UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Scalable, Adaptive, and Extensible Data Center Network Architecture

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

in

Computer Science

by

Mohammad A. Al-Fares

2012

The dissertation of Mohammad A. Al-Fares is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2012

DEDICATION

لوالدتي الكريمة .. معلمتي الأولى

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I cannot overstate the profound positive impact Prof. Amin Vahdat has had on the trajectory of my life. I do not know what possessed him to agree to take me on as a Ph.D. student, but I am very glad he did. Over the years, I have known him to be one of the most dedicated, hard-working, visionary, and humble people I have ever met. And with his constant encouragement, encyclopedic knowledge, and gentle guidance, he has helped improve these research projects far beyond any degree I thought possible. During group meetings and private discussions, I am constantly amazed by his remarkable and prolific ability to synthesize coherent, solid research directions out of whatever tenuous ideas and research threads his students brought up. For all his immeasurable contributions, on a personal and professional level, I am in his debt.

Working on the component projects of this dissertation, I have been singularly fortunate to collaborate with world-class computer scientists. I would like to thank my incredible co-authors: Alexander Loukissas, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, George Porter, Rishi Kapoor, Sambit Das, Hakim Weatherspoon, and Balaji Prabhakar. I am grateful for their expertise and awed by their skills.

I have had the privilege of working with the members of the DCSwitch research group since its inception. Their level of consistent excellence and teamwork is truly astounding. It has been an absolute pleasure and honor to work with: Nathan Farrington, Radhika Niranjan Mysore, Alex Rasmussen, Meg Walraed-Sullivan, Harsha Madhystha, Vikram Subramanya, Malveeka Tewari, Mike Conley, Pardis Miri, Hamid Bazzaz, Aram Shahinfard, Andreas Pamporis, Eric Rubow, and Harshit Chitalia. Through countless discussions over the years, they have immensely helped improve the technical content and presentation clarity of the ideas in this work.

I also wish to thank the sysnet faculty and students; professors George Varghese, Geoff Voelker, Stefan Savage, and Alex Snoeren, in particular, for their unbridled and infectious enthusiasm for the field. Their passion and dedication are awe-inspiring. Many thanks are due to the 3146 cohort: Kevin Webb, Qing Zhang, Xiao Ma, Diwaker Gupta, Kashi Vishwanath, and James Anderson, for their camaraderie and support. I also thank the incomparable sysnet system administrator, Brian Kantor, for his hard work behind the scenes in keeping our infrastructure running smoothly; the graduate program director, Julie Conner, for always going far above and beyond the call of duty; and Aditya Menon, for his cryptic wit.

On a personal level, this work would not have been possible without Thomas Yi and Dustin Huard. I am very grateful for their unwavering support and encouragement through the years. Finally and above all, I thank my parents and my family for their unconditional faith and support; Khaled Al-Rubie, for leading by example; and my sister Dalal Al-Fares especially, for that Kinder chocolate bar.

Chapters 1, 2, 3, and 6, in part, contain material as it appears in the Proceedings of the ACM Conference on Data Communication (SIGCOMM) 2008. Al-Fares, Mohammad; Loukissas, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapters 1, 2, 4, and 6, in part, contain material as it appears in the Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2010. Al-Fares, Mohammad; Radhakrishnan, Sivasankar; Raghavan, Barath; Huang, Nelson; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapters 1, 2, and 5, and 6, in part, contain material submitted for publication. Al-Fares, Mohammad; Porter, George; Kapoor, Rishi; Das, Sambit; Weatherspoon, Hakim; Prabhakar, Balaji; Vahdat, Amin. The dissertation author was the lead graduate student on this paper.

VITA

2003	B.S. in Computer Engineering, *summa cum laude* Oregon State University
2009	M.S. in Computer Science, University of California, San Diego
2012	Ph.D. in Computer Science, University of California, San Diego

PUBLICATIONS

"Overclocking the Yahoo! CDN for Faster Web Page Loads." Mohammad Al-Fares, Khaled Elmeleegy, Benjamin Reed, Igor Gashinsky. *Proceedings of Internet Measurement Conference* (**IMC**), 2011.

"Scale-Out Networking in the Data Center." Amin Vahdat, Mohammad Al-Fares, Nathan Farrington, Radhika Niranjan Mysore, George Porter, Sivasankar Radhakrishnan. *IEEE Micro*, July/August 2010.

"Hedera: Dynamic Flow Scheduling for Data Center Networks." Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation* (**NSDI**), 2010.

"SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies." Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, Jeffery C. Mogul. *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation* (**NSDI**), 2010.

"A Scalable, Commodity Data Center Network Architecture." Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (**SIGCOMM**), 2008.

"Flexible Resource Allocation and Composition Across GSM/3G Networks and WLANs." Mohammad Al-Fares, Martin Johnsson, Per Johansson, and Amin Vahdat. *Proceedings of the 3rd international Workshop on Mobility in the Evolving Internet Architecture* (**MobiArch**), 2008.

ABSTRACT OF THE DISSERTATION

A Scalable, Adaptive, and Extensible
Data Center Network Architecture

by

Mohammad A. Al-Fares

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Amin Vahdat, Chair

Today's largest data centers contain tens of thousands of servers, and they will encompass hundreds of thousands in the very near future. These machines are designed to serve a rich mix of applications and clients with significant aggregate bandwidth requirements; distributed computing frameworks like MapReduce/Hadoop significantly stress the network interconnect, which when compounded with progressively oversubscribed topologies and inefficient multipath forwarding, can cause a major bottleneck for large computations spanning hundreds of racks. Non-uniform bandwidth among data center nodes also complicates application design and limits overall system performance. Furthermore, using the highest-end, high port-density commercial switches at the core and aggregation layers incurs tremendous cost.

To overcome this limitation, this dissertation advocates three major goals: First, the horizontal, rather than vertical, expansion of data center networks, using commodity off-the-shelf switch components, and rearrangeably non-blocking topologies such as fat-trees. We show that these topologies have several advantages in overall equipment and operational cost and power compared to traditional hierarchical trees. However, the corresponding increase in the degree of multipathing makes traffic forwarding more challenging. Traditional multipath techniques like static hashing (ECMP) can waste bisection bandwidth due to local and downstream hash collisions. To overcome this inefficiency, we next describe the architecture, implementation, and evaluation of Hedera: a centralized flow scheduling system for data center networks with global knowledge of traffic patterns and link utilization. Hedera computes max-min fair flow-bandwidth demands and uses one of several online placement heuristics to find flow paths that maximize the achievable network bisection bandwidth.

Finally, to enable rapid network extensibility, we describe the system architecture and implementation of NetBump: a platform for data-plane modifications "on the wire." By using low-latency kernel bypass and user-level application development, NetBump allows examining, marking, and forwarding packets at line-rate, and enables a host of active queue management disciplines and congestion control mechanisms. This allows the prototyping and adoption of innovative functionality such as DCTCP and 802.1Qau quantized congestion notification (QCN). We show that augmenting top-of-rack switches with NetBumps effectively enables bypassing the slow adoption of data center protocols by commercial switch vendors.

Chapter 1

Introduction

At a rate and scale unforeseen just a few years ago, large organizations, from universities and research labs to large companies, are building progressively more enormous data centers that support tens and hundreds of thousands of machines, and span over 1 million square feet [136]. Imagine an area of over 24 acres, filled with nothing but server racks, networking equipment, HVAC systems, and backup power generators. Other organizations are moving their computation, storage, and operations to cloud-computing hosting providers. One such cloud-computing provider, Amazon's EC2 platform [6], has grown so large that it has become a so-called "accidental" and distributed supercomputer.[1] Important classes of applications being run on this massive scale include scientific computing, financial analysis, data analysis and warehousing, and large-scale network services. Many of these applications—from commodity application hosting to scientific computing to web search and MapReduce—require substantial *intra-cluster* bandwidth. As data centers and their applications continue to scale, scaling the capacity of the network fabric for potential all-to-all communication becomes a compelling engineering challenge.

From a networking research perspective, however, this exciting environment also presents powerful opportunities and advantages not seen before on the open internet. A data center exists under a single administrative domain; the data center operator has full control of all networking infrastructure, all end-host hardware, operating systems,

[1]As of Nov. 2011, EC2 is ranked 42nd in the TOP500 list of the largest supercomputer sites, and the top such site employing a 10G Ethernet interconnect [125].

installed packages, and custom networking settings. This complete control is an unprecedented advantage in large computing clusters, and not previously encountered at this scale outside of dedicated supercomputers.

One interesting complication to the growth story of data centers is the adoption of existing, "road-tested" networking technologies and protocols designed for the internet at large. In particular, the phenomenal and tenacious success of Ethernet in the face of competing interconnect technologies has made it the *de facto* standard in commercial data centers (and still the leading interconnect in use at 45% of the top 500 supercomputing sites [125]). This historic longevity and almost universal adoption have commoditized Ethernet components and made it the most cost-effective choice in terms of performance and compatibility. In addition, data center applications and services are typically developed and deployed on top of existing internet protocols (i.e. TCP/IP); protocols originally designed for wide-area networks and not optimized for an environment with such stringent requirements for ultra-low latency and high bandwidth communication.

This adoption can create, or exacerbate, new classes of performance problems. *Incast* is an example [22], where a server's multiple and simultaneous outgoing requests can create synchronized responses that can overwhelm small switch buffers, causing drops and TCP timeouts. Another important example is the efficient utilization of multiple paths; an aspect that TCP/IP/Ethernet were not designed with in mind. Standard Ethernet requires spanning trees that disallow loops, IP's multipathing support using static hashing is inefficient, and TCP's performance degrades drastically with out-of-order packet arrival when multiple paths are used (an issue we discuss in detail in chapter 4).

1.1 The Evolution of Networking

It is a very exciting time for networking research. Besides the sheer magnitude and scale of the incoming crop of data centers in the next few years, there is a different reason: We are on the cusp of a major shift in how data center and campus networks are engineered and used, and this relates to how the networking industry itself is evolving toward open standards of control.

Throughout the internet boom in the 90's, 2000's, and still today to a large extent, the networking industry has been severely vertically integrated. Everything on high-end switches and routers, all the way from the implementation of supported features and protocols, the switch operating system, and down to the underlying hardware chips, has been developed by its manufacturer, and sold in a proprietary bundle (often at exorbitant margins and with expensive service contracts). One unfortunate consequence of this was that this monolithic and deeply integrated model, combined with a virtual monopoly in the industry, left little room for networking innovation by academia and small startups.[2]

This structure is slowly changing. And this transformation is closely analogous to the computing industry's move from large mainframes of the 70's, with their integrated set of specialized applications, operating systems and hardware, to the personal computers and servers of the 80's and 90's, running a myriad of applications on top of a large set of operating systems on general-purpose, commodity processors. It is argued that the clear separation and open interfaces between the applications and OS, and between the OS and the underlying hardware was a major factor in the rapid innovation and growth in that industry. Frustrated by the difficulty the research community faces in experimentation with large-scale networks (without resorting to small-scale experiments or simulations), networking researchers are calling for the same thing: decoupling network services and control from the underlying physical devices.

This is at the heart of what is termed *Software-Defined Networking* (SDN) [101, 126], which has two basic principles:

1. Software-Defined Forwarding: Switches are abstracted as simple forwarding engines, and they export an open forwarding interface that defines a set of actions (e.g. forward to a given port, drop) to perform on incoming packets.

2. Global Management Abstractions: A network OS would collect the network "view" (the abstract network graph, so that this task would not have to be replicated across

[2]Additionally, incorporation of new functionality was notoriously difficult. The best-case scenario when big customers request new features is a very long lead-time, if considered at all.

applications), and would present possible event triggers (e.g. link failures, addition of a new link/switch, etc.) to running management applications.

The SDN philosophy is gaining significant traction and momentum in the networking community in recent years, both in academia and industry. And while SDN has been a hot area of networking research for some time (e.g. RCP [17] and 4D [50]), perhaps a key turning point was the Ethane [18] project, which is a centralized networking control architecture that allows the application of fine-grained admission and routing policies in enterprise networks. This work eventually led to the formalization of the control protocol in the groundbreaking OpenFlow project [91], which codifies the management API that the physical switches export to the control engine. OpenFlow is quickly becoming an industry-wide standard with support being incorporated into switches made by virtually all major vendors like Juniper, HP ProCurve, IBM Blade, Brocade, Extreme Networks and others.[3] In parallel, research into an OpenFlow-based network OS is becoming more mature with projects like NOX [53], arguably the first such platform with a general-purpose application API, and Onix [78], which aims to be a far more general and robust production-quality system. On the whole, OpenFlow and the SDN paradigm have undeniably sparked a new wave of networking innovation and research, enabling exciting projects like PortLand [97], Hedera [3], Helios [39], ElasticTree [57], DIFANE [140], FlowVisor [115], MiniNet [81], among others too numerous to list. This dissertation certainly would not have been possible without it.

1.2 Data Center Design Challenges

While the emerging area of data center network design and architecture is teeming with exciting unsolved problems and research opportunities, this dissertation attempts to tackle three specific challenges that arise from their exponential growth; namely, 1) Topological Scalability, 2) Forwarding Adaptability, and 3) Rapid Networking Extensibility. We discuss each of these challenges in turn.

[3]Perhaps surprising, but definitely welcome, is switching giant Cisco's plan to support OpenFlow on the Nexus switch series [25].

1.2.1 Topological Scalability

Today, the principal bottleneck in large-scale clusters is often inter-node communication bandwidth. Many applications must exchange information with remote nodes to proceed with their local computation. For example, MapReduce [29] must perform significant data shuffling to transport the output of its map phase before proceeding with its reduce phase. Applications running on cluster-based file systems [30, 47, 111, 138] often require remote-node access before proceeding with their I/O operations. A query to a web search engine often requires parallel communication with every node in the cluster hosting the inverted index to return the most relevant results [15]. Even between logically distinct clusters, there are often significant communication requirements, e.g., when updating the inverted index for individual clusters performing search from the site responsible for building the index. Internet services increasingly employ service oriented architectures [30], where the retrieval of a single web page can require coordination and communication with literally hundreds of individual sub-services running on remote nodes. Finally, the significant communication requirements of parallel scientific applications are well known [23, 112].

There are two high-level choices for building the communication fabric for large-scale clusters. One option leverages specialized hardware and communication protocols, such as InfiniBand [64] or Myrinet [14]. While these solutions can scale to clusters of thousands of nodes with high bandwidth, they do not leverage commodity parts (and are hence more expensive) and are not natively compatible with TCP/IP applications. The second choice leverages commodity Ethernet switches and routers to interconnect cluster machines. This approach supports a familiar management infrastructure along with unmodified applications, operating systems, and hardware. Unfortunately, aggregate cluster bandwidth scales poorly with cluster size, and achieving the highest levels of bandwidth incurs non-linear cost increases with cluster size.

Historically, for compatibility and cost reasons, most cluster communication systems follow the second approach. However, communication bandwidth in large clusters may become oversubscribed by a significant factor depending on the communication patterns. That is, two nodes connected to the same physical switch may be able to communicate at full bandwidth (e.g., 1Gbps) but moving between switches, potentially across multiple levels in a hierarchy, may limit available bandwidth severely. Addressing these bottlenecks requires non-commodity solutions, e.g., large 10Gbps switches and routers. Further, typical single path routing along trees of interconnected switches means that overall cluster bandwidth is limited by the bandwidth available at the root of the communication hierarchy. Even as we are at a transition point where 10Gbps technology is becoming cost-competitive, the highest port-density 10Gbps switches still incur significant cost and still limit overall available bandwidth for the largest clusters.

In this context, the first goal of this dissertation is to design a data center communication architecture that meets the following goals:

- Scalable interconnection bandwidth: it should be possible for an arbitrary host in the data center to communicate with any other host in the network at the full bandwidth of its local network interface.
- Economies of scale: just as commodity personal computers became the basis for large-scale computing environments, we hope to leverage the same economies of scale to make cheap off-the-shelf Ethernet switches the basis for large-scale data center networks.
- Backward compatibility: the entire system should be backward compatible with hosts running Ethernet and IP. That is, existing data centers, which almost universally leverage commodity Ethernet and run IP, should be able to take advantage of the new interconnect architecture with no modifications.

