad Naghshvar for input on our analytical analysis. We woild also like to thank authors of [81] for sharing RCU implementation. This work was supported in part by NSF Grants CSR-1116079 and MRI CNS-0923523, and a NetApp Faculty Fellowship.

This chapter in part, contains material as it appears in Kapoor, Rishi; Porter, George; Tewari, Malveeka; Voelker, Geoffrey M. ; Vahdat, Amin. “Chronos: Predictable Low Latency for Data Center Applications”, Proceedings of the ACM Symposium on

Cloud Computing (SOCC), San Jose, CA, October 2012 The dissertation author was the primary investigator and author on this paper.

Chapter 7

End-host support for Reconfigurable Topologies

7.1 Introduction

In chapter 6 we looked at an end-host architecture to reduce data center application latency especially at the tail. Though latency is an important metric but the existing large-scale data center installations are limited by the ability to provide sufficient internal network connectivity. Delivering scalable packet-switched interconnects that can support the continually increasing data rates required between literally hundreds of thousands of servers is an extremely challenging problem that is only getting harder. Fundamentally, the packet-switching technology underlying current data-center interconnects limits their ability to scale: implementing control logic that is capable of deciding how to forward each packet individually is costly at present, and will rapidly cease to be feasible as link data rates increase.

Researchers have attempted to address this issue by adopting the superior power and cost scaling enabled by optical circuit switching [33]. Traditionally, circuit switching has been at odds with the packet-switch discipline that many applications depend upon (to provide, for example, low latency connectivity to a large number of destinations). Researchers have tried to address this discrepancy by proposing hybrid architectures that combine packet- and circuit-switched interconnects [33, 85]. At their core, these approaches search out large, stable flows and route them over circuits, while forwarding the bulk of the traffic through the packet network. Initial designs are limited by the slow switching time (10s of milliseconds) of commercially available MEMS-based optical circuit switching technology, which makes it necessary to combine traffic from multiple end hosts to get traffic aggregates that remain stable at the timescales required (seconds) to achieve reasonable levels of circuit efficiency.

These “hybrid” networks propose to schedule appropriately large traffic demands via a high-rate circuit switch and handle any remaining traffic with a low-rate packet switch. All recent proposals for such hybrid designs assume a perfect closed-loop control plane, including an omniscient scheduling oracle that can compute a switch configuration instantly and map traffic to these circuits in an optimal fashion. In practice, the performance of any hybrid network is critically dependent on all aspects of the closed-loop control plane including the speed of the demand estimation, how that demand is used to calculate the schedule in near real-time, and the ability to synchronize endpoints across circuits.

In this chapter, we explore ways to build a closed-control plane for “hybrid” ToR that is itself both electrical and optical. We describe the requirements on the closed-loop control plane and on the transport protocol that are necessary to make it practical to implement in both existing and recently proposed hybrid switches. We propose and experimentally evaluate a practical first-generation closed-loop control plane for a hybrid network that features a data center network. We show how the network traffic demand can be estimated in the host stack and communicated to a controller. Next, we show that the transport protocol TCP is insusceptible if the underlying reconfigurable technology is rapidly switching. We demonstrate that TCP protocol performs poorly if reconfigurable topology has paths of different capacity. Finally, we show that MPTCP is better suited

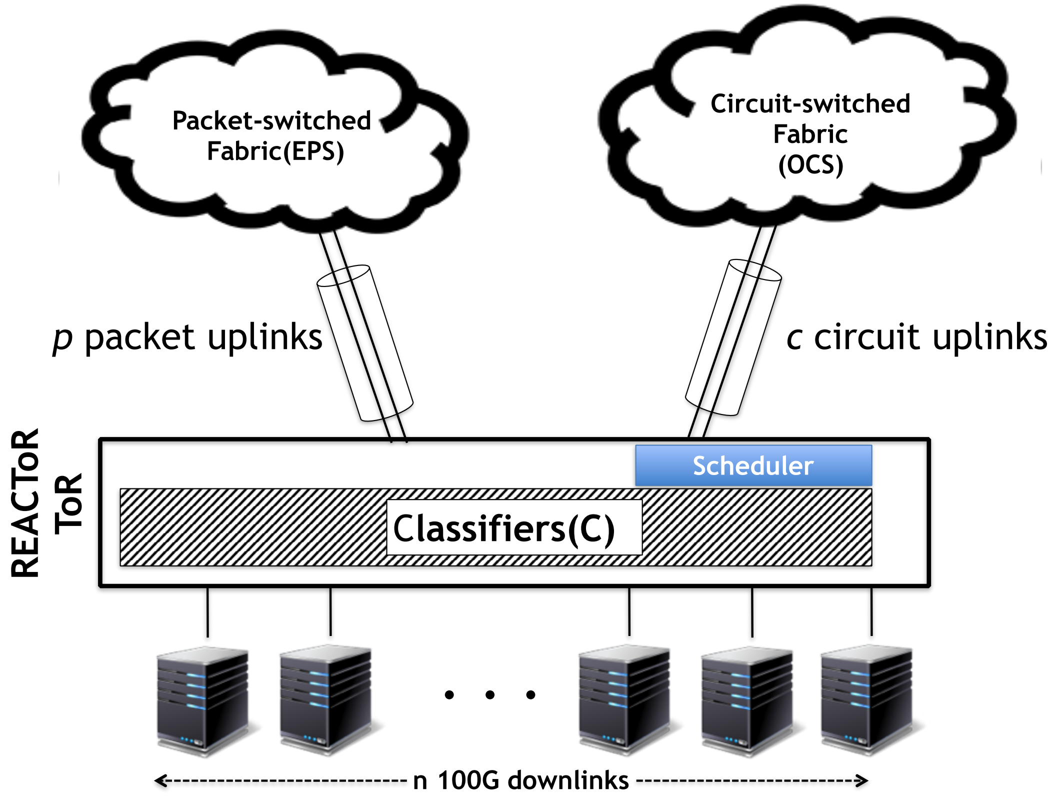


Figure 7.1. 100-Gb/s hosts connect to REACToRs, which are in turn dual-homed to a 10-Gb/s packet-switched network and a 100-Gb/s circuit-switched optical network.

for reconfigurable topology and evaluate the closed-loop control plane.

7.2 REACToR overview

We now give an overview of a REACToR-enabled data center network. A REACToR-enabled data center consists of N servers grouped into R racks, each consisting of n nodes. REACToR assumes a preexisting 10-Gb/s packet-switched network (EPS) and overlays it with an additional 100-Gb/s circuit-switched network (OCS). At each rack is a hybrid ToR called a REACToR, which is connected to the packet-switched network with p uplinks and is connected to the circuit-switched network with a separate set of c uplinks.

Referring to Figure 7.1, an (n, p, c)-port REACToR consists of n downward-facing ports connected to servers at 100 Gb/s, $p = n$ uplinks connected to the packet-switched network at 10 Gb/s, and $c = n$ uplinks connected to the 100-Gb/s circuit-switched network. At each of the n server-facing input ports, there is a classifier (marked 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 circuit uplinks, instead packets are buffered in the end-host. 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.