1.2.2 Forwarding Adaptability

However, scaling out the underlying topology is not the only challenge; there are several other properties of cloud-based applications that make the problem of data center network design difficult. First, data center workloads are *a priori* unknown to the network designer and will likely be variable over both time and space. As a result, static resource allocation is insufficient. Second, customers wish to run their software on commodity operating systems; therefore, the network must deliver high bandwidth without requiring software or protocol changes. Third, virtualization technology—commonly used by cloud-based hosting providers to efficiently multiplex customers across physical machines—makes it difficult for customers to have guarantees that virtualized instances of applications run on the same physical rack. Without this physical locality, applications face inter-rack network bottlenecks in traditional data center topologies [26].

Applications alone are not to blame. The routing and forwarding protocols used in data centers were designed for very specific deployment settings. Traditionally, in ordinary enterprise/intranet environments, communication patterns are relatively predictable with a modest number of popular communication targets. There are typically only a handful of paths between hosts and secondary paths are used primarily for fault tolerance. In contrast, recent data center designs *rely* on the path multiplicity to achieve horizontal scaling of hosts [2, 51, 52, 54, 55]. For these reasons, data center topologies are very different from typical enterprise networks.

Some data center applications often initiate connections between a diverse range of hosts and require significant aggregate bandwidth. Because of limited port densities in the highest-end commercial switches, data center topologies often take the form of a multi-rooted tree with higher-speed links but decreasing aggregate bandwidth moving up the hierarchy [26]. These multi-rooted trees have many paths between all pairs of hosts. A key challenge is to simultaneously and dynamically forward flows along these paths to minimize/reduce link oversubscription and to deliver acceptable aggregate bandwidth.

Unfortunately, existing network forwarding protocols are optimized to select a single path for each source/destination pair in the absence of failures. Such static single-path forwarding can significantly underutilize multi-rooted trees with any fanout. State of the art forwarding in enterprise and data center environments uses ECMP [62] (Equal Cost Multipath) to statically stripe flows across available paths using flow hashing. This static mapping of flows to paths does not account for either current network utilization or flow size, with resulting collisions overwhelming switch buffers and degrading overall switch utilization.

1.2.3 Rapid Networking Extensibility

One of the ultimate goals in datacenter networking is predictable, congestion-responsive, low-latency communication. This is a challenging problem and one that requires tight cooperation between end-host protocol stacks, network interface cards, and the switching infrastructure. While there have been a range of interesting ideas in this space, their evaluation and deployment have been hamstrung by the need to develop new hardware to support functionality such as Active Queue Management (AQM) [48, 79], QoS [133], traffic shaping [71], and congestion control [4, 5, 72, 88]. While simulation can show the merits of an idea and support publication, convincing hardware manufacturers to actually support new features requires real-world evidence that a particular technique will actually deliver promised benefits for a range of application and communication scenarios.

Given the significant investment in the data center's networking infrastructure, and the strong disconnect between the need to support rapidly evolving networking requirements and the difficulty of patching in this functionality and protocol support to existing switches and routers, we believe that an experimental platform to support such rapid networking extensibility is absolutely essential to keep up with data center network evolution.

The main requirements for this experimental system would be: rapid prototyping and evaluation, ease of deployment, support for line rate data processing, low latency (i.e., tens of μs), packet marking/transformation for a range of AQM and congestion control policies, and support for distributed deployment to support datacenter multipath topologies. We require low latency since we target LAN switches, rather than WAN router deployments. We expect these bumps on the wire to be part of the production network that will form a proving ground to inform eventual hardware development.

There are several implementation choices for building such a system. While it is possible to develop custom silicon for such functionality, the time to market is long (2-3 years) and the non-recurring engineering costs are often prohibitive for functionality that does not have a pre-existing market. Programmable network processors have been a hot topic for number of years [113]; however, their utility has been hampered by a difficult programming model, which is also the case for FPGA-based designs that are programmed in Verilog or VHDL. The complexity and lead time of these approaches prevent experimenting with novel network programming ideas.

On the other hand, software programmable routers like Click [77] and Route-

Bricks [32] provide a powerful and easy-to-program language for implementing in-network datapath extensions. However, we found that RouteBricks sacrifices latency for higher aggregate throughput (its underlying NIC device driver batches packets). And Click, on the other hand, was designed as a general-purpose software router from the beginning and hasn't been optimized for performance for years [76]. As we are targeting a simpler pass-through AQM system that *augments* existing hardware switches, reproducing the switching functionality is unnecessary and comes at a significant performance cost.

1.3 Hypothesis

We are reaching an inflection point in the design of large computing clusters; the combination of exponential data center growth and the stringent requirements of high bisection-bandwidth, ultra low-latency, and evolving extensibility are pushing the boundaries of the networking community's existing expertise with enterprise and wide-area networks.

Towards achieving these goals, this dissertation aims to prove the following hypothesis: That by combining commodity, off-the-shelf networking equipment with the power of network programmability using open standards, it is possible to overcome all of the following three challenges:

1. Building scalable data center topologies. Specifically, those that can accomodate tens of thousands of servers, and support all-to-all communication at the full speed of the edge.

2. Overcoming forwarding scalability limitations. Namely, the ability to actually achieve the full bisection bandwidth possible for multipath topologies.

3. Support continually and rapidly extensible networking functionality.

In the next section, we describe how we tackle each of these challenges in turn.

1.4 Contributions

1.4.1 A Scalable, Commodity Data Center Architecture

To address the first challenge, we show in chapter 3 that by interconnecting commodity switches in a fat-tree architecture, we can achieve the full bisection bandwidth of clusters consisting of tens of thousands of nodes. Specifically, one instance of our architecture employs 48-port Ethernet switches capable of providing full bandwidth to up 27,648 hosts. By leveraging strictly commodity switches, we achieve lower cost than existing solutions while simultaneously delivering more bandwidth. Our solution requires no changes to end-hosts, is fully TCP/IP compatible, and imposes only modest modifications to the forwarding functions of the switches themselves.

We also show that this approach is the only reasonable and cost-effective way to deliver full bandwidth for large clusters now that 10GigE switches have become commodity at the edge, given the recent ratification of the 40/100 Gigabit Ethernet standards, and the high-cost and limited port-density of these newly available modules.

1.4.2 Dynamic Flow Scheduling

Next, to address the problem of forwarding adaptability, we present Hedera, a dynamic flow scheduling system for multi-stage switch topologies found in data centers. Hedera collects flow information from constituent switches, computes non-conflicting paths for flows, and instructs switches to re-route traffic accordingly. Our goal is to maximize aggregate network utilization—bisection bandwidth—and to do so with minimal scheduler overhead or impact on active flows. By taking a global view of routing and traffic demands, we enable the scheduling system to see bottlenecks that switch-local schedulers cannot.

We completed a full implementation of Hedera on the PortLand testbed [97]. For both our implementation and large-scale simulations, our algorithms deliver performance that is within a few percent of optimal—a hypothetical non-blocking switch—for numerous interesting and realistic communication patterns, and deliver in our testbed up to four times more bandwidth than state of the art ECMP techniques. Hedera delivers these bandwidth improvements with modest control and computation overhead.

One requirement for our placement algorithms is an accurate view of the demand of individual flows under ideal conditions. Unfortunately, due to constraints at the end-host or elsewhere in the network, measuring current TCP flow bandwidth may have no relation to the bandwidth the flow could achieve with appropriate scheduling. Thus, we also present an efficient algorithm to estimate idealized bandwidth share that each flow would achieve under max-min fair resource allocation, and describe how this algorithm assists in the design of our scheduling techniques.

1.4.3 User-extensible Active Queue Management

Finally, in chapter 5, we consider a model where new AQM disciplines can be deployed and evaluated directly in production datacenter networks without modifying existing switches or end-hosts. Instead of adding programmability to existing switches themselves, we instead deploy "bumps on the wire," called *NetBumps*, to augment the existing switching infrastructure. Each NetBump exports a virtual, or phantom, queue primitive that emulates a range of AQM mechanisms [48, 49] at line rate that would normally have to be implemented in the switches themselves.

A contribution of NetBump is providing an efficient and easy way to deploy and manage active queue management separate from switches and end-hosts. NetBump enables AQM functions to be incrementally deployed and evaluated by placement at key points in the network. This makes implementing new functions straightforward. In our experience, new queuing disciplines, congestion control strategies, protocol-specific packet headers (e.g., for XCP [72]), and new packets (for a new congestion control protocol we implement in this work) can be built and deployed at line rate into existing networks. Developers can experiment with protocol specifics by simply modifying software within the bump. Furthermore, we greatly reduce the latency imposed by NetBump because our functionality is limited to modifications of packets in flight, using a user-level, zero-copy, kernel-bypass network API, with no actual queuing or buffering done within NetBump.

Specifically, the primary contributions of chapter 5 are:

- The design of a "bump on the wire," specifically focusing on evaluating and deploying new buffer management packet processing functions.

- A simple virtual Active Queue Management (vAQM) implementation to indirectly manage the buffers of neighboring, unmodified switches.
- Several new programs implemented on top of NetBump, including an implementation of IEEE 802.1Qau-QCN L2 congestion control.
- And finally, an extensible and *distributed* traffic update and management platform for remote physical switch queues.

1.5 Organization

Chapter 2 lays out the required background for the three aforementioned components of this dissertation and their related work.

In chapter 3, we introduce and discuss the fat-tree architecture as it applies to data center networks, as well as the scalability issues and real-world implementation consequences of building data centers based on these designs using commodity off-the-shelf switches.

Next, in chapter 4 we present the idea of dynamic flow scheduling for data center networks. We describe the concept, motivation behind it, and describe the design, implementation and evaluation of Hedera.

In chapter 5, we describe the motivation for and the design, implementation, and evaluation of NetBump, an in-line forwarding framework that enables the rapid development, evaluation and deployment of new network protocols in the data center environment.

And finally, chapter 6 presents avenues of future research as it relates to the aforementioned systems and gives the summary and conclusions of the dissertation.

1.6 Acknowledgement

Chapter 1, in part, contains material as it appears in the Proceedings of the ACM Conference on Data Communication (SIGCOMM) 2008. Al-Fares, Mohammad; Loukissas, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, contains material as it appears in the Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2010. Al-Fares, Mohammad; Radhakrishnan, Sivasankar; Raghavan, Barath; Huang, Nelson; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, contains material submitted for publication. Al-Fares, Mohammad; Porter, George; Kapoor, Rishi; Das, Sambit; Weatherspoon, Hakim; Prabhakar, Balaji; Vahdat, Amin. The dissertation author was the lead graduate student on this paper.

Chapter 2

Background and Related Work

In this chapter, we give some background and historical perspective about current data center networking practices, as well as explore some of the issues that arise from data center multipath forwarding. We finally discuss in § 2.3 some of the related research regarding topological scalability, multipath forwarding, and user-extensible active queue management.

2.1 Current Data Center Practices

We conducted a study to determine the current best practices for data center communication networks. We focus here on commodity designs leveraging Ethernet and IP; we discuss the relationship of our work to alternative technologies in § 2.3.1.

2.1.1 Topology

Typical architectures today consist of either two- or three-level trees of switches or routers. A three-tiered design (Fig. 2.1) has a *core* tier in the root of the tree, an *aggregation* tier in the middle and an *edge* tier at the leaves of the tree. A two-tiered design has only the core and the edge tiers. Typically, a two-tiered design can support between 5K to 8K hosts. Since we target approximately 25,000 hosts, we restrict our attention to the three-tier design, and consider designs that deliver different levels of bandwidth oversubscription.

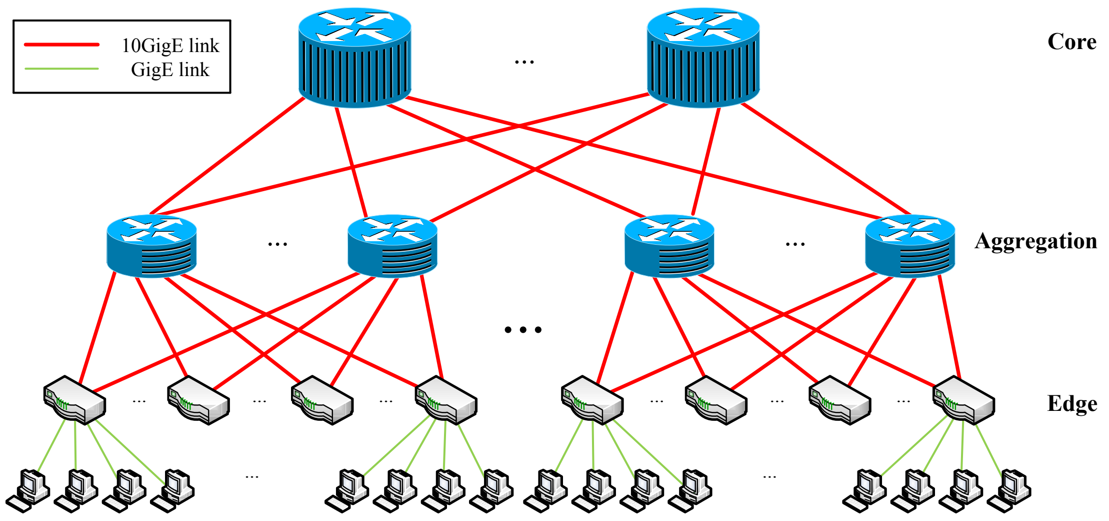


Figure 2.1: A common multi-rooted hierarchical tree. Host to top-of-rack switch links are 1GigE and links between switches are 10GigE.

Switches[1] at the leaves of the tree have some number of GigE ports (48–288) as well as some number of 10GigE uplinks to one or more layers of network elements that aggregate and transfer packets between the leaf switches. In the higher levels of the hierarchy there are switches with 10GigE ports (typically 32–128) and significant switching capacity to aggregate traffic between the edges.

We assume the use of two types of switches, which represent the high-end in both port density and bandwidth. The first, used at the edge of the tree, is a 48-port GigE switch, with four 10GigE uplinks. For higher levels of a communication hierarchy, we consider 10GigE switches with 128-ports or more. Both types of switches allow all directly connected hosts to communicate with one another at the full speed of their network interface.

2.1.2 Oversubscription

Many data center designs introduce oversubscription as a means to lower the total cost of the design. We define the term *oversubscription* to be the ratio of the worst-case achievable aggregate bandwidth among the end-hosts to the total bisection bandwidth of

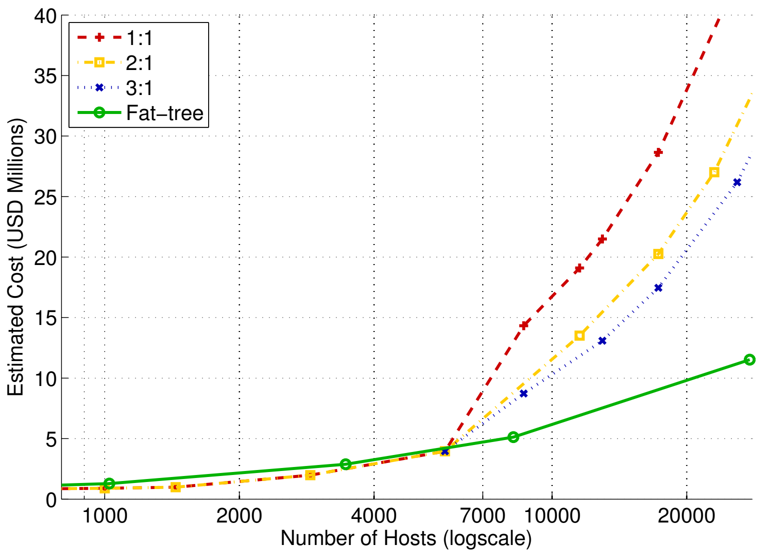
[1]We use the term *switch* throughout the rest of the chapter to refer to devices that perform both layer 2 switching and layer 3 routing.

a particular communication topology. An oversubscription of 1:1 indicates that all hosts may potentially communicate with arbitrary other hosts at the full bandwidth of their network interface (e.g. 1Gbps for commodity Ethernet designs). An oversubscription value of 5:1 means that only 20% of available host bandwidth is available for some communication patterns. Typical designs are oversubscribed by factors from 2.5:1 (400Mbps) and 8:1 (125Mbps) [26], all the way up to 80:1 (12.5Mbps) and 240:1 (4.2Mbps) [51]. Although data centers with oversubscription of 1:1 are possible for 1Gbps Ethernet, as we discuss in § 2.1.3, the cost for such designs is typically prohibitive, even for modest-size data centers.

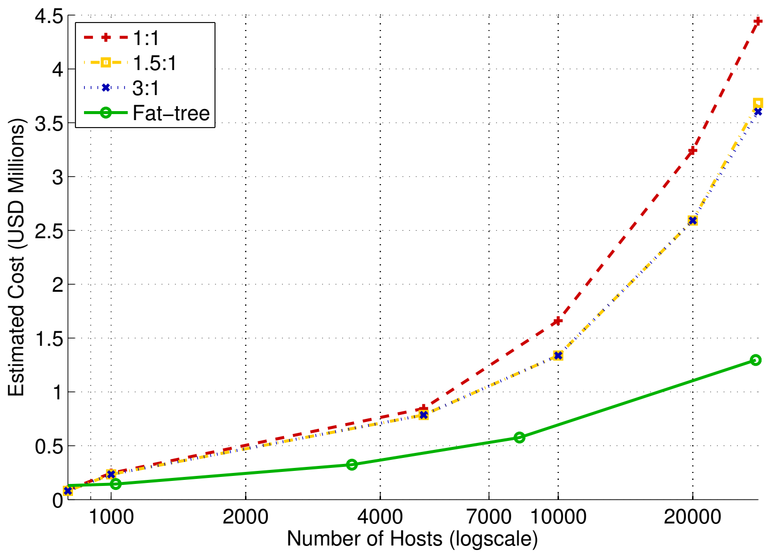
2.1.3 Network Interconnect Cost Trends

The cost for building a network interconnect for a large cluster greatly affects design decisions. As we discussed above, oversubscription is typically introduced to lower the total cost. Here we give the rough cost of various configurations for different number of hosts and oversubscription using current best practices. We do not consider cabling costs in these calculations.

Fig. 2.2 plots the cost in millions of US dollars as a function of the total number of end-hosts on the x-axis, both for 2008 and 2012. Each curve represents a target oversubscription ratio. For instance, in 2008 the switching hardware to interconnect 20,000 hosts, using a traditional hierarchical tree and full bandwidth among all hosts, came to approximately \$34M, and came down to \$3.2M in 2012. The curve corresponding to an oversubscription of 3:1 plots the cost to interconnect end-hosts where the maximum available bandwidth for arbitrary end-host communication would be limited to approximately 330Mbps. We also include the cost to deliver an oversubscription of 1:1 using our fat-tree architecture we propose in chapter 3 for comparison.



(a) Cost in 2008.



(b) Cost in 2012.

Figure 2.2: Cost estimate vs. number of hosts for different oversubscription ratios in 2008 and 2012. Note the dramatic drop in the y-axis.

Overall, we find that existing techniques for delivering high levels of bandwidth in large clusters incur significant cost and that fat-tree based cluster interconnects hold significant promise for delivering scalable bandwidth at moderate cost. However, in some sense, Fig. 2.2 understates the difficulty and expense of employing the highest-end components in building data center architectures. In 2008, 10GigE switches were on the verge of becoming commodity parts; there was roughly a factor of 5 differential in price per port per bit/sec when comparing GigE to 10GigE switches, and this differential continues to shrink. To explore the historical trend, we show in Table 2.1 the cost of the largest cluster configuration that could be supported using the highest-end switches available in a particular year. We based these values on a historical study of product announcements from various vendors of high-end 10GigE switches from 2002-2012.

We use our findings to build the largest cluster configuration that technology in that year could support while maintaining an oversubscription of 1:1. Table 2.1 shows the largest 10GigE switch available in a particular year; we employ these switches in the core and aggregation layers for the hierarchical design. Table 2.1 also shows the largest commodity GigE switch available in that year; we employ these switches at all layers of the fat-tree and at the edge layer for the hierarchical design. For comparison, since 10GigE is virtually commodity now and being pushed into the end-hosts, we also show in Table 2.2 the size and cost-per-port of the largest networks required to deliver 1:1 oversubscription to 10GigE end-hosts in 2012.

The maximum cluster size supported by traditional techniques employing high-end switches has been limited by available port density until recently. Further, the high-end switches incurred prohibitive costs when 10GigE switches were first introduced. Note that we are being somewhat generous with our calculations for traditional hierarchies since commodity 10GigE switches at the aggregation layer did not have the necessary 40GigE uplinks until quite recently. Clusters based on fat-tree topologies on the other hand scale well, with the total cost dropping more rapidly and earlier (as a result of following commodity pricing trends earlier). Also, there is no requirement for higher-speed uplinks in the fat-tree topology.

It is interesting to note that, back in 2008, it was technically infeasible to build a 27,648-node cluster with 10Gbps bandwidth potentially available among all nodes using

Table 2.1: The maximum possible cluster size, for 1GigE hosts and an oversubscription ratio of 1:1 for different years. Note the 100-fold decrease in per-port cost for 1GigE from 2002 to 2012.

Hierarchical design				Fat-tree		
Year	**10GigE**	**Hosts**	**Cost/ GigE**	**GigE**	**Hosts**	**Cost/ GigE**
	4,480	$25.3K	28-port	5,488	$4.5K	
	7,680	$4.4K	48-port	27,648	$1.6K	
	10,240	$2.1K	48-port	27,648	$1.2K	
128-port	20,480	$1.8K	48-port	27,648	$417	
140-port	98,000	$1.0K	48-port	27,648	$312	
768-port	2,949,120	$435	48-port	27,648	$47	
-	-	96-port	221,184	$302		

Table 2.2: The maximum possible cluster size, for 10GigE hosts and an oversubscription ratio of 1:1.

Hierarchical design				Fat-tree		
Year	**40GigE**	**Hosts**	**Cost/ 10GigE**	**10GigE**	**Hosts**	**Cost/ 10GigE**
96-port	46,080	$2,094	48-port	27,648	$520	

aggregation. This was because Ethernet standards faster than 10GigE were not yet complete at that time. However, the 802.3ba standard that defines 40GigE and 100GigE speeds was ratified in June 2010 [63], and vendors finally began shipping 40/100GigE modules to their high-end core switch offerings in 2011.

Furthermore, as 10GigE NICs are now being considered for the end servers, providing non-blocking bandwidth capacity using traditional hierarchical designs using 40GigE switches for aggregation and core layers is still prohibitively expensive and carries a four-fold per-port cost premium over the proposed fat-tree designs (Table 2.2). A fat-tree switch architecture for 27,648 hosts with 10GigE NICs would leverage now-commodity 48-port 10GigE switches and incur a cost of $14.4M (or about $690M back in 2008).

Perhaps the most important point we would like to make in this section is that network designs that depend on aggregation with just-introduced technology would be cost-prohibitive and not economically sound for large deployments, as it takes a few years for these components' prices to come down. When the aggregation technology finally becomes commodity too, we can see a significant drop in the price difference—in absolute terms—compared to fat-trees. However, by that time this "previous-generation" technology migrates to the end servers, thereby repeating the cycle, as the proposal and approval process for new standards seems to be slower than this technology migration from the core to the edge.

2.2 Adaptive Forwarding

The recent development of powerful distributed computing frameworks such as MapReduce [29], Hadoop [10] and Dryad[2] [65] as well as web services such as search, e-commerce, and social networking have led to the construction of massive computing clusters composed of commodity-class servers. Simultaneously, we have witnessed unprecedented growth in the size and complexity of datasets, up to several petabytes, stored on tens of thousands of machines [47].

These cluster applications can often be bottlenecked on the network, not by local resources [12, 20, 30, 47, 51]. Hence, improving application performance may hinge on improving network performance. With the push to build larger data centers encompassing tens of thousands of machines, recent research advocates the horizontal—rather than vertical—expansion of data center networks [51,52]; instead of using expensive core routers with higher speeds and port-densities, networks will leverage a larger number of parallel paths between any given source and destination edge switches.

Thus we find ourselves at an impasse—with network designs using multi-rooted topologies that have the potential to deliver full bisection bandwidth among all communicating hosts, but without an efficient protocol to forward data within the network or a scheduler to appropriately allocate flows to paths to take advantage of this high degree of parallelism.

[2]The world of big-data processing systems seems to be consolidating; Microsoft announced in Nov. 2011 its move to discontinue development of Dryad in favor of a Windows Azure implementation of Hadoop [8].

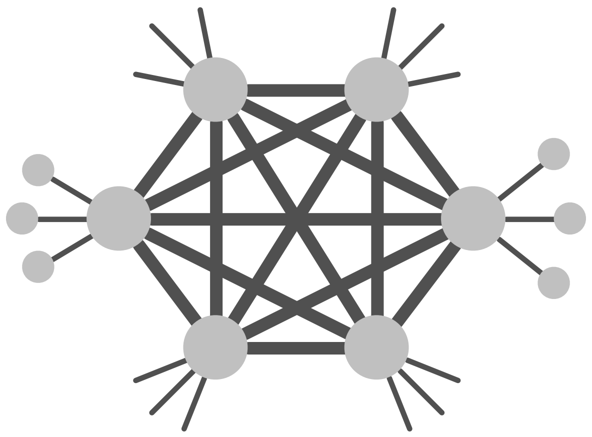
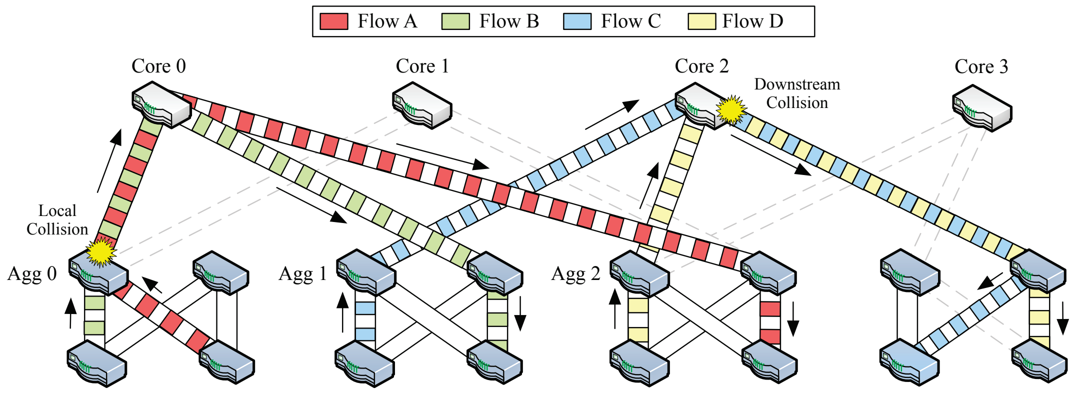


Figure 2.3: An example of a Valiant Load Balancing mesh. Traffic is first forwarded to random intermediate switches, and from there to its destination switch.

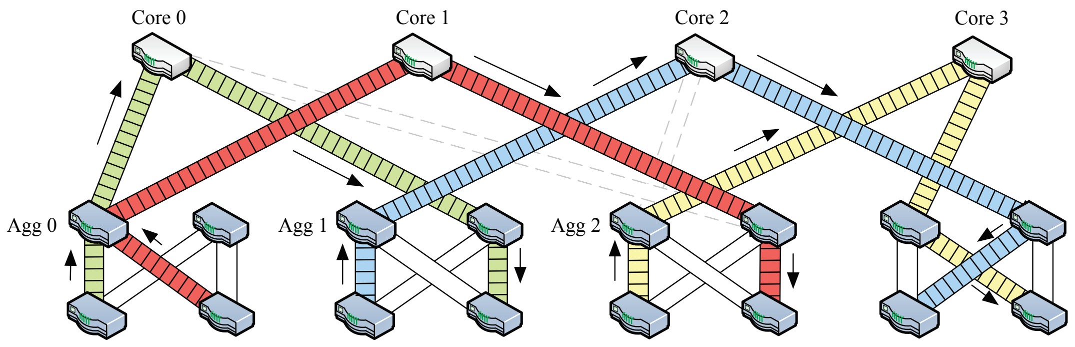
2.2.1 Current Data Center Multipathing

To take advantage of multiple paths in data center topologies, the current state of the art is to use Equal-Cost Multi-Path forwarding (ECMP) [26]. ECMP-enabled switches are configured with several possible forwarding paths for a given subnet. When a packet with multiple candidate paths arrives, it is forwarded on the one that corresponds to a hash of selected fields of that packet's headers modulo the number of paths [62], splitting load to each subnet across multiple paths. This way, a flow's packets all take the same path, and their arrival order is maintained (TCP's performance is significantly reduced when packet reordering occurs because it interprets that as a sign of packet loss due to network congestion).

A closely-related method is Valiant Load Balancing (VLB) [51, 52, 130], shown in Fig. 2.3. VLB essentially guarantees equal-spread load-balancing in a mesh network by bouncing individual packets from a source switch in the mesh off of randomly chosen intermediate switch, which finally forwards those packets to their destination switch. In the data center context, the intermediate switches would be the core switches. Recent realizations of VLB [51] perform randomized forwarding on a per-flow rather than on a per-packet basis to preserve packet ordering. Note that per-flow VLB becomes effectively equivalent to ECMP.



(a) ECMP forwarding.



(b) With scheduled flows.

Figure 2.4: Example of the effect of ECMP local and downstream collisions on bisection bandwidth. Unused links omitted for clarity.

A key limitation of ECMP is that two or more large, long-lived flows can collide on their hash and end up on the same output port, creating an avoidable bottleneck as illustrated in Fig. 2.4(a). Here, we consider a sample communication pattern among a subset of hosts in a multi-rooted, 1Gbps network topology. We identify two types of collisions caused by hashing. First, TCP flows *A* and *B* interfere locally at switch *Agg0* due to a hash collision and are capped by the outgoing link's 1Gbps capacity to *Core0*. Second, with downstream interference, *Agg1* and *Agg2* forward packets independently and cannot foresee the collision at *Core2* for flows *C* and *D*.

In this example, all four TCP flows could have reached capacities of 1Gbps with improved forwarding; flow *A* could have been forwarded to *Core1*, and flow *D* could have been forwarded to *Core3* (Fig. 2.4(b)). But due to these collisions, all four flows are

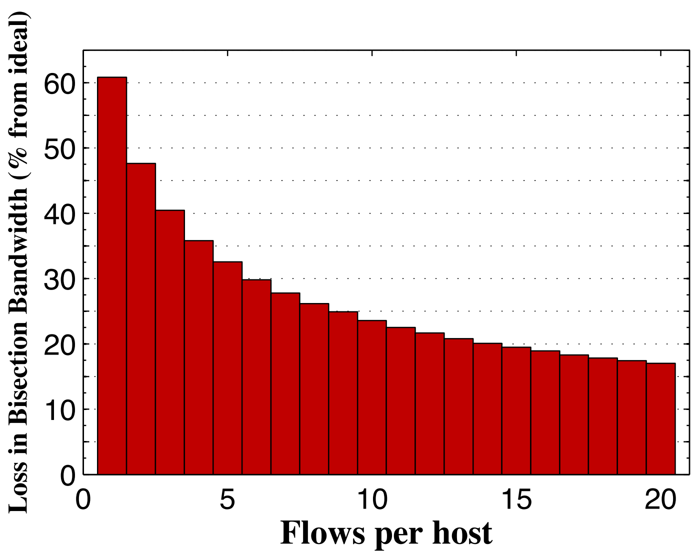


Figure 2.5: Example of ECMP bisection bandwidth losses vs. number of TCP flows per host for a k=48 fat-tree.

bottlenecked at a rate of 500Mbps each, a 50% bisection bandwidth loss.