End-hosts: 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. 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. Within each host, there is 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. End-host rate limit the EPS traffic class to to link speed of EPS to ensure that EPS is not overrun. Similarly, OCS traffic class is rate limited at the end-host to ensure that EPS and OCS traffic can be merged at downward facing port of REACToR .

Host control plane: 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. REACToR also uses this control plane to manage end-host traffic classes and buffering. Each traffic class in the end host corresponds to a PFC class. 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.

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[1], and corresponding durations, ϕ_k. REACToR uses Solstice , a scheduler designed specifically for hybrid switching. Solstice is a stateless scheduler i.e. when Solstice computes a schedule, it only considers demand collected during a given accumulation period W

Traffic Demand Estimation: Effective utilization of circuits require accurate

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

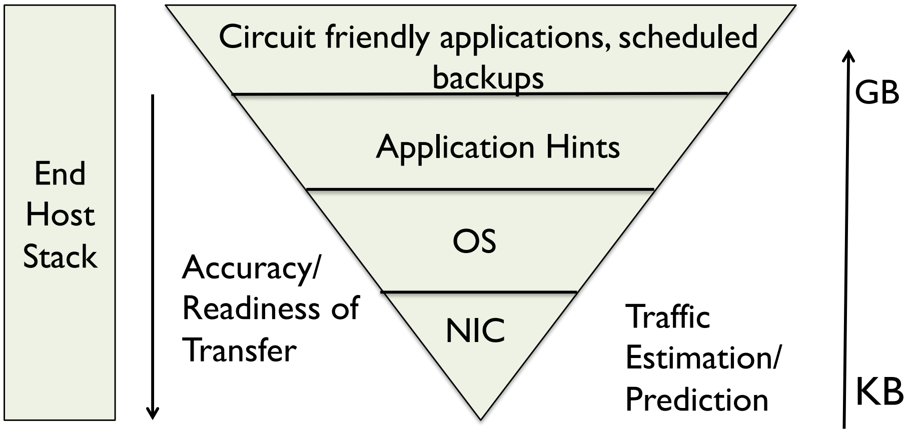


Figure 7.2. Network traffic estimates across the end-host layers

and timely estimation of network traffic demand. To this need, an end-host can be instrumented to collect network traffic statistics. Figure 7.2 shows different layers across the end-host stack from which network traffic demand can be collected. From the figure two trends are apparent. The size of network traffic that can be estimated from a layer increases as you go up the pyramid. At the bottom of the pyramid is the NIC buffers with size varying from 1-2 MB [41]. The next layer is the operating system buffers which can hold up to several Megabytes of network traffic. The socket system API can indicate transfer size up to few Gigabytes. A network demand estimator can report the size of these buffers indicating amount of bytes queued up at the end-host. Centralized job schedulers e.g.,Yarn Resource Manager [38] has a list of map tasks which are completed and are waiting for being fetched by the reduce tasks. These transfers can be reported hundreds of millisec ahead of the actual transfer.

However, the readiness of transfer follows an inverse trend i.e. timeliness of transfer decreases as we move down the pyramid. The reason being, packets queued up in the NIC buffers are ready to leave the host, whereas the flows reported by a centralized job scheduler e.g., Yarn Resource Manager could take few 100s of millisec to start.

Implementation: We instrumented the end host stack to collect network traffic demand from the OS socket buffers and the NIC queues. We wrote a kernel module that keeps track of send and sendto calls made by applications. This connection information is stored in a hash table. A separate control thread loops over the hash table to obtain socket buffer occupancy. The control thread aggregates demand across these connections and builds a *demand packet*. This *demand* packet is sent to a centralized controller which further aggregates the individual estimates from end hosts to build a network wide traffic matrix.

7.3 TCP and control plane

The BulletTrains study (chapter 1) shows that any application flow exhibits intrinsic short-term correlated bursts as a consequence of batching in the end-host stack. This is important for REACToR because if a circuit is allocated an interval at a time, any moment when the instantaneous demand does not fully saturate (to at least 90%, in our example) the circuit's link rate implies that some of the circuit bandwidth is wasted.

In this section we consider the coherence properties under a circuit scheduler, and particularly we focus on interaction of the control-plane with end-host transport protocol TCP. TCP is used by applications to communicate within data centers; hence its perfomance need to be evaluated. We study the impact of pausing and unpausing on TCP followed by studying the impact of circuit scheduling and multipath (EPS and OCS) on TCP.

7.3.1 TCP under TDMA scheduling

Here, we consider a scheduler that pauses flows while they wait for a circuit to be assigned to them, and then unpauses them when that circuit is established. During the time that a flow is paused, additional packets from that flow may be generated by the corresponding application, which should increase the length of the burst when the flow is unpaused. However, the increased latency and latency variation induced by pausing and unpausing the flows may have a detrimental impact on the transport protocol (e.g., TCP) or the application itself. As a first step we study the impact of pause and unpause behavior on a bi-directional circuit, i.e., pausing both data and TCP ACKs at the same time.

To study this impact, we recreated several workloads taken from the Helios [33] work on a set of six nodes connected to the same switch (a Fulcrum Monaco 10-Gb/s Ethernet switch), using Intel 82599-based NICs with TCP segmentation offload enabled. To collect fine-grained time stamps for traffic emanating from a source host in this test bed, we installed optical transceivers in both the source host and the Monaco switch, connected with a fiber. Along this fiber we interposed an optical splitter, which sends the signal to the switch, and an identical copy of the signal to a monitoring host with a Myricom Sniffer10G packet capture tool.

We also connected a scheduler host containing a 1QB/s-Netflix card to the switch using an RJ45-to-SFP+ interface. The NetFPGA is responsible for sending out 802.1Qbb pause frames at precise timings to a multicast address, which causes it to arrive to each of the hosts at close to the same time. By emitting alternating pause and unpause frames, we can emulate a circuit schedule with varying-length day and night time intervals.

Figure 7.3(a) shows the resulting burst distribution when varying the day length for the all-to-all workload. For short and medium-length days (up to 500 μs), the resulting bursts are quite close to the ideal length. For example, for a 500 μs day, the burst length is 412 1500-byte packets, which is 494 μs. At 750 μs, about 10% of the bursts are shorter than the day length, distributed uniformly between about 180 μs and 750 μs. The stride workload (not shown) is quite similar to the all-to-all workload, except that nearly all of the 750-μs day lengths are fully utilized.