Note that the performance of ECMP and flow-based VLB intrinsically depends on flow size and the number of flows per host. Hash-based forwarding performs well in cases where hosts in the network perform all-to-all communication with one another simultaneously, or with individual flows that last only a few RTTs. Non-uniform communication patterns, especially those involving transfers of large blocks of data, require more careful scheduling of flows to avoid network bottlenecks.

We defer a full evaluation of these trade-offs to chapter 4, however we can capture the intuition behind performance reduction of hashing with a simple Monte Carlo simulation. Consider a 3-stage fat-tree composed of 1GigE 48-port switches, with 27k hosts performing a data shuffle. Flows are hashed onto paths and each link is capped at 1GigE. If each host transfers an equal amount of data to all remote hosts one at a time, hash collisions will reduce the network's bisection bandwidth by an average of 60.8% (Fig. 2.5). However, if each host communicates to remote hosts in parallel across 1,000 simultaneous flows, hash collisions will only reduce total bisection bandwidth by 2.5%. The intuition here is that if there are many simultaneous flows from each host, their individual rates will be small and collisions will not be significantly costly: each link has 1,000 slots to fill and performance will only degrade if substantially more than 1,000 flows hash to the same link. Overall, Hedera *complements* ECMP, supplementing default ECMP behavior for communication patterns that cause ECMP problems.

2.2.2 Data Center Traffic Patterns

Currently, since no data center traffic traces are publicly available due to privacy and security concerns, we generate patterns along the lines of traffic distributions in published work to emulate typical data center workloads for evaluating our techniques. We also create synthetic communication patterns likely to stress data center networks. Recent data center traffic studies [12, 51, 70] show tremendous variation in the communication matrix over space and time; a typical server exhibits many small, transactional-type RPC flows (e.g. search results), as well as few large transfers (e.g. backups, backend operations such as MapReduce jobs). We believe that the network fabric should be robust to a range of communication patterns and that application developers should not be forced to match their communication patterns to what may achieve good performance in a particular network setting, both to minimize development and debugging time and to enable easy porting from one network environment to another.

Therefore we focus in chapter 4 on generating traffic patterns that stress and saturate the network, and comparing the performance of Hedera to current hash-based multipath forwarding schemes.

2.2.3 Dynamic Flow Demand Estimation

Fig. 2.4 illustrates another important requirement for any dynamic network scheduling mechanism. The straightforward approach to find a good network-wide schedule is to measure the utilization of all links in the network and move flows from highly-utilized links to less utilized links. The key question becomes which flows to move. Again, the straightforward approach is to measure the bandwidth consumed by each flow on constrained links and move a flow to an alternate path with sufficient capacity for that flow. Unfortunately, a flow's current bandwidth may not reflect actual demand. We define a TCP flow's *natural* demand to mean the rate it would grow to in a fully non-blocking network, such that eventually it becomes limited by either the sender or receiver NIC speed. For example, in Fig. 2.4(a), all flows communicate at 500Mbps, though all could communicate at 1Gbps with better forwarding, as shown in Fig. 2.4(b). In chapter 4, we show how to efficiently estimate the natural demands of flows to better inform Hedera's placement algorithms.

2.3 Related Work

2.3.1 Topological Scalability

Our work in data center network architecture necessarily builds upon work in a number of related areas. Perhaps most closely related to our efforts are various efforts in building scalable interconnects, largely coming out of the supercomputer and massively parallel processing (MPP) communities. Many MPP interconnects have been organized as fat-trees, including systems from Thinking Machines [82,127] and SGI [135]. Thinking Machines employed pseudo-random forwarding decisions to perform load balancing among fat-tree links. While this approach achieves good load balancing, it is prone to packet reordering. Myrinet switches [14] also employ fat-tree topologies and have been popular for cluster-based supercomputers. Myrinet employs source routing based on predetermined topology knowledge, enabling cut-through low latency switch implementations. Hosts are also responsible for load balancing among available routes by measuring round-trip latencies. Relative to all of these efforts, we focus on leveraging commodity Ethernet switches to interconnect large-scale clusters, showing techniques for appropriate routing and packaging.

InfiniBand [64] is a popular interconnect for high-performance computing environments and is currently migrating to data center environments. InfiniBand also achieves scalable bandwidth using variants of Clos topologies. For instance, Sun recently announced a 3,456-port InfiniBand switch built from 720 24-port InfiniBand switches arranged in a 5-stage fat-tree [123]. However, InfiniBand imposes its own layer 1-4 protocols, making Ethernet/IP/TCP more attractive in certain settings especially as the price of 10Gbps Ethernet continues to drop.

Another popular MPP interconnect topology is a Torus, for instance in the BlueGene/L [13] and the Cray XT3 [131]. A torus directly interconnects a processor to some number of its neighbors in a k-dimensional lattice. The number of dimensions determines the expected number of hops between source and destination. In an MPP environment, a torus has the benefit of not having any dedicated switching elements along with electrically simpler point-to-point links. In a cluster environment, the wiring complexity of a torus quickly becomes prohibitive and offloading all routing and forwarding functions to commodity hosts/operating systems is typically impractical.

Our proposed forwarding techniques are related to existing routing techniques such as OSPF2 and Equal-Cost Multipath (ECMP) [62, 94, 124]. Our proposal for multipath leverages particular properties of a fat-tree topology to achieve good performance. Relative to our work, ECMP proposes three classes of stateless forwarding algorithms: (i) Round-robin and randomization; (ii) Region splitting where a particular prefix is split into two with a larger mask length; and (iii) A hashing technique that splits flows among a set of output ports based on the source and destination addresses. The first approach suffers from potential packet reordering issues, especially problematic for TCP. The second approach can lead to a blowup in the number of routing prefixes. In a network with 25,000 hosts, this will require approximately 600,000 routing table entries. In addition to increasing cost, the table lookups at this scale will incur significant latency. For this reason, current enterprise-scale routers allow for a maximum of 16-way ECMP routing. The final approach does not account for flow bandwidth in making allocation decisions, which can quickly lead to oversubscription even for simple communication patterns.

2.3.2 Data Center Multipath Forwarding

There has been a recent flood of new research proposals for data center networks; however, none satisfyingly addresses the issue of the network's bisection bandwidth. VL2 [51] and Monsoon [52] propose using per-flow Valiant Load Balancing, which can cause bandwidth losses due to long-term collisions as demonstrated in this work. SEATTLE [75] proposes a single Layer 2 domain with a one-hop switch DHT for MAC address resolution, but does not address multipathing. DCell [55] and BCube [54] suggest using recursively-defined topologies for data center networks, which involves multi-NIC servers and can lead to oversubscribed links with deeper levels. Once again, multipathing is not explicitly addressed.

Researchers have also explored scheduling flows in a multi-path environment from a wide-area context. TeXCP [69] and MATE [36] perform dynamic traffic engineering across multiple paths in the wide-area by using explicit congestion notification packets, which require as yet unavailable switch support. They employ *distributed* traffic engineering, whereas we leverage the data center environment using a tightly-coupled central scheduler. FLARE [117] proposes multipath forwarding in the wide-area on the granularity of *flowlets* (TCP packet bursts); however, it is unclear whether the low intra-data center latencies meet the timing requirements of flowlet bursts to prevent packet reordering and still achieve good performance. Recent efforts towards making TCP more suitable for multi-path environments and resistent to packet reordering are indeed very promising; especially the ongoing research and standardization effort on Multi-path TCP (MPTCP) [105]. When used in conjunction with per-packet Valiant Load Balancing, multipath TCP has the potential to dramatically increase the utilization efficiency of multipath data center topologies.

Miura *et al.* exploit fat-tree networks by multipathing using tagged-VLANs and commodity PCs [93], similar to Ethernet multipathing efforts like SPAIN [95] and VIKING [114]. Centralized router control to enforce routing or access control policy has been proposed before by the 4D architecture [50], and projects like Tesseract [139], Ethane [18], and RCP [17], similar in spirit to Hedera's approach to centralized flow scheduling.

Much work has focused on virtual switching fabrics and on individual Clos networks in the abstract, but do not address building an operational multi-level switch architecture using existing commodity components. Turner proposed an optimal non-blocking virtual circuit switch [129], and Smiljanić improved Turner's load balancer and focused on the guarantees the algorithm could provide in a generalized Clos packet-switched network [119]. Oki *et al.* design improved algorithms for scheduling in individual 3-stage Clos switches [99], and Holmburg provides models for simultaneous and incremental scheduling of multi-stage Clos networks [60]. Geoffray and Hoefler describe a number of strategies to increase bisection bandwidth in multistage interconnection networks, specifically focusing on source-routed, per-packet dispersive approaches that break the ordering requirement of TCP/IP over Ethernet [46].

2.3.3 User-extensible Active Queue Management

This section describes previous work that we build upon to design and implement NetBump.

Virtual Queuing and AQM:

In virtual queuing (VQ), metadata about an incoming packet stream is maintained to simulate the behavior of those packets in a hypothetical physical queue. We differ from previous work in that we maintain VQs outside of the switch itself. VQ provides a basis for a variety of active queue management (AQM) techniques. AQM manipulates packets in buffers in the network to enact changes in the control loop of that traffic, typically to anticipate and reduce packet drops and queue overflows, or reduce buffer sizes. A large amount of work examined AQM in a variety of settings [59,92]. One proposal, Active Virtual Queue [79], reduces queue sizes in traffic with small flows, which typically pose challenges for the TCP control loop. Random early detection (RED) [58] signals congestion by dropping packets with a particular probability as congestion builds, and unconditionally dropping packets after a certain threshold. Due to the inefficiency of dropping packets to signal congestion, the early congestion notification ECN [80] field was developed to decouple packet drops from congestion indicators. Several proposals for improving on RED have been made [11], including Data Center TCP (DCTCP) [5].

Quantized congestion notification [4,88] was proposed as an L2 congestion control mechanism. QCN tries to ensure that a switch buffer stays below a configurable maximum size. QCN provides congestion control for non-TCP traffic, and can respond faster than the round-trip time. Implementations of QCN have been developed on 1Gbps networks [87], as well as emulated within FPGAs at 10Gbps networks [1,98]. Our deployment is done at 10Gbps and distributed across multiple network hops. Approximate-Fairness QCN (AF-QCN) [68] is an extension that modifies notifications to input links weighted by the ratio of their queue occupancy.

Datapath Programming in Software:

Software-based packet switches and routers have a long history as a platform for rapidly developing new functionality. The Click Modular Router [77] is a pipeline-oriented, modular software router consisting of a large number of building blocks, each performing a simple packet-handling task. Click's library of modules can be extended by writing code in C++ designed to work in the Linux kernel or userspace. The Route-Bricks [32] project has focused on scaling out a Click runtime to support forwarding rates in excess of tens of Gbps by relying on distribution of packet processing across cores, as well as across a small cluster of servers. ServerSwitch [87], is another recent software router design that allows programming commodity Ethernet switching chips (with matching/modification of standard header fields), but delegates general packet processing to the CPU (e.g. for XCP). Besides avoiding the associated latency of crossing the kernel/user-space boundary, NetBump leverages kernel-bypass to allow *arbitrary* packet modification to support new protocol headers at line rate. A key distinction from these projects is that Click, RouteBricks and ServerSwitch are all multi-port software switches focused on packet routing, while NetBump focuses on pass-through virtual queuing within a pre-existing switching layer.

SideCar [116], on the other hand, is a recent proposal to delegate a small fraction of traffic requiring special processing from the ToR switch to a companion server. While superficially similar, the redirection and traffic sampling are not applicable for NetBump's vAQM use-case, where low-latency is a key design requirement. For these reasons, we consider these efforts to be orthogonal to this work.

Several efforts have looked at ways of mapping packet handling tasks necessary to support software routers to multi-core, multi-NIC queue commodity servers. Egi et al. [35], and Dobrescu et al. [31] investigate the effects of casting forwarding paths across multiple cores, and find that minimizing core transitions is necessary for high performance. NetBump takes a similar approach to the "split traffic" and "cloning" functionality described, in which an entire forwarding path resides on a single core and cache hierarchy. Manesh et al. [90] study the performance of multi-queue NICs as applied to packet forwarding workloads.

Typically the OS kernel translates streams of raw packets to and from a higher-level interface such as a socket. While a useful primitive, the involvement of the kernel can become a bottleneck and an alternative set of user-level networking techniques have been developed [16, 37, 132, 134]. Here, user-space programs are responsible for TCP sequence reassembly, retransmission, etc. User-level networking is typically coupled with zero-copy buffering, in which the memory that a packet is initially stored in is shared with target applications. Kernel-bypass drivers also enable applications to directly access packets from NIC memory, avoiding kernel involvement on the datapath. Commercially-available NICs already support these mechanisms [21, 96, 104, 118, 121].

Datapath Programming in Hardware:

One drawback of software-based packet forwarding is that it historically suffered from low performance, and alternative hardware architectures have been proposed. Perhaps the best-known and widely-used hardware forwarding platform is the NetFPGA [86], a powerful development tool for FPGA devices. However, the complexity of FPGA programming remains a challenge. Two recent projects sought to address this: Switchblade [9] provides modular building blocks that can support a wide variety of datapaths, and Chimpp [109] converts datapaths specified in the Click language into Verilog code suitable for an FPGA.

In addition, network processors (NPs) [113] have been used to prototype and deploy new network functionality. They have the disadvantage of a difficult-to-use programming model and limited production runs. Their primary advantage is their multiple functional units, providing significant parallelism to support faster data rates. Commodity CPUs have since greatly increased their number of cores, and can also provide significant per-packet processing at high line rates.

2.4 Acknowledgement

Chapter 2, in part, contains material as it appears in the Proceedings of the ACM Conference on Data Communication (SIGCOMM) 2008. Al-Fares, Mohammad; Loukissas, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, contains material as it appears in the Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2010. Al-Fares, Mohammad; Radhakrishnan, Sivasankar; Raghavan, Barath; Huang, Nelson;

Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in part, contains material submitted for publication. Al-Fares, Mohammad; Porter, George; Kapoor, Rishi; Das, Sambit; Weatherspoon, Hakim; Prabhakar, Balaji; Vahdat, Amin. The dissertation author was the lead graduate student on this paper.

Chapter 3

A Scalable, Commodity Data Center Network Architecture

Today's data centers may contain tens of thousands of computers with significant aggregate bandwidth requirements. As discussed in chapter 2, the network architecture typically consists of a tree of routing and switching elements with progressively more specialized and expensive equipment moving up the network hierarchy. Unfortunately, even when deploying the highest-end IP switches/routers, resulting topologies may only support a fraction of the aggregate bandwidth available at the edge of the network, due to oversubscription, while still incurring tremendous cost.

Today, the price differential between commodity and non-commodity switches provides a strong incentive to build large-scale communication networks from many small commodity switches rather than fewer larger and more expensive ones. More than fifty years ago, similar trends in telephone switches led Charles Clos to design a network topology that delivers high levels of bandwidth for many end devices by appropriately interconnecting smaller commodity switches [28].

In this chapter, we propose a special instance of a Clos topology called a *fat-tree* [83] to interconnect commodity Ethernet switches. We organize a k-ary fat-tree as shown in Fig. 3.1. There are k pods, each containing two layers of $k/2$ switches. Each k-port switch in the lower layer is directly connected to $k/2$ hosts. Each of the remaining $k/2$ ports is connected to $k/2$ of the k ports in the aggregation layer of the hierarchy.

There are $(k/2)^2$ k-port core switches, and each core switch has one port con-

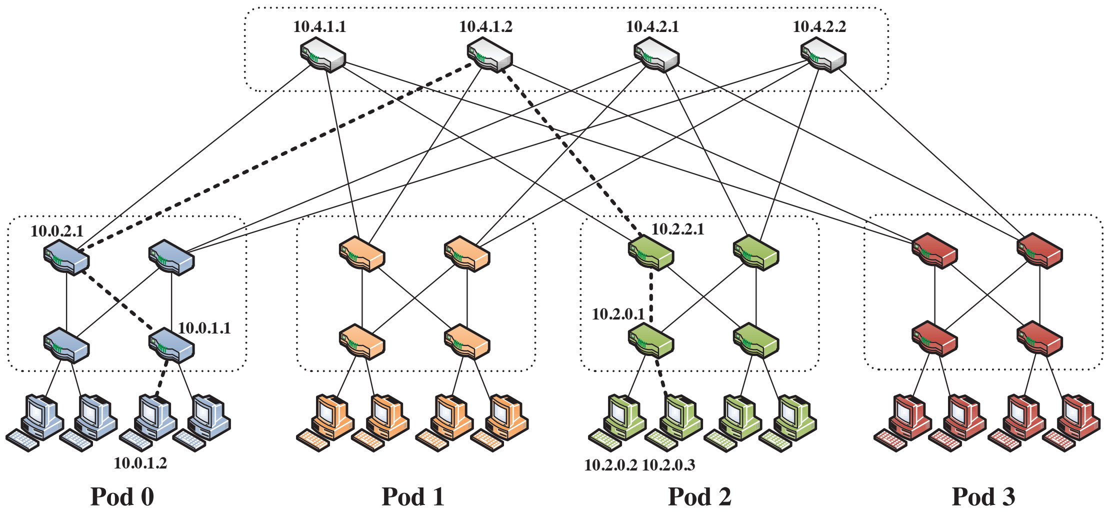
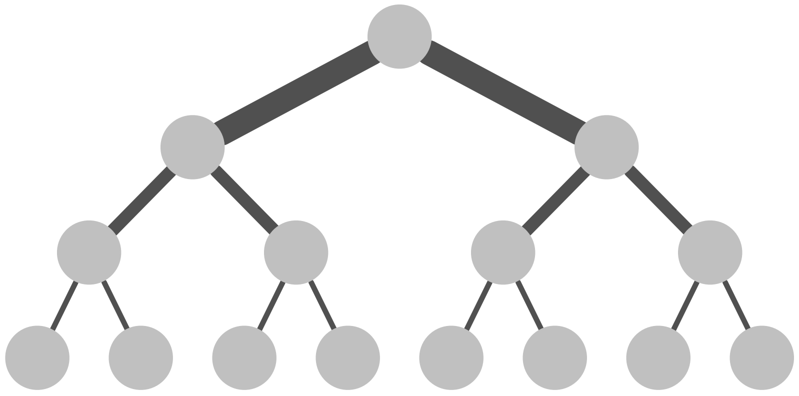


Figure 3.1: Proposed fat-tree topology. Using the two-level routing tables described in § 3.1.3, packets from source 10.0.1.2 to destination 10.2.0.3 would take the dashed path.

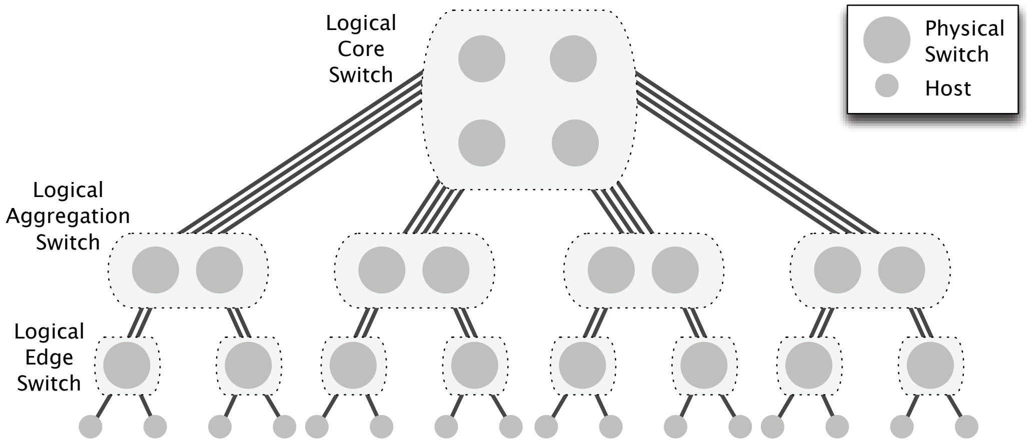
nected to each of k pods. The i^{th} port of any core switch is connected to pod i such that consecutive ports in the aggregation layer of each pod switch are connected to core switches on $(k/2)$ strides. In general, a fat-tree built with k-port switches supports $k^3/4$ hosts. In this chapter, we focus on designs up to $k = 48$. Our approach generalizes to arbitrary values for k.

To clarify the terminology used here, the classical Leiserson "fat-tree" [83] was proposed as a routing network for parallel processors in a supercomputer context, and was strictly defined as a simple tree whose communication channels were progressively thicker towards the root (Fig. 3.2(a)). We borrow this terminology for our data center architecture by using the analogy of "logical switches." In this sense, all the core switches should act as a single non-blocking core switch, and similarly for the aggregation switches in each pod (Fig. 3.2(b)). In reality, since this fat-tree and all multi-rooted trees contain cycles, they are not *trees* in the graph theory sense. Indeed, the topology proposed in this work is a special instance of a 5-stage folded Clos network. However, whereas all traffic traverses the spine (i.e. core) of a Clos network, intra-pod traffic can short-circuit this step in the fat-tree.

An advantage of the fat-tree topology is that all switching elements are identical, enabling us to leverage cheap commodity parts for all of the switches in the communi-



(a) Leiserson's original fat-tree topology.



(b) Fat-tree topology of Fig. 3.1, rearranged with logical, non-blocking core and aggregation switches.

Figure 3.2: Original Leiserson fat-tree vs. proposed topology with logical switches.

cation architecture.[1] Further, fat-trees are *rearrangeably non-blocking*, meaning that for arbitrary communication patterns, there is some set of paths that will saturate all the bandwidth available to the end-hosts in the topology. Achieving an oversubscription ratio of 1:1 in practice may be difficult because of the need to prevent packet reordering for TCP flows.

Fig. 3.1 shows the simplest non-trivial instance of the fat-tree with $k = 4$. All hosts connected to the same edge switch form their own subnet. Therefore, all traffic to a host

[1]Note that switch homogeneity is not required, as bigger switches could be used at the core (e.g. for multiplexing). While these likely have a longer mean time to failure (MTTF), this defeats the cost benefits, and maintains the same cabling overhead.

connected to the same lower-layer switch is switched, whereas all other traffic is routed.

As an example instance of this topology, a fat-tree built from 48-port GigE switches would consist of 48 pods, each containing an edge layer and an aggregation layer with 24 switches each. The edge switches in every pod are assigned 24 hosts each. The network supports 27,648 hosts, made up of 1,152 subnets with 24 hosts each. There are 576 equal-cost paths between any given pair of hosts in different pods. The cost of deploying such a network architecture would be $8.64M, compared to $37M for the traditional techniques described earlier.

Given our target network architecture, in the rest of this chapter we address two principal issues with adopting this topology in Ethernet deployments. First, IP/Ethernet networks typically build a single routing path between each source and destination. For even simple communication patterns, such single-path routing will quickly lead to bottlenecks up and down the fat-tree, significantly limiting overall performance. We describe simple extensions to IP forwarding to effectively utilize the high fan-out available from fat-trees. Second, fat-tree topologies can impose significant wiring complexity in large networks. To some extent, this overhead is inherent in fat-tree topologies, but in § 3.3 we present packaging and placement techniques to ameliorate this overhead. Finally, we have built a prototype of our architecture in Click [77] as described in § 3.1. An initial performance evaluation presented in § 3.5 confirms the potential performance benefits of our approach in a small-scale deployment.

3.1 Architecture

In this section, we describe an architecture to interconnect commodity switches in a fat-tree topology. We first motivate the need for a slight modification in the routing table structure. We then describe how we assign IP addresses to hosts in the cluster. Next, we introduce the concept of two-level route lookups to assist with multi-path routing across the fat-tree. We then present the algorithms we employ to populate the forwarding table in each switch. We also describe flow classification and flow scheduling techniques as alternate multi-path routing methods. And finally, we present a simple fault-tolerance scheme, as well as describe the heat and power characteristics of our approach.

3.1.1 Motivation

Achieving maximum bisection bandwidth in this network requires spreading outgoing traffic from any given pod as evenly as possible among the core switches. Routing protocols such as OSPF2 [94] usually take the hop-count as their metric of "shortest-path," and in the k-ary fat-tree topology, there are $(k/2)^2$ such shortest-paths between any two hosts on different pods, but only one is chosen. Switches, therefore, concentrate traffic going to a given subnet to a single port even though other choices exist that give the same cost. Furthermore, depending on the interleaving of the arrival times of OSPF messages, it is possible for a small subset of core switches, perhaps only one, to be chosen as the intermediate links between pods. This will cause severe congestion at those points and does not take advantage of path redundancy in the fat-tree.

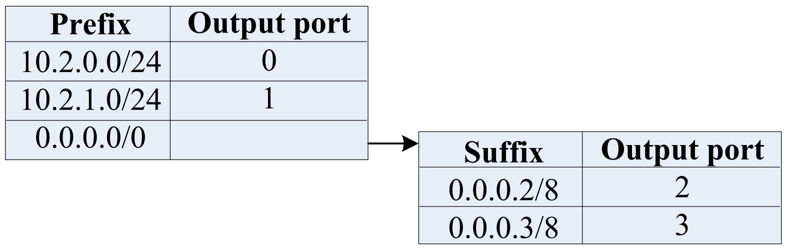
Extensions such as OSPF-ECMP [124], support for which has recently been added in the class of switches under consideration, cause an explosion in the number of required prefixes. These commercial ECMP implementations also restrict the number of possible paths, up to 4 paths in most cases, which is fewer than the fanout required for these topologies.[2] A lower-level pod switch would need $(k/2)$ prefixes for *every* other subnet; a total of $k * (k/2)^2$ prefixes.

We therefore need a simple, fine-grained method of traffic diffusion between pods that takes advantage of the structure of the topology. The switches must be able to recognize, and give special treatment to, the class of traffic that needs to be evenly spread. To achieve this, we propose using two-level routing tables that spread outgoing traffic based on the low-order bits of the destination IP address (see § 3.1.3).

3.1.2 Addressing

We allocate all the IP addresses in the network within the private 10.0.0.0/8 block. We follow the familiar quad-dotted form with the following conditions: The pod switches are given addresses of the form 10.*pod*.*switch*.1, where *pod* denotes the pod number (in $[0, k-1]$), and *switch* denotes the position of that switch in the pod (in $[0, k-1]$, starting from left to right, bottom to top). We give core switches addresses of the form $10.k.j.i$,

[2] At the time of this work's publication [2], this support was not yet available.



Prefix	Output port
10.2.0.0/24	0
10.2.1.0/24	1
0.0.0.0/0	

Suffix	Output port
0.0.0.2/8	2
0.0.0.3/8	3

Figure 3.3: Two-level table example. This is the table at switch 10.2.2.1. An incoming packet with destination IP address 10.2.1.2 is forwarded on port 1, whereas a packet with destination IP address 10.3.0.3 is forwarded on port 3.

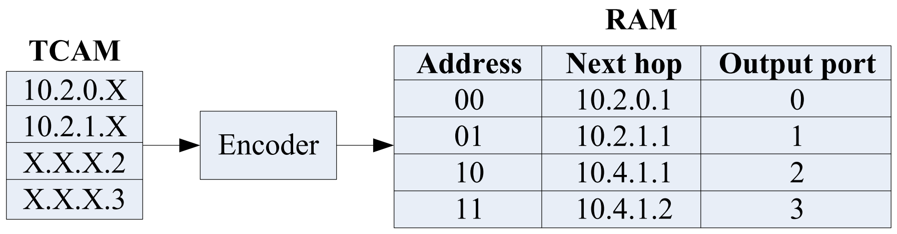
where j and i denote that switch's coordinates in the $(k/2)^2$ core switch grid (each in $[1, (k/2)]$, starting from top-left).

The address of a host follows from the pod switch it is connected to; hosts have addresses of the form: 10.*pod.switch.ID*, where *ID* is the host's position in that subnet (in $[2, k/2+1]$, starting from left to right). Therefore, each lower-level switch is responsible for a /24 subnet of $k/2$ hosts (for $k < 256$). Fig. 3.1 shows examples of this addressing scheme for a fat-tree corresponding to $k = 4$. Even though this is relatively wasteful use of the available address space, it simplifies building the routing tables, as seen below. Nonetheless, this scheme scales up to 4.2M hosts.

3.1.3 Two-Level Routing Table

To provide the even-distribution mechanism motivated in § 3.1.1, we modify routing tables to allow two-level prefix lookup. Each entry in the main routing table will potentially have an additional pointer to a small secondary table of *(suffix, port)* entries. A first-level prefix is *terminating* if it does not contain any second-level suffixes, and a secondary table may be pointed to by more than one first-level prefix. Whereas entries in the primary table are left-handed (i.e., */m prefix* masks of the form 1^m0^{32-m}), entries in the secondary tables are right-handed (i.e. */m suffix* masks of the form $0^{32-m}1^m$). If the longest-matching prefix search yields a non-terminating prefix, then the longest-matching suffix in the secondary table is found and used.

This two-level structure will slightly increase the routing table lookup latency, but the parallel nature of prefix search in hardware should ensure only a marginal penalty (see



Address	Next hop	Output port
00	10.2.0.1	0
01	10.2.1.1	1
10	10.4.1.1	2
11	10.4.1.2	3

Figure 3.4: TCAM two-level routing table implementation.

below). This is helped by the fact that these tables are meant to be very small. As shown below, the routing table of any pod switch will contain no more than $k/2$ prefixes and $k/2$ suffixes.

3.1.4 Two-Level Lookup Implementation

We now describe how the two-level lookup can be implemented in hardware using Content-Addressable Memory (CAM) [24]. CAMs are used in search-intensive applications and are faster than algorithmic approaches [34, 122] for finding a match against a bit pattern. A CAM can perform parallel searches among all its entries in a single clock cycle. Lookup engines use a special kind of CAM, called Ternary CAM (TCAM). A TCAM can store *don't care* bits in addition to matching 0's and 1's in particular positions, making it suitable for storing variable length prefixes, such as the ones found in routing tables. On the downside, CAMs have rather low storage density, they are very power hungry, and expensive per bit. However, in our architecture, routing tables can be implemented in a TCAM of a relatively modest size (k entries each 32 bits wide).

Fig. 3.4 shows our proposed implementation of the two-level lookup engine. A TCAM stores address prefixes and suffixes, which in turn indexes a RAM that stores the IP address of the next hop and the output port. We store left-handed (prefix) entries in numerically smaller addresses and right-handed (suffix) entries in larger addresses. We encode the output of the CAM so that the entry with the numerically smallest matching address is output. This satisfies the semantics of our specific application of two-level lookup: when the destination IP address of a packet matches both a left-handed and a right-handed entry, then the left-handed entry is chosen. For example, using the routing table in Fig. 3.4, a packet with destination IP address 10.2.0.3 matches the left-handed entry 10.2.0.X and the right-handed entry $X.X.X.3$. The packet is correctly forwarded on port 0. However, a packet with destination IP address 10.3.1.2 matches only the right-handed entry $X.X.X.2$ and is forwarded on port 2.

3.1.5 Routing Algorithm

The first two levels of switches in a fat-tree act as filtering traffic diffusers; the lower- and upper-layer switches in any given pod have terminating prefixes to the subnets in that pod. Hence, if a host sends a packet to another host in the same pod but on a different subnet, then all upper-level switches in that pod will have a terminating prefix pointing to the destination subnet's switch.