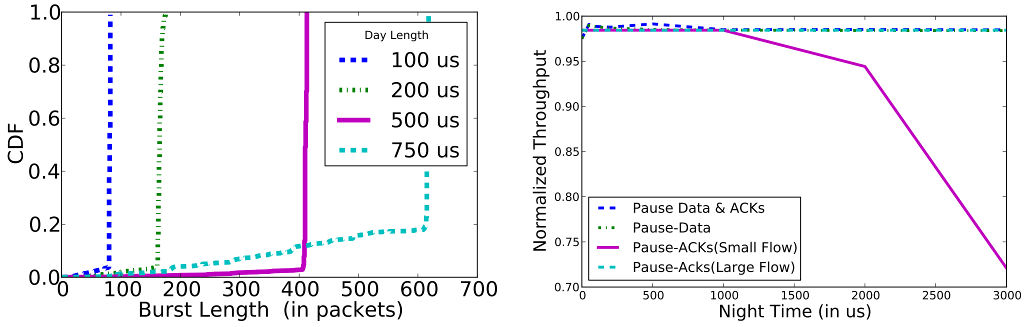
Next, we consider cases where we pause the data in the flow, but allow ACKs to return unimpeded (e.g., via the EPS), as well as cases where we enable the data in the flow, but pause the ACKs. Figure 7.3(b) shows the resulting normalized throughput when varying the night time under the stride workload. In the first case, we see that the normalized throughput of uni-directional and bi-directional circuits is close to ideal, showing that pausing the data portion of flows on the end hosts does not affect throughput for pause lengths considered by REACToR. In the case of pausing ACKs, we find that there are two regimes to consider. During slow start, pausing ACKs decreases the overall throughput of the flow—up to 30% for 3-ms night times. For shorter periods (e.g., $\leq$1 ms) there is no detectable effect for pausing ACKs). Once the flow leaves slow start, there is no effect on throughput regardless of the night time.

These experiments consider the effect of circuit scheduling on TCP traffic in the absence of packet loss. In reality, packets may be lost for a variety of reasons, so it is important to understand the impact that scheduling has on TCP's loss recovery mechanisms. We repeated the experiments where each end host drops packets uniformly at random with a configurable probability. While TCP throughput suffers as expected with increasing drop rates, we find that the difference in performance with and without scheduling (e.g., with and without issuing PFC pause frames) is insignificant for steady state loss rates up to 1%; higher loss rates pose significant challenges for TCP in our setting in either case.

7.3.2 TCP and stateless scheduling

We now consider the interaction of TCP with the other components of a control loop using using the REACToR hybrid switch testbed [52]. The testbed consists of a 10-Gb/s hybrid ToR switch connected to eight end hosts.

When Solstice computes a schedule, it only considers demand collected during a given accumulation period W, making it a stateless scheduler. As a result, it is possible



(a) Scheduled all-to-all burst sizes

(b) Scheduled stride burst sizes

Figure 7.3. Scheduled Burst sizes for various workloads.

that over several consecutive invocations, it might alternate the assignment of a particular flow between the packet-switched path and one or more circuit-switched paths. Given the short duration of scheduling periods (e.g., a few milliseconds), this oscillation has the potential to disrupt the control plane of transport protocols such as TCP. In this section, we discuss this problem and propose an endhost-based mechanism to mitigate its effects.

7.3.3 Multipath packet reorder

When a new flow begins, its sending rate is small (due to slow-start) and so unless a circuit was already established to its destination, this new flow will be initially assigned to the packet-switched path. As its rate increases, its demand will grow and eventually trigger a circuit assignment from Solstice. During the next scheduling period, packets are sent over a dedicated circuit that does not have any intermediate queues. As a consequence, packets from both paths can arrive at the receiver in an interleaved fashion, triggering duplicate acknowledgments that cut the sender's congestion window, lowering overall throughput. Figure 7.8 depicts this scenario during a simple 128 MB file transfer using TCP CUBIC on Linux 3.14.22.

To verify out of order receive, we assigned VLAN numbers to queues in the end-host NIC. This VLAN number was then used to distinguish the path a packet took

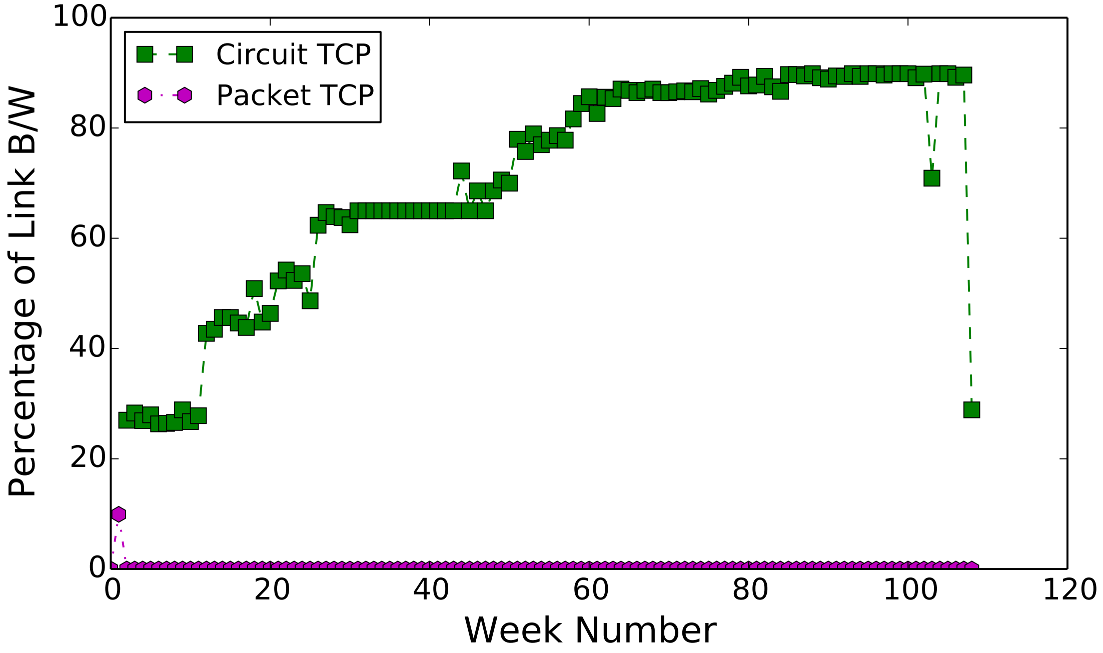


Figure 7.4. Solstice promotes a flow from the packet switch to a circuit. Packets from both paths initially arrive interleaved, and TCP triggers duplicate ACKs and fast re-transmits, lowering throughput.

i.e EPS or circuit switch. Figure 7.5 shows a zoom in version of TCP sequence number vs time. In the experiment, we have a single TCP flow between a client and a server. Initially flow is assigned to the packet switch path and is promoted to circuit switch path after few ms. In the graph, we can see that packets are initially send out on the packet switch queue and then when circuit is assigned there is a TCP sequence number jump. Also, we can see that same sequence number packets are sent over the packet switch queue and circuit queue indicating "re-transmissions".