For all other outgoing inter-pod traffic, the pod switches have a default /0 prefix with a secondary table matching host IDs (the least-significant byte of the destination IP address). We employ the host IDs as a source of deterministic entropy; they will cause traffic to be evenly spread upward among the outgoing links to the core switches.[3] This will also cause subsequent packets to the same host to follow the same path, and therefore avoid packet reordering.

In the core switches, we assign terminating first-level prefixes for all network IDs, each pointing to the appropriate pod containing that network. Once a packet reaches a core switch, there is exactly one link to its destination pod, and that switch will include a terminating /16 prefix for the pod of that packet $(10.\mathit{pod}.0.0/16, \mathit{port})$. Once a packet reaches its destination pod, the receiving upper-level pod switch will also include a $(10.\mathit{pod}.\mathit{switch}.0/24, \mathit{port})$ prefix to direct that packet to its destination subnet switch, where it is finally switched to its destination host. Hence, traffic diffusion occurs only in the first half of a packet's journey.

It is possible to design distributed protocols to build the necessary forwarding state incrementally in each switch. For simplicity however, we assume a central entity with full knowledge of cluster interconnect topology. This central route control is responsible for statically generating all routing tables and loading the tables into the switches at the network setup phase. Dynamic routing protocols would also be responsible for

[3]Since the tables are static, it is possible to fall short of perfect distribution. We examine worst-case communication patterns in § 3.5

detecting failures of individual switches and performing path fail-over (see § 3.1.8). Below, we summarize the steps for generating forwarding tables at both the pods and core switches.

Pod Switches: In each pod switch, we assign terminating prefixes for subnets contained in the same pod. For inter-pod traffic, we add a /0 prefix with a secondary table matching host IDs. Fig. 3.5 shows the pseudo-code for generating the routing tables for the upper pod switches. The reason for the modulo shift in the outgoing port is to avoid traffic from different lower-layer switches addressed to a host with the same host ID going to the same upper-layer switch.

For the lower pod switches, we simply omit the /24 subnet prefix step, in line 3, since that subnet's own traffic is switched, and intra- and inter-pod traffic should be evenly split among the upper switches.

Core Switches: Since each core switch is connected to every pod (port i is connected to pod i), the core switches contains only terminating /16 prefixes pointing to their destination pods, as shown in Fig. 3.7. This algorithm generates tables whose size is linear in k. No switch in the network contains a table with more than k first-level prefixes or $k/2$ second-level suffixes.

Routing Example: To illustrate network operation using the two-level tables, we give an example for the routing decisions taken for a packet from source 10.0.1.2 to destination 10.2.0.3, as shown in Fig. 3.1. First, the gateway switch of the source host (10.0.1.1) will only match the packet with the /0 first-level prefix, and therefore will forward the packet based on the host ID byte according to the secondary table for that prefix. In that table, the packet matches the 0.0.0.3/8 suffix, which points to port 2 and switch 10.0.2.1. Switch 10.0.2.1 also follows the same steps and forwards on port 3, connected to core switch 10.4.1.1. The core switch matches the packet to a terminating 10.2.0.0/16 prefix, which points to the destination pod 2 on port 2, and switch 10.2.2.1. This switch belongs to the same pod as the destination subnet, and therefore has a terminating prefix, 10.2.0.0/24, which points to the switch responsible for that subnet, 10.2.0.1 on port 0. From there, standard switching techniques deliver the packet to the destination host 10.2.0.3.

Note that for simultaneous communication from 10.0.1.3 to another host 10.2.0.2,

```
foreach pod x in $[0, k-1]$ do
    foreach switch z in $[(k/2), k-1]$ do
        foreach subnet i in $[0, (k/2)-1]$ do
            addPrefix(10.$x$.$z$.1, 10.$x$.$i$.0/24, $i$);
        end
        addPrefix(10.$x$.$z$.1, 0.0.0.0/0, 0);
        foreach host ID i in $[2, (k/2)+1]$ do
            addSuffix(10.$x$.$z$.1, 0.0.0.$i$/8, $(i-2+z)\bmod(k/2)+(k/2)$);
        end
    end
end
```

Figure 3.5: Generating aggregation switch routing tables. Assume function signatures *addPrefix(switch, prefix, port)*, *addSuffix(switch, suffix, port)* and that *addSuffix* adds a second-level suffix to the last-added first-level prefix.

traditional single-path IP routing would follow the same path as the flow above because both destinations are on the same subnet. Unfortunately, this would eliminate all of the fan-out benefits of the fat-tree topology. Instead, our two-level table lookup allows switch 10.0.1.1 to forward the second flow to 10.0.3.1 based on right-handed matching in the two-level table.

3.1.6 Flow Classification

In addition to the two-level routing technique described above, we also consider two optional dynamic routing techniques, as they are currently available in several commercial routers [27,66]. Our goal is to quantify the potential benefits of these techniques but acknowledge that they will incur additional per-packet overhead. Importantly, any maintained state in these schemes is soft and individual switches can fall back to two-level routing in case the state is lost.

As an alternate method of traffic diffusion to the core switches, we perform flow classification with dynamic port-reassignment in pod switches to overcome cases of

```
foreach pod x in $[0, k-1]$ do
    foreach switch z in $[0, (k/2)-1]$ do
        addPrefix(10.$x$.$z$.1, 0.0.0.0/0, 0);
        foreach host ID i in $[2, (k/2)+1]$ do
            addSuffix(10.$x$.$z$.1, 0.0.0.$i$/8, $(i-2+z)\text{mod}(k/2)+(k/2)$);
        end
    end
end
```

Figure 3.6: Generating the routing tables of the edge-level pod switches.

```
foreach j in $[1, (k/2)]$ do
    foreach i in $[1, (k/2)$ do
        foreach destination pod x in $[0, k-1]$ do
            addPrefix(10.$k$.$j$.$i$, 10.$x$.0.0/16, x);
        end
    end
end
```

Figure 3.7: Generating core switch routing tables.

avoidable local congestion (e.g. when two flows compete for the same output port, even though another port that has the same cost to the destination is underused). We define a *flow* as a sequence of packets with the same entries for a subset of fields of the packet headers (typically source and destination IP addresses, destination transport port). In particular, pod switches:

1. Recognize subsequent packets of the same flow, and forward them on the same outgoing port.

2. Periodically reassign a minimal number of flow output ports to minimize any disparity between the aggregate flow capacity of different ports.

Step 1 is a measure against packet reordering, while step 2 aims to ensure fair distribution on flows on upward-pointing ports in the face of dynamically changing flow sizes. § 3.4.2 describes our implementation and flow distribution heuristic of the flow classifier in more detail.

3.1.7 Flow Scheduling

Several studies have indicated that the distribution of transfer times and burst lengths of Internet traffic is long-tailed [33], and characterized by few large long-lived flows (responsible for most of the bandwidth) and many small short-lived ones [44]. We argue that routing large flows plays the most important role in determining the achievable bisection bandwidth of a network and therefore merits special handling. In this alternative approach to flow management, we schedule large flows to minimize overlap with one another. A central scheduler makes this choice, with global knowledge of all active large flows in the network. In this initial design, we only consider the case of a single large flow originating from each host at a time.

Edge Switches

As before, edge switches locally assign a new flow to the least-loaded port initially. However, edge switches additionally detect any outgoing flow whose size grows above a predefined threshold, and periodically send notifications to a central scheduler specifying the source and destination for all active large flows. This represents a request by the edge switch for placement of that flow in an uncontended path.

Note that unlike § 3.1.6, this scheme does not allow edge switches to independently reassign a flow's port, regardless of size. The central scheduler is the only entity with the authority to order a re-assignment.

Central Scheduler

A central scheduler, possibly replicated, tracks all active large flows and tries to assign them non-conflicting paths if possible. The scheduler maintains boolean state for all links in the network signifying their availability to carry large flows.

For inter-pod traffic, recall that there are $(k/2)^2$ possible paths between any given pair of hosts in the network, and each of these paths corresponds to a core switch. When the scheduler receives a notification of a new flow, it linearly searches through the core switches to find one whose corresponding path components do not include a reserved link.[4] Upon finding such a path, the scheduler marks those links as reserved, and notifies the relevant lower- and upper-layer switches in the source pod with the correct outgoing port that corresponds to that flow's chosen path. A similar search is performed for intra-pod large flows; this time for an uncontended path through an upper-layer pod switch. The scheduler garbage collects flows whose last update is older than a given time, clearing their reservations. Note that the edge switches do not block and wait for the scheduler to perform this computation, but initially treat a large flow like any other.

3.1.8 Fault-Tolerance

The redundancy of available paths between any pair of hosts makes the fat-tree topology attractive for fault-tolerance. We propose a simple failure broadcast protocol that allows switches to route around link- or switch-failures one or two hops downstream.

In this scheme, each switch in the network maintains a *Bidirectional Forwarding Detection* session (BFD [73]) with each of its neighbors to determine when a link or neighboring switch fails. From a fault-tolerance perspective, two classes of failure can be weathered: (a) between lower- and upper-layer switches inside a pod, and (b) between core and a upper-level switches. Clearly, the failure of a lower-level switch will cause disconnection for the directly connected hosts; redundant switch elements at the leaves are the only way to tolerate such failures. We describe link failures here because switch failures trigger the same BFD alerts and elicit the same responses.

Lower- to Upper-layer Switches

Link failure between lower- and upper-level switches affects three classes of traffic:

[4]Finding the optimal placement for all large flows requires either knowing the source and destination of all flows ahead of time or path reassignment of existing flows; however, this greedy heuristic gives a good approximation and achieves in simulations 94% efficiency for randomly destined flows among 27k hosts.

1. Outgoing inter- and intra-pod traffic originating from the lower-layer switch. In this case the local flow classifier sets the 'cost' of that link to infinity and does not assign it any new flows, and chooses another available upper-layer switch.

2. Intra-pod traffic using the upper-layer switch as an intermediary. In response, this switch broadcasts a tag notifying all other lower-layer switches in the same pod of the link failure. These switches would check when assigning new flows whether the intended output port corresponds to one of those tags and avoid it if possible.[5]

3. Inter-pod traffic coming into the upper-layer switch. The core switch connected to the upper-layer switch has it as its only access to that pod, therefore the upper-layer switch broadcasts this tag to all its core switches signifying its inability to carry traffic to the lower-layer switch's subnet. These core switches in turn mirror this tag to all upper-layer switches they are connected to in other pods. Finally, the upper-layer switches avoid the single affected core switch when assigning new flows to that subnet.

Upper-layer to Core Switches

A failure of a link from an upper-layer switch to a core affects two classes of traffic:

1. Outgoing inter-pod traffic, in which case the local routing table marks the affected link as unavailable and locally chooses another core switch.

2. Incoming inter-pod traffic. In this case the core switch broadcasts a tag to all other upper-layer switches it is directly connected to signifying its inability to carry traffic to that entire *pod*. As before, these upper-layer switches would avoid that core switch when assigning flows destined to that pod.

Naturally, when failed links and switches come back up and re-establish their BFD sessions, the previous steps are reversed to cancel their effect. In addition, adapting the scheme of § 3.1.7 to accommodate link- and switch-failures is relatively simple. The scheduler marks any link reported to be down as busy or unavailable, thereby disqualifying any path that includes it from consideration, in effect routing large flows around the fault.

[5]We rely on end-to-end mechanisms to restart interrupted flows

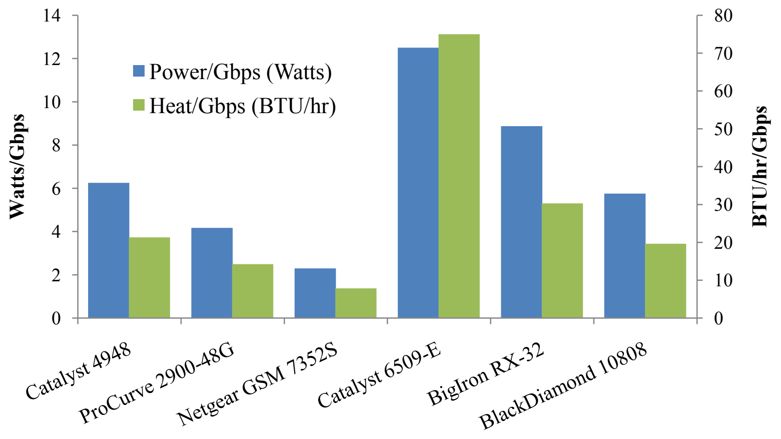


Figure 3.8: Comparison of power and heat dissipation.

3.2 Power and Heat Issues

Besides performance and cost, another major issue that arises in data center design is power consumption. The switches that make up the higher tiers of the interconnect in data centers typically consume thousands of Watts, and in a large-scale data center the power requirements of the interconnect can be hundreds of kilowatts. Almost equally important is the issue of heat dissipation from the switches. Enterprise-grade switches generate considerable amounts of heat and thus require dedicated cooling systems.

In this section we analyze the power requirements and heat dissipation in our architecture and compare it with other typical approaches. We base our analysis on numbers reported in the switch data sheets, though we acknowledge that these reported values are measured in different ways by different vendors and hence may not always reflect system characteristics in deployment.

To compare the power requirement for each class of switch, we normalize the total power consumption and heat dissipation by the switch over the total aggregate bandwidth that a switch can support in Gbps. Fig. 3.8 plots the average over three different switch models. As we can see, 10GigE switches (the last three on the x-axis) consume roughly double the Watts per Gbps and dissipate roughly three times the heat of commodity GigE switches when normalized for bandwidth.

Finally, we also calculated the estimated total power consumption and heat dissipation for an interconnect that can support roughly 27k hosts. For the hierarchical

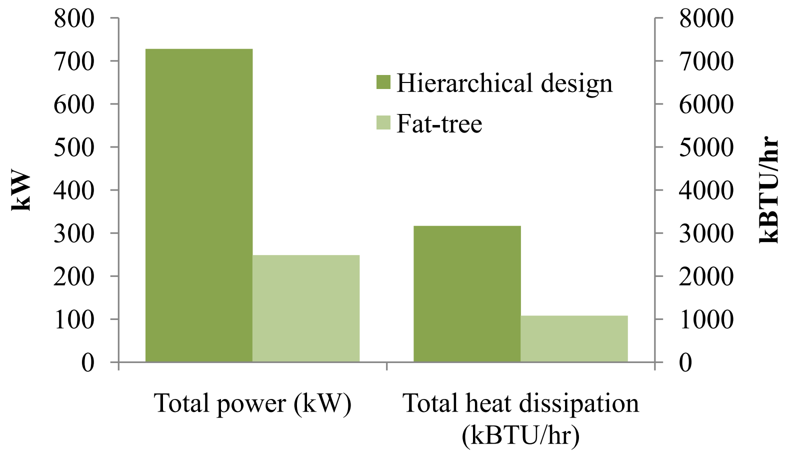


Figure 3.9: Comparison of total power consumption and heat dissipation.

design, we employ 576 ProCurve 2900 edge switches and 54 BigIron RX-32 switches (36 in the aggregation and 18 in the core layer). The fat-tree architecture employs 2,880 Netgear GSM 7252S switches. We are able to use the cheaper NetGear switch because we do not require 10GigE uplinks (present in the ProCurve) in the fat-tree interconnect. Fig. 3.9 shows that while our architecture employs more individual switches, the power consumption and heat dissipation is superior to those incurred by current data center designs, with 56.6% less power consumption and 56.5% less heat dissipation. Of course, the actual power consumption and heat dissipation must be measured in deployment; we refer the reader to a comprehensive and in-depth data center energy analysis in [61].