To verify *re-transmissions*, we used TCP probe module on the end-host. The TCP probe module exports TCP connection state information that can be accessed and analyzed offline. Figure 7.6 shows congestion window (cwnd), ssthreshold (ssthresh) and total re-transmissions (re-transmit) for the connection. In this case the re-transmissions occur because packets from the circuit path appear before the packet switch path packets

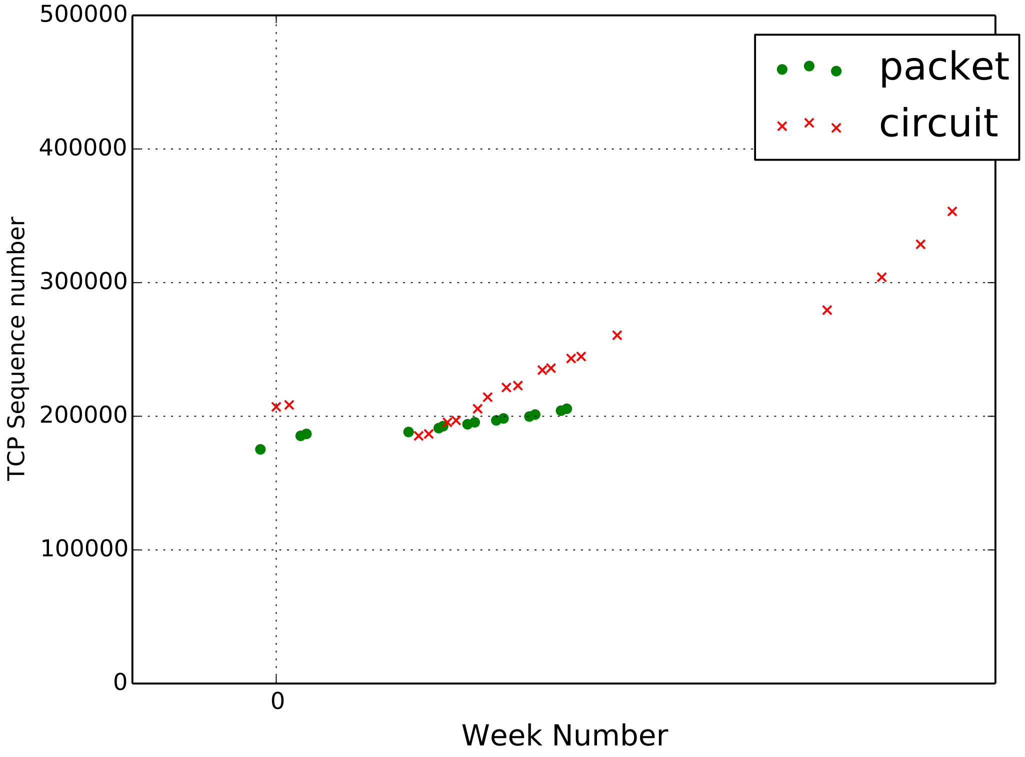


Figure 7.5. Packets from both EPS path and Circuit path arrive interleaved.

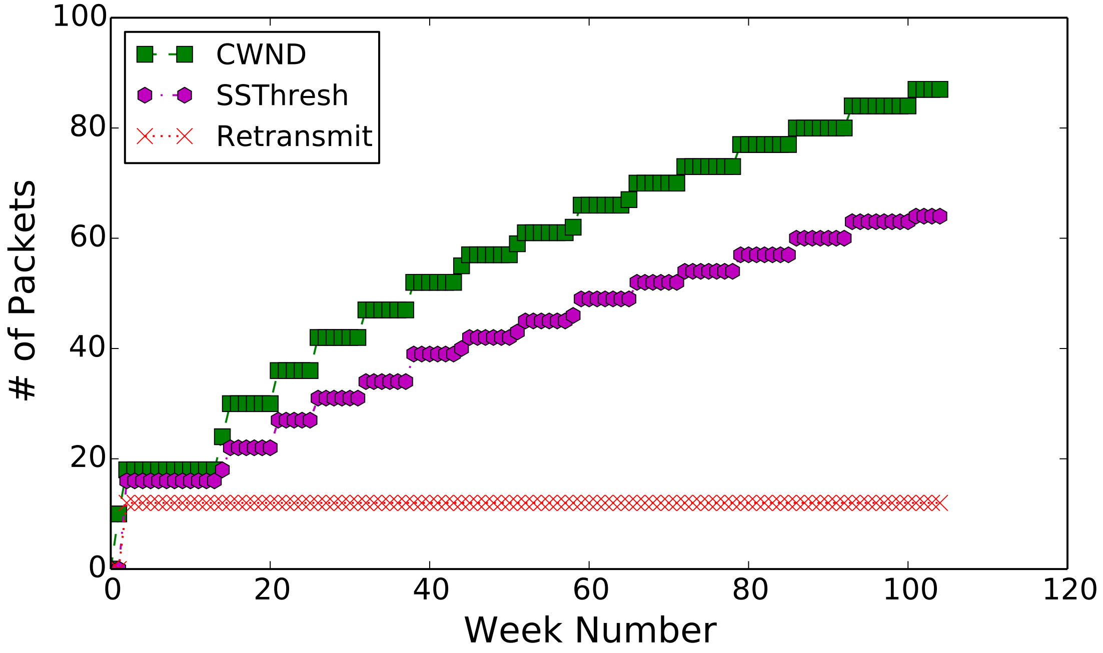


Figure 7.6. Solstice promotes a flow from the packet switch to a circuit. Packets from both paths initially arrive interleaved, and TCP triggers duplicate ACKs and fast re-transmits, lowering throughput. 'Retransmit' shows the numbers of packets re-transmitted. 'CWIND' indicate TCP congestion window size. 'SSThresh' indicates TCP SSTHRESHOLD value.

resulting in out of order delivery that triggers fast re-transmission from the sender.

Mitigation technique: To avoid performance loss due to reordering, TCP reordering buffer can be introduced. Another solution is to disable TCP fast recovery and fast re-transmit. Yet another solution is to increase the tcp_reordering parameter in Linux. This parameter indicates number of packets that can be received out of order without assuming packet has been lost on the way. Downside of these schemes is that recovering from packet loss is delayed further.

7.3.4 Packet switch incast

Depending on the specific configuration of the circuit schedule, TCP can also suffer performance issues even after this startup phase. Specifically, if the delay-bandwidth product of a circuit is higher than the packet-switched path, a flow at steady-state might achieve a much larger congestion window than can be supported on the packet switch. This situation exists for the prototype control plane. As a result, when Solstice migrates that flow from a dedicated circuit back to the packet-switched path, the sender will transmit a burst of packets into the packet-switched path, potentially causing congestion and interfering with other flows. If the flow continues to remain on the packet-switched path, TCP will eventually converge to a fair sending rate, however it will be too late to mitigate the damage done by this initial "incast" behavior.

Mitigation technique: Currently, Solstice leaves a residual amount of the traffic matrix for the packet switch to "take care of." Instead, another method is to compute a deliberate schedule for the packet switch as well, for example via FastPass [61]. Solstice can determine the appropriate rate for each sender based on the residual traffic matrix assigned to the packet switch, and can share those rates with endhosts for each scheduling period. Constructing a deliberate schedule for the packet switch in concert with the circuit switch is future work.

7.4 MPTCP and stateless routing

A straight-forward way to mitigate TCP reordering problem is to support separate congestion windows and sequence spaces for the packet-switched path and the dedicated circuit path. To do that we added support for MPTCP [66] to the hardware testbed. Figure 7.7 shows the end-host stack with MPTCP. At a flow creation time, we instantiate two MPTCP subflows: one assigned to the packet-switched path, and one to the circuit-switched path. Unlike a true multipath environment, we only want one subflow to be active for any given time interval. This means that either the packet subflow is active or the circuit subflow is active. To enforce this, we assign each subflow to its own hardware NIC queue, associated with a unique 802.1Qbb Priority Flow Control (PFC) [62] class

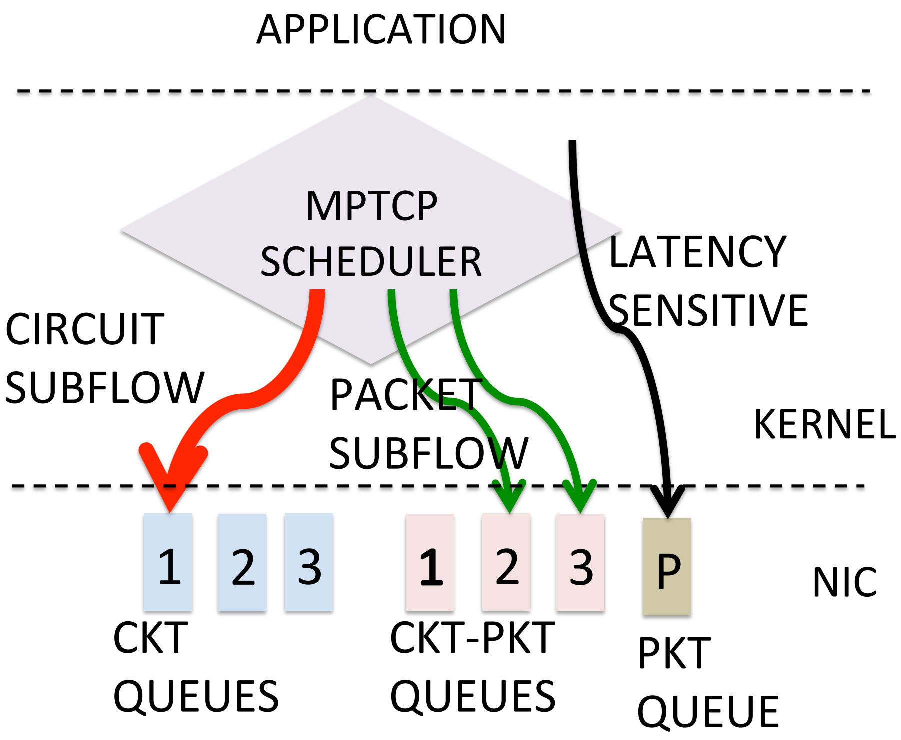


Figure 7.7. End-host stack with MPTCP.

of service. We then configure a controller colocated with the Solstice scheduler to issue 802.1Qbb pause and unpause messages to these NIC queues.

Figure 7.8 also shows the same file transfer as before, except relying on MPTCP. The file transfer initial takes the packet-switched path, until Solstice promotes it to a circuit. Although packets arrive interleaved as before, MPTCP's maintenance of separate state machines per path prevents the sender's congestion window from shrinking prematurely, reducing the transfer time.

7.5 Closed loop evaluation

We now consider applications performance on the closed-loop REACToR hybrid switch testbed. This testbed consists of a 10-Gb/s hybrid ToR switch connected to eight

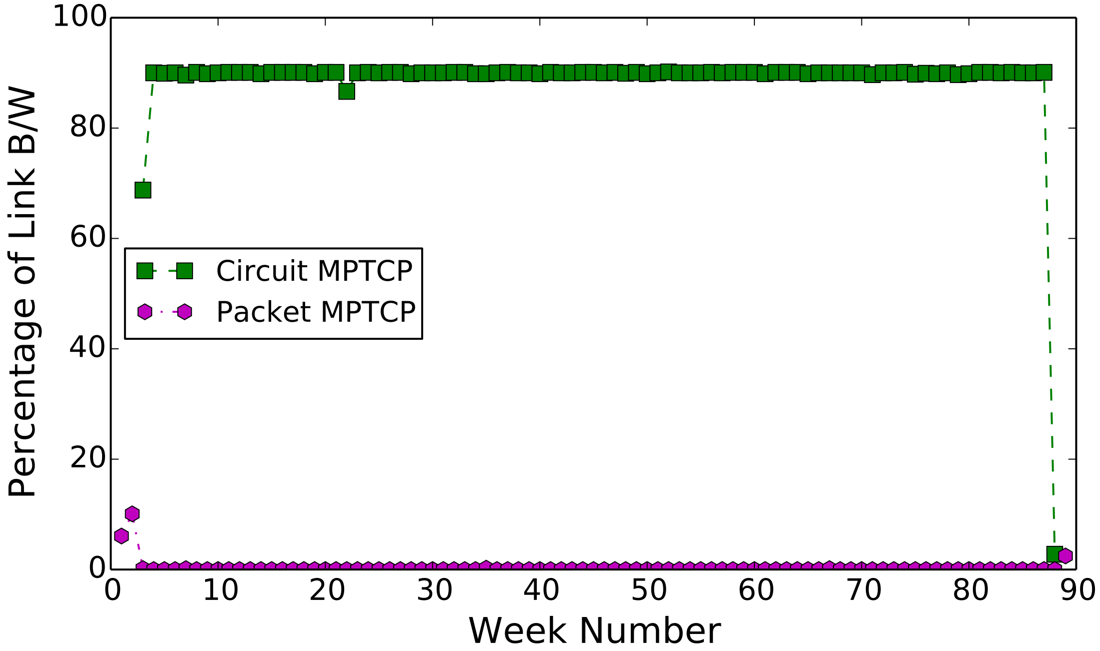


Figure 7.8. MPTCP maintains good performance using separate per-path TCP state machines.

end hosts. For this experiment, we have instrumented REACToR with a closed-loop control plane using Solstice for the scheduling. This control plane collects demand using a custom-written kernel module described in section 7.2. The kernel module sends a demand estimate packet at configurable time intervals (e.g., 1.5 millisecond). The demand estimate packet is currently sent via a separate control network. We use MPTCP as the transport protocol unless otherwise stated. We instrumented NETFPGA to provide bytes and packets transferred for every circuit switch configuration.

To begin our evaluation, we first measure the reaction time of our closed-loop control plane. Next, we measure performance of a set of synthetic microbenchmarks, which are generated via simple memory-to-memory data transfers to eliminate any effects from the disks, application logic, and think time. Finally, we analyze the behavior of two real world applications, Hadoop terasort (bandwidth intensive) and Memcached (latency sensitive).

Performance: The overall speed of the closed-loop control plane is bounded by the speed of demand estimation at end-host, latency of communicating the estimate to a controller, computation of the schedule and the switch reconfiguration time. To quantify the overall control plane latency we microbenchmark individual components and also the complete system. In our 8 node setup, the demand estimate loop takes 400 nanoseconds, packet generation and delivery takes 2-5 μs, schedule computation for 8 node testbed take 50-100 μs and the switch reconfiguration time is 30 μs.

Next, we evaluate the reaction time of our closed-loop control plane that is how long does it take when a new flow appears at the end-host and when the circuit is assigned to the flow. To measure this latency, we start a UDP flow between two hosts. For the purpose of this experiment we assume that flow has sufficient demand to trigger circuit assignment. We take timestamps at the application layer, the time the demand estimate packet is received at controller and when the first packet is sent over the OCS. We find that the worst-case reaction time is 1.9 ms and the primary component of that latency is the accumulation period which is set to 1.5 ms. In addition, the scheduler starts computing future schedule 200 - 400 μs period in advance.