3.3 Packaging

One drawback of the fat-tree topology for cluster interconnects is the number of cables needed to interconnect all the machines. One trivial benefit of performing aggregation with 10GigE switches is the factor of 10 reduction in the number of cables required to transfer the same amount of bandwidth up the hierarchy. In our proposed fat-tree topology, we do not leverage 10GigE links or switches both because non-commodity pieces would inflate cost and, more importantly, because the fat-tree topology critically depends upon a large fan-out to multiple switches at each layer in the hierarchy to achieve its scaling properties.

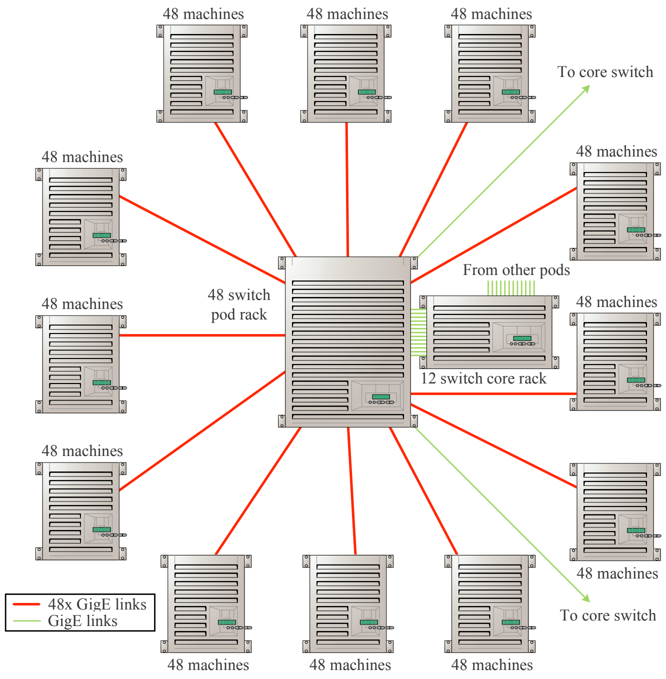


Figure 3.10: Proposed packaging solution. The only external cables are between the pods and the core nodes.

Acknowledging that increased wiring overhead is inherent to the fat-tree topology, in this section we consider some packaging techniques to mitigate this overhead. In sum, our proposed packaging technique eliminates most of the required external wiring and reduces the overall length of required cabling, which in turn simplifies cluster management and reduces total cost. Moreover, this method allows for incremental deployment of the network.

We present our approach in the context of a maximum-capacity 27,648-node cluster leveraging 48-port Ethernet switches as the building block of the fat-tree. This design generalizes to clusters of different sizes. We begin with the design of individual pods that make up the replication unit for the larger cluster, see Fig. 3.10. Each pod consists of 576 machines and 48 individual 48-port GigE switches. For simplicity, we assume each end-host takes up one rack unit (1RU) and that individual racks can accommodate 48 machines. Thus, each pod consists of 12 racks with 48 machines each.

We place the 48 switches that make up the first two layers of the fat-tree in each pod in a centralized rack. However, we assume the ability to package the 48 switches into a single monolithic unit with 1,152 user-facing ports. We call this the *pod switch*. Of these ports, 576 connect directly to the machines in the pod, corresponding to connectivity at the edge. Another 576 ports fan out to one port on each of the 576 switches that make up the core layer in the fat-tree. Note that the 48 switches packaged in this manner actually have 2,304 total ports ($48 * 48$). The other 1,152 ports are wired internally in the pod switch to account for the required interconnect between the edge and aggregation layers of the pod (see Fig. 3.1).

We further spread the 576 required core switches that form the top of the fat-tree across the individual pods. Assuming a total of 48 pods, each will house 12 of the required core switches. Of the 576 cables fanning out from each pod switch to the core, 12 will connect directly to core switches placed nearby in the same pod. The remaining cables would fan out, in sets of 12, to core switches housed in remote pods. Note that the fact that cables move in sets of 12 from pod to pod and in sets of 48 from racks to pod switches opens additional opportunities for appropriate “cable packaging” to reduce wiring complexity.

Finally, minimizing total cable length is another important consideration. To do so, we place racks around the pod switch in two dimensions, as shown in Fig. 3.10 (we do not consider three dimensional data center layouts). Doing so will reduce cable lengths relative to more “horizontal” layouts of individual racks in a pod. Similarly, we lay pods out in a 7×7 grid (with one missing spot) to accommodate all 48 pods. Once again, this grid layout will reduce inter-pod cabling distance to appropriate core switches and will support some standardization of cable lengths and packaging to support inter-pod connectivity.

We also considered an alternate design that did not collect the switches into a central rack. In this approach, two 48-port switches would be distributed to each rack. Hosts would interconnect to the switches in sets of 24. This approach has the advantage of requiring much shorter cables to connect hosts to their first hop switch and for eliminating these cables all together if the racks were appropriately internally packaged. We discarded this approach because we would lose the opportunity to eliminate the 576 cables within each pod that interconnect the edge and aggregation layers. These cables would need to crisscross the 12 racks in each pod, adding significant complexity.

3.4 Implementation

To validate the communication architecture described in this chapter, we built a simple prototype of the forwarding algorithms described in the previous section. We have completed a prototype using NetFPGAs [86]. The NetFPGA contains an IPv4 router implementation that leverages TCAMs. We appropriately modified the routing table lookup routine, as described in § 3.1.4. Our modifications totalled less than 100 lines of additional code and introduced no measurable additional lookup latency, supporting our belief that our proposed modifications can be incorporated into existing switches.

To carry out larger-scale evaluations, we also built a prototype using Click, the focus of our evaluation in the next section. Click [77] is a modular software router architecture that supports implementation of experimental router designs. A Click router is a graph of packet processing modules called *elements* that perform tasks such as routing table lookup or decrementing a packet's TTL. When chained together, Click elements can carry out complex router functionality and protocols in software.

3.4.1 TwoLevelTable

We build a new Click element, *TwoLevelTable*, which implements the idea of a two-level routing table described in § 3.1.3. This element has one input, and two or more outputs. The routing table's contents are initialized using an input file that gives all the prefixes and suffixes. For every packet, the TwoLevelTable element looks up the longest-matching first-level prefix. If that prefix is terminating, it will immediately forward the packet on that prefix's port. Otherwise, it will perform a right-handed longest-matching suffix search on the secondary table and forward on the corresponding port.

This element can replace the central routing table element of the standards-compliant IP router configuration example provided in [77]. We generate an analogous 4-port version of the IP router with the added modification of bandwidth-limiting elements on all ports to emulate link saturation capacity.

3.4.2 FlowClassifier

To provide the flow classification functionality described in § 3.1.6, we describe our implementation of the Click element *FlowClassifier* that has one input and two or more outputs. It performs simple flow classification based on the source and destination IP addresses of the incoming packets, such that subsequent packets with the same source and destination exit the same port (to avoid packet reordering). The element has the added goal of minimizing the difference between the aggregate flow capacity of its highest- and lowest-loaded output ports.

Even if the individual flow sizes are known in advance, this problem is a variant of the NP-hard Bin Packing optimization problem [45]. However, the flow sizes are in fact not known *a priori*, making the problem more difficult. We follow the greedy heuristic outlined in Fig. 3.11. Every few seconds, the heuristic attempts to switch, if needed, the output port of at most three flows to minimize the difference between the aggregate flow capacity of its output ports.

Recall that the FlowClassifier element is an alternative to the two-level table for traffic diffusion. Networks using these elements would employ ordinary routing tables. For example, the routing table of an upper pod switch contains all the subnet prefixes assigned to that pod like before. However, in addition, we add a /0 prefix to match all remaining inter-pod traffic that needs to be evenly spread upwards to the core layer. All packets that match only that prefix are directed to the input of the FlowClassifier. The classifier tries to evenly distribute outgoing inter-pod flows among its outputs according to the described heuristic, and its outputs are connected directly to the core switches. The core switches do not need a classifier, and their routing tables are unchanged.

Note that this solution has soft state that is not needed for correctness, but only used as a performance optimization. This classifier is occasionally disruptive, as a minimal number of flows may be re-arranged periodically, potentially resulting in packet reordering. However, it is also adaptive to dynamically changing flow sizes and 'fair' in the long-term.[6]

[6]Fair in the sense that initial placement decisions are constantly being corrected since all flows' sizes are continually tracked to approximate the optimal distribution of flows to ports.

```
// Call on every incoming packet
IncomingPacket(*packet*)
**begin**
    Hash source and destination IP fields of packet;
    // Have we seen this flow before?
    **if** *seen(hash)* **then**
        Lookup previously assigned port $x$;
        Send packet on port $x$;
    **else**
        Record the new flow $f$;
        Assign $f$ to the least-loaded upward port $x$;
        Send the packet on port $x$;
    **end**
**end**
// Call every $t$ seconds
RearrangeFlows()
**begin**
    **for** $i = 0$ **to** 2 **do**
        Find upward ports $p_{max}$ and $p_{min}$ with the largest and smallest aggregate
        outgoing utilization, respectively;
        Calculate $D$, the utilization difference between$p_{max}$ and $p_{min}$;
        Find the largest flow $f$ assigned to port $p_{max}$ whose size is smaller than $D$;
        **if** *flow f exists* **then**
            Switch the output port of flow $f$ to $p_{min}$;
        **end**
    **end**
**end**
```

Figure 3.11: The flow classifier heuristic. For the experiments in § 3.5, t is 1 second.

3.4.3 FlowScheduler

As described in § 3.1.7, we implemented the element *FlowReporter*, which resides in all edge switches, and detects outgoing flows whose size is larger than a given threshold. It sends regular notifications to the central scheduler about these active large flows.

The *FlowScheduler* element receives notifications regarding active large flows from edge switches and tries to find uncontended paths for them. To this end, it keeps the binary status of all the links in the network, as well as a list of previously placed flows. For any new large flow, the scheduler performs a linear search among all equal-cost paths between the source and destination hosts to find one whose path components are all unreserved. Upon finding such a path, the flow scheduler marks all the component links as reserved and sends notifications regarding this flow's path to the concerned pod switches. We also modify the pod switches to process these port re-assignment messages from the scheduler.

The scheduler maintains two main data structures: a binary array of all unidirectional links in the network (a total of $4 * k * (k/2)^2$ links), and a hashtable of previously placed flows and their assigned paths. The linear search for new flow placement requires on average $2 * (k/2)^2$ memory accesses, making the computational complexity of the scheduler to be $O(k^3)$ for space and $O(k^2)$ for time. A typical value for k (the number of ports per switch) is 48, making both these values manageable, as quantified in § 3.5.3.

3.5 Evaluation

To measure the total bisection bandwidth of our design, we generate a benchmark suite of communication mappings to evaluate the performance of the 4-port fat-tree using the TwoLevelTable switches, the FlowClassifier and the FlowScheduler. We compare these methods to a standard hierarchical tree with a $3.6 : 1$ oversubscription ratio, similar to ones found in current data center designs.

3.5.1 Experiment Description

In the 4-port fat-tree, there are 16 hosts, four pods (each with four switches), and four core switches. Thus, there is a total of 20 switches and 16 end-hosts (for larger clusters, the number of switches will be smaller than the number of hosts). We multiplex these 36 elements onto ten physical machines, interconnected by a 48-port ProCurve 2900 switch with 1 Gigabit Ethernet links. These machines have dual-core Intel Xeon CPUs at 2.33GHz, with 4096KB cache and 4GB of RAM, running Debian GNU/Linux 2.6.17.3. Each pod of switches is hosted on one machine; each pod's hosts are hosted on one machine; and the two remaining machines run two core switches each. Both the switches and the hosts are Click configurations, running in user level. All virtual links between the Click elements in the network are bandwidth-limited to 96Mbit/s to ensure that the configuration is not CPU limited.

For the comparison case of the hierarchical tree network, we have four machines running four hosts each, and four machines each running four pod switches with one additional uplink. The four pod switches are connected to a 4-port core switch running on a dedicated machine. To enforce the 3.6:1 oversubscription on the uplinks from the pod switches to the core switch, these links are bandwidth-limited to 106.67Mbit/s, and all other links are limited to 96Mbit/s.

Each host generates a constant 96Mbit/s of outgoing traffic. We measure the rate of its incoming traffic. The minimum aggregate incoming traffic of all the hosts for all bijective communication mappings is the effective bisection bandwidth of the network.

3.5.2 Benchmark Suite

We generate the communicating pairs according to the following strategies, with the added restriction that any host receives traffic from exactly one host (i.e. the mapping is 1-to-1):

- Random: A host sends to any other host in the network with uniform probability.
- Stride(i): A host with index x will send to the host with index $(x + i)\ mod\ 16$.

- Staggered Prob (*SubnetP*, *PodP*): Where a host will send to another host in its subnet with probability *SubnetP*, and to its pod with probability *PodP*, and to anyone else with probability $1 - SubnetP - PodP$.

- Inter-pod Incoming: Multiple pods send to different hosts in the same pod, and all happen to choose the same core switch. That core switch's link to the destination pod will be oversubscribed. The worst-case *local* oversubscription ratio for this case is $(k-1) : 1$.

- Same-ID Outgoing: Hosts in the same subnet send to different hosts elsewhere in the network such that the destination hosts have the same host ID byte. Static routing techniques force them to take the same outgoing upward port. The worst-case ratio for this case is $(k/2) : 1$. This is the case where the FlowClassifier is expected to improve performance the most.

3.5.3 Results

Table 3.1 shows the results of the above described experiments. These results are averages across 5 runs/permutations of the benchmark tests, over 1 minute each. As expected, for any all-inter-pod communication pattern, the traditional tree saturates the links to the core switch, and thus achieves around 28% of the ideal bandwidth for all hosts in that case. The tree performs significantly better the closer the communicating pairs are to each other.

The two-level table switches achieve approximately 75% of the ideal bisection bandwidth for random communication patterns. This can be explained by the static nature of the tables; two hosts on any given subnet have a 50% chance of sending to hosts with the same host ID, in which case their combined throughput is halved since they are forwarded on the same output port. This makes the expectation of both to be 75%. We expect the performance for the two-level table to improve for random communication with increasing k as there will be less likelihood of multiple flows colliding on a single link with higher k. The inter-pod incoming case for the two-level table gives a 50% bisection bandwidth; however, the same-ID outgoing effect is compounded further by congestion in the core router.

Table 3.1: Aggregate Bandwidth of the network, as a percentage of ideal bisection bandwidth for the Tree, Two-Level Table, Flow Classification, and Flow Scheduling methods. The ideal bisection bandwidth for the fat-tree network is 1.536Gbps.

Traffic Pattern	**Tree**	**Two-Level Table**	**Flow Classification**	**Flow Scheduling**
Random	53.4%	75.0%	76.3%	93.5%
Stride (1)	100.0%	100.0%	100.0%	100.0%
Stride (2)	78.1%	100.0%	100.0%	99.5%
Stride (4)	27.9%	100.0%	100.0%	100.0%
Stride (8)	28.0%	100.0%	100.0%	99.9%
Staggered Prob (1.0, 0.0)	100.0%	100.0%	100.0%	100.0%
Staggered Prob (0.5, 0.3)	83.6%	82.0%	86.2%	93.4%
Staggered Prob (0.2, 0.3)	64.9%	75.6%	80.2%	88.5%
Worst-cases:				
Inter-pod Incoming	28.0%	50.6%	75.1%	99.9%
Same-ID Outgoing	27.8%	38.5%	75.4%	87.4%

Table 3.2: The flow scheduler's time and memory requirements.

k	**Hosts**	**Avg Time/ Req (μs)**	**Link-state Memory**	**Flow-state Memory**
4	16	50.9	64 B	4 KB
16	1,024	55.3	4 KB	205 KB
24	3,456	116.8	14 KB	691 KB
32	8,192	237.6	33 KB	1.64 MB
48	27,648	754.4	111 KB	5.53 MB

Because of its dynamic flow assignment and re-allocation, the flow classifier outperforms both the traditional tree and the two-level table in all cases, with a worst-case bisection bandwidth of approximately 75%. However, it remains imperfect because the type of congestion it avoids is entirely local; it is possible to cause congestion at a core switch because of routing decisions made one or two hops upstream. This type of suboptimal routing occurs because the switches only have local knowledge available.