Microbenchmarks: The microbenchmark traffic patterns we consider is one to all traffic pattern. In a one to all, one host sends data to remaining hosts. With microbenchmark traffic we show that our control loop is functional across several hosts, demand estimate is accurate and the Solstice computed schedule is along the expected lines. We use a variant of one-to-all traffic pattern where flows are added and removed over a period of time. This traffic tests whether our control loop can react to changes in the demand. Figure 7.9 shows the bandwidth utilization of flows for one such traffic pattern. At the start of the experiment there is a single flow and it gets an exclusive full circuit. As more flows are added into the system the bandwidth is equally shared between the flows, week 3K to 5K. Between week 7.5K to 8K the demand varies across the flows and Solstice assign circuits to the flows proportional to the demand, hence the bandwidth is different between two flows. The key takeaway from the graph is that our control loop can quickly react to changes in the system.

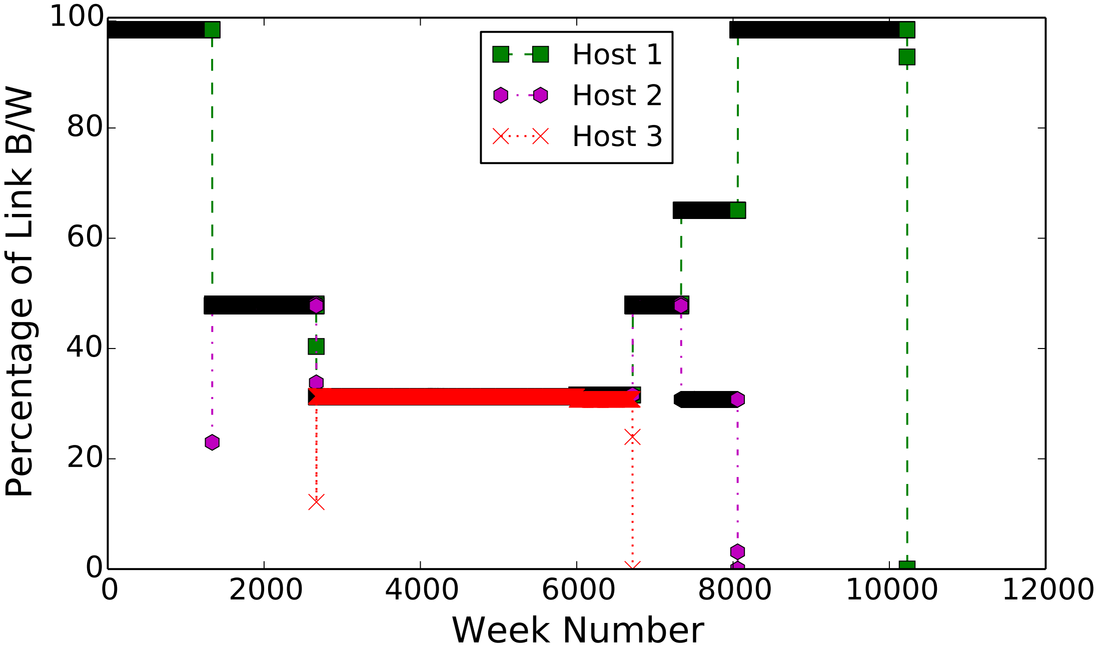


Figure 7.9. Circuit utilization for a variant of one-to-all traffic pattern.

Next, we analyze the behavior of a bandwidth-sensitive application (i.e., Hadoop terasort) [87] and a latency-sensitive application (i.e., Memcached [53]).

Hadoop: We configured the NameNode and Yarn resource manager to run on a single server. Three other hosts on the testbed were configured as DataNode with data stored in memory to speedup the network transfer. Hadoop control traffic between nodes was always classified to the packet switch queue. The number of map and reduce tasks are chosen such each host is assigned one map and one reduce task. We ran terasort application, a bandwidth intensive application, on closed loop REACToR and compare its performance to equivalent 10Gbps EPS. We logged the time it took to complete the job. Figure 7.10 shows circuit utilization for different flows during the shuffle phase of

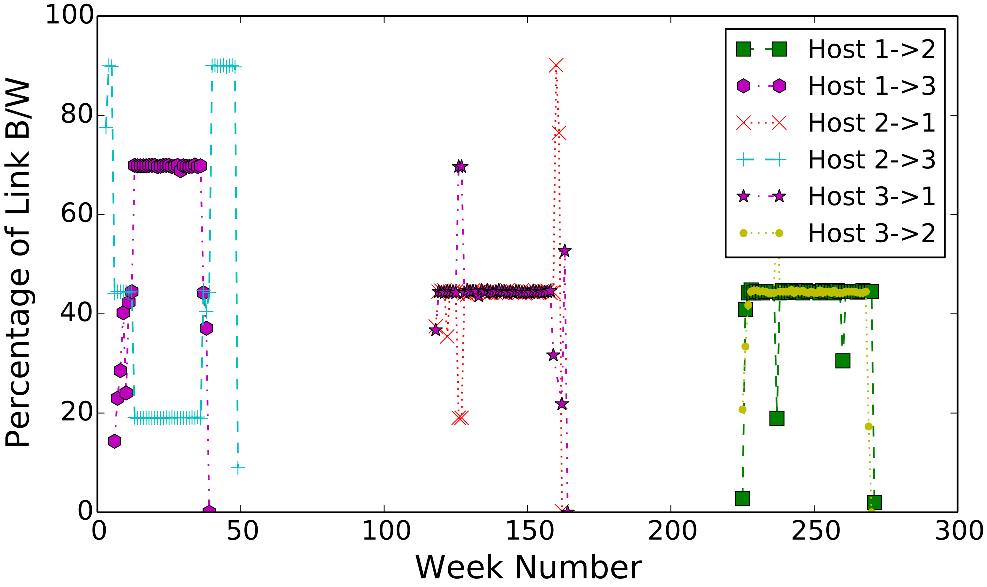


Figure 7.10. Circuit utilization for Hadoop terasort shuffle transfers

terasort job. In the graph, we can see that these shuffle operations can utilize full available circuit bandwidth. The job completion time was same for REACToR and 10Gbps EPS. Ideally, the job should took more time on REACToR when compared to equivalent packet switch but the terasort application is not network limited even when data is stored in memory. The takeaway from this experiment is that applications can run on REACToR without any performance degradation.

Memcached: Next we configured Memcached a latency sensitive application and observe latency on REACToR closed-loop prototype. For this experiment we have a Memcached server running version 1.4.22. We also have a host running Memslap client. The Memslap client, a multi-threaded client, requests objects from the server. For each request client logs the time it takes to get a response back from the server. Our experiment shows that if server response is sent out on the EPS path, latency incurred is not significant. However, if the response is sent on circuit switch path it incurs additional latency of circuit set up time which is 1.5 ms in our current setup.

In summary, we found that REACToR performance for bandwidth-intensive applications is akin to equivalent packet switch. For latency-sensitive applications REACToR incurs additional delay penalty to the flows assigned to the circuit path.

7.6 Summary

The ever-increasing demand for low-cost, high-performance network fabrics in data center environments has generated tremendous interest in alternative switching architectures. Researchers have recently proposed hybrid switches that combine various flavors of circuit and packet switching technologies but, so far, have stopped short of addressing the resulting control loop problem. We propose and experimentally evaluate a practical first-generation closed-loop control plane for a hybrid network. We show how the network traffic demand can be estimated in the host stack and communicated to a controller. We demonstrate that the transport protocol TCP is not affected if the underlying reconfigurable technology is rapidly switching, but performs poorly if reconfigurable topology has paths of different capacity. Finally, we demonstrate that our control loop plane responds to dynamic application demands.

7.7 Acknowledgements

This chapter in part, contains material that has prepared for submission for publication. Liu, He; Kapoor, Rishi; Tewari, Malveeka; Forencich, Alex; Zhang, Sen; Savage, Stefan; Voelker, Geoffrey M.; Papen, George; Snoeren, Alex C.; George, Porter. “Scheduling Circuits in a Packet World”. The dissertation author is the second author on this paper.

This chapter in part, contains material as it appears in Liu, He; Lu, Feng; Forencich, Alex; Kapoor, Rishi; Tewari, Malveeka; Voelker, Geoffrey M.; Papen, George; Snoeren, Alex C.; George, Porter. “Circuit Switching Under the Radar with REACToR”,

11th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Seattle, WA, April 2014 The dissertation author was the fourth author on this paper

Chapter 8

Conclusions and Future Research

As an increasing array of services and applications move to the cloud, Internet data centers must adapt to meet their needs. Unlike software installed on a single machine, data center applications are spread across potentially thousands of hosts. The variety of applications and the scale of operations impose onerous challenge of meeting application performance requirements while maintaining efficiency. To meet this challenge, we harness the fact that a single administrative entity owns the entire end-host stack and the networking interconnect, that allows us to customize the end-host stack to achieve higher levels of efficiency. Our approach has been to conduct a series of systematic measurement studies to identify performance bottleneck(s) in systems and use these observations to design and build more performant and efficient systems.

As we discussed in chapter 3, data center end-host traffic exhibits large bursts at sub-100 microsecond timescales, and these bursts are highly correlated with the size of TSO segments, disk read-ahead settings, and application send sizes. Our results indicate that irrespective of higher layer application behavior, packets come out of a 10-Gbps server in bursts due to batching. We have shown that this short-term packet bursts have implications for the design and performance requirements of packet processing devices along the path, including lower cost reconfigurable topologies.

In chapter 4, we show that end-to-end latency for data center communication patterns is driven by the tail-latency at scale. Thus, it is important to reduce the variance in latency. In chapter 5, we show that operating system accounts for more than 90% of latency inside data centers, and that kernel scheduling and processing increase latency variation, especially at high request loads.

Based on these observations, we designed and built Chronos (chapter 6), a framework to reduce data center application latency especially at the tail. Chronos removes significant sources of application latency by removing the kernel and network stack from the critical path of communication by partitioning requests based on application-level packet header fields in the NIC itself, and by load balancing requests across application instances via an in-NIC load balancing module. Through an evaluation of Memcached, OpenFlow, and a Web search application implemented on Chronos, we show that we can reduce latency by up to a factor of twenty, while significantly reining in latency outliers. Reducing the tail latency of data center applications results in improving efficiency of data center applications since more clients can be served from a limited set of resources. The result is a system that can enable more throughput by increasing predictability, a key contribution to improving data center efficiency.

In Chapter 7, we described a closed-loop system for REACToR, a reconfigurable topology. Through REACToR TCP based experiments we have shown that independent states at the end-host (stateful, single path) and network fabric (stateless, multipath) leads to severe performance degradation. With MPTCP and the demand estimate control plane modifications we demonstrate that network efficiency can be achieved by leveraging the end-host stack and coupling the host stack with network interconnect.

Taken together, we have built systems that demonstrate we can meet performance requirements of data center applications while running data centers at high levels of efficiency.

8.1 Limitations and future research

We highlight the limitations of the proposed system and how these limitation could be overcome in future.

Traffic pattern study: One avenue for future research is to conduct a Bullet-Trains type study in real-world data centers and corroborate the observations. Lack of access to real world data center traces prevented us from conducting the study.

Predictable latency by replicating data: Chronos addresses latency variance by making overall latency as small as possible – but nothing fundamental has changed i.e., close to Chronos peak utilization, the variance is still high (albeit Chronos runs at 10x higher utilization compared to previous systems). Another question we can ask, is it possible to fundamentally change the variance observed by applications aside from just making latency small? One standard technique would be to replicate data across multiple machines, and any application read is actually a read to multiple machines and the system just uses the fastest one. Obviously that's a costly approach because it doubles resource utilization. In future we may want to explore whether the $2x$ increase in network load is worth the substantial increase in predictability.

Other sources of latency: Like any other system the end to end performance of Chronos depends heavily on the network interconnect. Given our improvements in the host, end-to-end latency may become dominated by multiple switch hops, in-network queuing, and congestion losses, a problem worth revisiting.

Circuit scheduling with application hints: In our closed-loop control plane, the demand estimator reports buffer occupancy based on packets queued in the operating system and NIC. Though these demand estimates are accurate but the flows will need to wait for the next scheduling period before being sent out from the host. To reduce the scheduling latency we can predict the network traffic ahead of time. To this end, demand can be obtained directly from the applications ahead of time. An application can provide hints to network about impending transfer e.g., when application reads from storage followed by a network transfer. Similarly, job schedulers like Map-Reduce scheduler can co-ordinate with TDMA scheduler for shuffle transfers. Another approach could be to build a hybrid compile and run time solution for predicting application demand. The compiler can instrument the application binary to provide run time hints about the size of transfer. The size of transfer could be obtained at the run time based on application buffer size for network send call or based on size of the file read from the disk. Network scheduler then combined these demand estimates of different time granularities to obtain a fast and optimal schedule.

Bibliography

[1] Memslap Benchmark. http://docs.libmemcached.org/memslap.html.

[2] 100Gb/s Ethernet Task Force. http://www.ieee802.org/3/ba/.

[3] Amit Aggarwal, Stefan Savage, and Thomas E. Anderson. Understanding the Performance of TCP Pacing. In *Proc. INFOCOM*, 2000.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[5] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, Rong Pan, B. Prabhakar, and M. Seaman. Data center transport mechanisms: Congestion control theory and IEEE standardization. In *CCC*, 2008.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[7] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center. In *Proc. NSDI*, 2012.

[8] Arnold Allen. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. Academic Press, 1978.

[9] Mark Allman and Ethan Blanton. Notes on Burst Mitigation for Transport Protocols. *SIGCOMM Comput. Commun. Rev.*, 35(2):53–60, April 2005.

[10] Ashok Anand, Steven Kappes, Aditya Akella, and Suman Nath. Building Cheap and Large CAMs Using BufferHash. Technical Report TR1651, University of Wisconsin Madison, 2009.

[11] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, 2009.

[12] Apache Software Foundation. HDFS Architecture Guide. http://hadoop.apache.org/docs/hdfs/current/hdfs_design.html.

[13] Jonathan Appavoo, Amos Waterland, Dilma Da Silva, Volkmar Uhlig, Bryan Rosenburg, Eric Van Hensbergen, Jan Stoess, Robert Wisniewski, and Udo Steinberg. Providing a cloud network infrastructure on a supercomputer. HPDC '10.

[14] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L. Peterson. HashCache: Cache Storage for the Next Billion. In *NSDI*, 2009.

[15] Gaurav Banga and Peter Druschel. Measuring the Capacity of a Web Server. In *USENIX USITS*, 1997.

[16] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. IMC*, 2010.

[17] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-Core Key-Value Store. In *IGCC*, 2011.

[18] BerkeleyDB. http://www.oracle.com/technology/products/berkeley-db/index.html/.

[19] Ethan Blanton and Mark Allman. On the Impact of Bursting on TCP Performance. In *Proc. PAM*, 2005.

[20] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *ISCA*, 1994.

[21] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 1995.

[22] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *OSDI*, 2008.

[23] Silas Boyd-Wickizer, Austin Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *OSDI*, 2010.

[24] Sergey Brin and Lawrence Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *WWW Conference*, 1998.

[25] Philip Buonadonna, Andrew Geweke, and David Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *SC*, 1998.

[26] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Commun. ACM*, 2008.

[27] OpenFlow Cbench Controller Benchmark. http://www.openflow.org/wk/index.php/Oflops#Benchmarks.

[28] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[29] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI*, 2004.

[30] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High Throughput Persistent Key-value Store. In *VLDB*, 2010.

[31] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *SIGMOD*, 2011.

[32] 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 *SOSP*, 2007.

[33] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proc. ACM SIGCOMM*, August 2010.

[34] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proc. ACM SIGCOMM*, August 2010.

[35] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *SOSP*, 1997.

[36] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proc. SOSP*, 2003.

[37] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. Trickle: Rate Limiting YouTube Video Streaming. In *Proc. ATC*, 2012.

[38] Apache Hadoop. Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[39] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proc. ACM SIGCOMM*, 2011.

[40] Infiniband. http://www.infinibandta.org/.

[41] Intel. Intel, 82599 10 Gigabit Ethernet Controller: Datasheet. http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html.

[42] R. Jain and S. Routhier. Packet Trains–Measurements and a New Model for Computer Network Traffic. *IEEE J.Sel. A. Commun.*, 4(6):986–995, September 1986.

[43] Hao Jiang and Constantinos Dovrolis. Source-level IP Packet Bursts: Causes and Effects. In *Proc. IMC*, 2003.

[44] Hao Jiang and Constantinos Dovrolis. Why is the Internet Traffic Bursty in Short Time Scales? In *Proc. SIGMETRICS*, 2005.

[45] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 2010.

[46] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. ICPP '11.

[47] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways To De-Congest Data Center Networks. In *Proc. ACM HotNets*, 2009.

[48] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 2010.

[49] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. Architectural Breakdown of End-to-End Latency in a TCP/IP Network. *IJPP*, 2009.

[50] LevelDB: A Fast and Lightweight Key/Value Database Library. http://code.google.com/p/leveldb/.

[51] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *SOSP*, 2011.

[52] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with REACToR. In *Proc. USENIX NSDI*, April 2014.

[53] Memcached. http://memcached.org/.

[54] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, nsdi'13, pages 157–170, Berkeley, CA, USA, 2013. USENIX Association.

[55] Mutrace. http://git.0pointer.de/?p=mutrace.git.

[56] Myricom Sniffer. http://www.myricom.com/sniffer.html.

[57] Radhika Niranjan Mysore, George Porter, Subramanya, and Amin Vahdat. FasTrak: Enabling Express Lanes in Multi-Tenant Data Centers. In *Proc. CoNEXT*, 2013.

[58] ONF. Software-Defined Networking: The New Norm for Networks. https://www.opennetworking.org/.

[59] OpenFlow Controller Source Code. http://www.openflow.org/wp/downloads/.

[60] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 2010.

[61] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized ”Zero-queue” Datacenter Network. In *Proc. ACM SIGCOMM*, August 2014.

[62] IEEE 802.1Qbb Priority Flow Control. http://www.ieee802.org/1/pages/802.1bb.html.

[63] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *Proc. SIGCOMM*, 2013.

[64] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of Interrupt Coalescence on Network Measurements. In *Proc. PAM*, 2004.

[65] Ian Pratt and Keir Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *INFOCOM*, 2001.

[66] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proc. ACM SIGCOMM*, August 2011.

[67] Ashwin Rao, Arnaud Legout, Yeon-sup Lim, Don Towsley, Chadi Barakat, and Walid Dabbous. Network Characteristics of Video Streaming Traffic. In *Proc. CoNEXT*, 2011.

[68] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *Proc. NSDI*, 2011.

[69] Redis. http://redis.io/.

[70] Yaoping Ruan and Vivek S. Pai. The Origins of Network Server Latency & the Myth of Connection Scheduling. In *SIGMETRICS*, 2004.

[71] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's Time for Low Latency. In *HotOS*, 2011.

[72] Avi Rushinek and Sara F. Rushinek. What makes users happy? *Communication ACM*, 29(7), 1986.

[73] Paul Saab. Scaling Memcached at Facebook. http://facebook.com/note.php?note_id=39391378919, 2008.

[74] Vyas Sekar, Norbertand Egi, Sylvia Ratnasamy, Michael K. Reiter, , and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. NSDI*, 2012.

[75] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *Proc. HotNets*, 2004.

[76] SMC SMC10GPCIe-10BT Network Adapter. http://www.smc.com/files/AY/DS_SMC10GPCIe-10BT.pdf.

[77] SolarFlare Solarstorm Network Adapters. http://www.solarflare.com/Enterprise-10GbE-Adapters.

[78] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *ACM PPoPP*, 2010.

[79] A. J. Thadhani. Interactive User Productivity. *IBM Systems Journal*, 20, December 1981.

[80] Niraj Tolia, David G. Andersen, and M. Satyanarayanan. Quantifying Interactive User Experience on Thin Clients. *IEEE Computer*, 39(3), 2006.

[81] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *USENIX ATC*, 2011.

[82] Bhanu Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. Practical TDMA for Datacenter Ethernet. In *Proc. ACM EuroSys*, 2012.

[83] VoltDB. http://voltdb.com/.

[84] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *SOSP*, 1995.

[85] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T. S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-through: Part-time optics in data centers. In *Proc. ACM SIGCOMM*, August 2010.

[86] D. Wischik. Buffer Sizing Theory for Bursty TCP Flows. In *Communications, 2006 International Zurich Seminar on*, pages 98–101, 2006.

[87] Apache Hadoop. http://hadoop.apache.org/.

[88] Takeshi Yoshino, Yutaka Sugawara, Katsushi Inagami, Junji Tamatsukuri, Mary Inaba, and Kei Hiraki. Performance Optimization of TCP/IP over 10 Gigabit Ethernet by Precise Instrumentation. In *Proc. SC*, 2008.

[89] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy H. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. Technical Report UCB/EECS-2012-33, 2012.

[90] Lixia Zhang, Scott Shenker, and Daivd D. Clark. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proc. SIGCOMM*, 1991.

[91] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y. Zhao, and Haitao Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. In *Proc. ACM SIGCOMM*, 2012.