The FlowScheduler, on the other hand, acts on global knowledge and tries to assign large flows to disjoint paths, thereby achieving 93% of the ideal bisection bandwidth for random communication mappings, and outperforming all other methods in all the benchmark tests. The use of a centralized scheduler with knowledge of all active large flows and the status of all links may be infeasible for large arbitrary networks, but the regularity of the fat-tree topology greatly simplifies the search for uncontended paths.

In a separate test, Table 3.2 shows the time and space requirements for the central scheduler when run on a modestly-provisioned 2.33GHz commodity PC. For varying k, we generated fake placement requests (one per host) to measure the average time to process a placement request, and the total memory required for the maintained link-state and flow-state data structures. For a network of 27k hosts, the scheduler requires a modest 5.6MB of memory and could place a flow in under 0.8ms.

3.6 Acknowledgement

Chapter 3, in part, contains material as it appears in the Proceedings of the ACM Conference on Data Communication (SIGCOMM) 2008. Al-Fares, Mohammad; Loukissas, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Dynamic Flow Scheduling for Data Center Networks

In this chapter, we aim to address the problem of limited forwarding adaptability. That is, the inability of current multipathing techniques to take full advantage of the available bisection bandwidth of large-scale multipath topologies. As we show in chapter 2, this is mainly due to (1) progressive hierarchical oversubscription and (2) poor multipath forwarding. While the previous chapter deals with the first problem, we here tackle the second: the efficient network utilization of multi-rooted tree topologies.

To this end, we present Hedera, a dynamic flow scheduling system for multi-stage data centers topologies. Hedera collects traffic information from the network switches, tries to compute non-conflicting paths for flows, and finally instructs switches to re-route traffic accordingly.[1] This is done with the goal of maximizing aggregate network utilization—bisection bandwidth—and to do so with minimal scheduler overhead or impact on active traffic. By taking a global view of routing and traffic demands, we enable the scheduling system to see bottlenecks that switch-local schedulers cannot.

For both our implementation and large-scale simulations, as we show in § 4.4, our algorithms deliver performance that is within a few percent of optimal—a hypothetical non-blocking switch—for a variety of communication patterns, and deliver in our testbed up to four times more bandwidth than currently deployed ECMP techniques. Hedera delivers these bandwidth improvements with modest control and computation overhead.

[1]In this work, we expand, implement, and evaluate the flow scheduling ideas introduced in chapter 3.

One requirement for our placement algorithms is an accurate view of the demand of individual flows under ideal conditions. Unfortunately, due to constraints at the end-host or elsewhere in the network, measuring current TCP flow bandwidth may have no relation to the bandwidth the flow could achieve with appropriate scheduling. Thus, we also present an efficient algorithm to estimate idealized bandwidth share that each flow would achieve under max-min fair resource allocation, and describe how this algorithm assists in the design of our scheduling techniques.

We first describe the architecture of Hedera in § 4.1, and detail its constituent scheduling and demand estimation algorithms in § 4.2. We next describe our implementation of Hedera on the PortLand testbed [97] in § 4.3, and finally evaluate the system both in simulations and on the testbed for a variety of traffic patterns in § 4.4.

4.1 Architecture

Described at a high-level, Hedera has a control loop of three basic steps. First, it detects large flows at the edge switches. Next, it estimates the natural demand of large flows and uses placement algorithms to compute good paths for them. And finally, these paths are installed on the switches. Our physical implementation was built on top of a fat-tree topology (shown in Fig. 4.5); however, we designed Hedera to support any general multi-rooted tree topology.

4.1.1 Switch Initialization

To take advantage of the path diversity in multi-rooted trees, we must spread outgoing traffic to or from any host as evenly as possible among all the core switches. Therefore, in our system, a packet's path is non-deterministic and chosen on its way up to the core, and is deterministic returning from the core switches to its destination edge switch. Specifically, for multi-rooted topologies, there is exactly one active minimum-cost path from any given core switch to any destination host.

To enforce this determinism on the downward path, we initialize core switches with the prefixes for the IP address ranges of destination pods. A *pod* is any sub-grouping down from the core switches (in our fat-tree testbed, it is a complete bipartite graph of aggregation and edge switches, see Fig. 4.5). Similarly, we initialize aggregation switches with prefixes for downward ports of the edge switches in that pod. Finally, edge switches forward packets directly to their connected hosts.

When a new flow starts, the default switch behavior is to forward it based on a hash on the flow's 10-tuple along one of its equal-cost paths (similar to ECMP). This path is used until the flow grows past a threshold rate, at which point Hedera dynamically calculates an appropriate placement for it. Therefore, all flows are assumed to be small until they grow beyond a threshold, 100Mbps in our implementation (10% of each host's 1GigE link). *Flows* are packet streams with the same 10-tuple of <src MAC, dst MAC, src IP, dst IP, EtherType, IP protocol, TCP src port, dst port, VLAN tag, input port>.

4.1.2 Scheduler Design

A central scheduler, possibly replicated for fail-over and scalability, manipulates the forwarding tables of the edge and aggregation switches dynamically, based on regular updates of current network-wide communication demands. The scheduler aims to assign flows to non-conflicting paths; more specifically, it tries to not place multiple flows on a link that cannot accommodate their combined natural bandwidth demands.

In this model, whenever a flow persists for some time and its bandwidth demand grows beyond a defined limit, we assign it a path using one of the scheduling algorithms described in § 4.2. Depending on this chosen path, the scheduler inserts flow entries into the edge and aggregation switches of the source pod for that flow; these entries redirect the flow on its newly chosen path. The flow entries expire after a timeout once the flow terminates. Note that the state maintained by the scheduler is only soft-state and does not have to be synchronized with any replicas to handle failures. Scheduler state is not required for correctness (connectivity); rather it aids as a performance optimization.

Of course, the choice of the specific scheduling algorithm is open. In this chapter, we compare two algorithms, Global First Fit and Simulated Annealing, to ECMP. Both algorithms search for flow-to-core mappings with the objective of increasing the aggregate bisection bandwidth for current communication patterns, supplementing default ECMP forwarding for large flows.

4.2 Estimation and Scheduling

Finding flow routes in a general network while not exceeding the capacity of any link is called the Multi-Commodity Flow problem, which is NP-complete for integer flows [38]. And while simultaneous flow routing is solvable in polynomial time for 3-stage Clos networks, no polynomial time algorithm is known for 5-stage Clos networks (i.e. 3-tier fat-trees) [60]. Since we do not aim to optimize Hedera for a specific topology, this chapter presents practical heuristics that can be applied to a range of realistic data center topologies.

4.2.1 Host- vs. Network-Limited Flows

A flow can be classified into two categories: network-limited (e.g. data transfer from RAM) and host-limited (e.g. limited by host disk access, processing, etc.). A network-limited flow will use all bandwidth available to it along its assigned path. Such a flow is limited by congestion in the network, not at the host NIC. A host-limited flow can theoretically achieve a maximum throughput limited by the "slower" of the source and destination hosts. In the case of non-optimal scheduling, a network-limited flow might achieve a bandwidth less than the maximum possible bandwidth available from the underlying topology. In this project, we focus on network-limited flows, since host-limited flows are a symptom of intra-machine bottlenecks, which are beyond the scope of this project.

4.2.2 Demand Estimation

A TCP flow's current sending rate says little about its natural bandwidth demand in an ideal non-blocking network (§ 2.2.3). Therefore, to make intelligent flow placement decisions, we need to know the flows' max-min fair bandwidth allocation as if they are limited only by the sender or receiver NIC. When network limited, a sender will try to distribute its available bandwidth fairly among all its outgoing flows. TCP's AIMD behavior combined with fair queueing in the network tries to achieve max-min fairness. Note that when there are multiple flows from a host *A* to another host *B*, each of the flows

$$\begin{bmatrix} & (\frac{1}{3})_1 & (\frac{1}{3})_1 & (\frac{1}{3})_1 \\ (\frac{1}{3})_2 & & (\frac{1}{3})_1 & 0_0 \\ (\frac{1}{2})_1 & 0_0 & & (\frac{1}{2})_1 \\ 0_0 & (\frac{1}{2})_2 & 0_0 & \end{bmatrix} \Rightarrow \begin{bmatrix} & [\frac{1}{3}]_1 & (\frac{1}{3})_1 & (\frac{1}{3})_1 \\ [\frac{1}{3}]_2 & & (\frac{1}{3})_1 & 0_0 \\ [\frac{1}{3}]_1 & 0_0 & & (\frac{1}{2})_1 \\ 0_0 & [\frac{1}{3}]_2 & 0_0 & \end{bmatrix} \Rightarrow$$

$$\Rightarrow \begin{bmatrix} & [\frac{1}{3}]_1 & (\frac{1}{3})_1 & (\frac{1}{3})_1 \\ [\frac{1}{3}]_2 & & (\frac{1}{3})_1 & 0_0 \\ [\frac{1}{3}]_1 & 0_0 & & (\frac{2}{3})_1 \\ 0_0 & [\frac{1}{3}]_2 & 0_0 & \end{bmatrix} \Rightarrow \begin{bmatrix} & [\frac{1}{3}]_1 & (\frac{1}{3})_1 & [\frac{1}{3}]_1 \\ [\frac{1}{3}]_2 & & (\frac{1}{3})_1 & 0_0 \\ [\frac{1}{3}]_1 & 0_0 & & [\frac{2}{3}]_1 \\ 0_0 & [\frac{1}{3}]_2 & 0_0 & \end{bmatrix}$$

Figure 4.1: Demand Estimation example. Matrix elements denote demand per flow as a fraction of the NIC bandwidth. Subscripts denote the number of flows from source (rows) to destination (columns). Entries in parentheses have yet to converge, and grayed entries have converged.

will have the same steady state demand. We now describe how to find TCP demands in a hypothetical equilibrium state.

The input to the demand estimator is the set F of source and destination pairs for all active large flows. The estimator maintains an $N \times N$ matrix M; N is the number of hosts. The element in the i^{th} row, j^{th} column contains 3 values: (1) the number of flows from host i to host j, (2) the estimated demand of each of the flows from host i to host j, and (3) a "converged" flag that marks flows whose demands have converged.

The demand estimator performs repeated iterations of increasing the flow capacities from the sources and decreasing exceeded capacity at the receivers until the flow capacities converge; Fig. 4.4 presents the pseudocode. Note that in each iteration of decreasing flow capacities at the receivers, one or more flows converge until eventually all flows converge to the natural demands. The estimation time complexity is $O(|F|)$.

Fig. 4.1 illustrates the process of estimating flow demands with a simple example. Consider 4 hosts (H_0, H_1, H_2 and H_3) connected by a non-blocking topology. Suppose H_0 sends 1 flow each to H_1, H_2 and H_3; H_1 sends 2 flows to H_0 and 1 flow to H_2; H_2 sends 1 flow each to H_0 and H_3; and H_3 sends 2 flows to H_1. The figure shows the iterations of the demand estimator. The matrices indicate the flow demands during successive stages

```
GLOBAL-FIRST-FIT($f$: flow)
1   if $f$.assigned then
2       return old path assignment for $f$
3   foreach $p \in P_{src \to dst}$ do
4       if $p$.used + $f$.rate < $p$.capacity then
5           $p$.used ← $p$.used + $f$.rate
6           return $p$
7   $h$ =HASH($f$)
8   return $p = P_{src \to dst}(h)$
```

Figure 4.2: Pseudocode for Global First Fit. Called for each flow in the system.

of the algorithm starting with an increase in flow capacity from the sender followed by a decrease in flow capacity at the receiver and so on. The last matrix indicates the final estimated natural demands of the flows.

For real communication patterns, the demand matrix for currently active flows is a sparse matrix since most hosts will be communicating with a small subset of remote hosts at a time. The demand estimator is also largely parallelizable, facilitating scalability. In fact, our implementation uses both parallelism and sparse matrix data structures to improve the performance and memory footprint of the algorithm.

4.2.3 Global First Fit

In a multi-rooted tree topology, there are several possible equal-cost paths between any pair of source and destination hosts. When a new large flow is detected, (e.g. 10% of the host's link capacity), the scheduler linearly searches all possible paths to find one whose link components can all accommodate that flow. If such a path is found, then that flow is "placed" on that path: First, a capacity reservation is made for that flow on the links corresponding to the path. Second, the scheduler creates forwarding entries in the corresponding edge and aggregation switches. To do so, the scheduler maintains the reserved capacity on every link in the network and uses that to determine which paths are available to carry new flows. Reservations are cleared when flows expire.

```
SIMULATED-ANNEALING($n$ : iteration count)
1   $s \leftarrow$ INIT-STATE()
2   $e \leftarrow E(s)$
3   $s_B \leftarrow s, e_B \leftarrow e$
4   $T_0 \leftarrow n$
5   for $T \leftarrow T_0 \ldots 0$ do
6       $s_N \leftarrow$ NEIGHBOR($s$)
7       $e_N \leftarrow E(s_N)$
8       if $e_N < e_B$ then
9           $s_B \leftarrow s_N, e_B \leftarrow e_N$
10      if $P(e, e_N, T) >$ RAND() then
11          $s \leftarrow s_N, e \leftarrow e_N$
12  return $s_B$
```

Figure 4.3: Pseudocode for Simulated Annealing. s denotes the current state with energy $E(s) = e$. e_B denotes the best energy seen so far in state s_B. T denotes the temperature. e_N is the energy of a neighboring state s_N.

Note that this corresponds to a first fit algorithm; a flow is greedily assigned the *first* path that can accommodate it. When the network is lightly loaded, finding such a path among the many possible paths is likely to be easy; however, as the network load increases and links become saturated, this choice becomes more difficult. Global First Fit does not guarantee that all flows will be accommodated, but this algorithm performs relatively well in practice as shown in § 4.4. We show the pseudocode for Global First Fit in Fig. 4.2.

4.2.4 Simulated Annealing

Next we describe the Simulated Annealing scheduler, which performs a probabilistic search to efficiently compute paths for flows. The key insight of our approach is to assign a single core switch for each destination host rather than a core switch for each flow. This reduces the search space significantly. Simulated Annealing forwards all flows destined to a particular host A through the designated core switch for host A.

The input to the algorithm is the set of all large flows to be placed, and their flow demands as estimated by the demand estimator. Simulated Annealing searches through a solution state space to find a near-optimal solution (Fig. 4.3). A function E defines the energy in the current state. In each iteration, we move to a neighboring state with a certain acceptance probability P, depending on the energies in the current and neighboring states and the current temperature T. The temperature is decreased with each iteration of the Simulated Annealing algorithm and we stop iterating when the temperature is zero. Allowing the solution to move to a higher energy state allows us to avoid local minima.

1. State s: A set of mappings from destination hosts to core switches. Each host in a pod is assigned a particular core switch that it receives traffic from.

2. Energy function E: The total exceeded capacity over all the links in the current state. Every state assigns a unique path to every flow. We use that information to find the links for which the total capacity is exceeded and sum up exceeded demands over these links.

3. Temperature T: The remaining number of iterations before termination.

4. Acceptance probability P for transition from state s to neighbor state s_n, with energies E and E_n.

$$P(E_n, E, T) = \begin{cases} 1 & \text{if } E_n < E \\ e^{c(E-E_n)/T} & \text{if } E_n \geq E \end{cases}$$

   where c is a parameter that can be varied. We empirically determined that $c = 0.5 \times T_0$ gives best results for a 16 host cluster and $c = 1000 \times T_0$ is best for larger data centers.

5. Neighbor generator function NEIGHBOR(): Swaps the assigned core switches for a pair of hosts in any of the pods in the current state s.

While simulated annealing is a known technique, our contribution lies in an optimization to significantly reduce the search space and the choice of appropriate energy and neighbor selection functions to ensure rapid convergence to a near optimal schedule. A straightforward approach is to assign a core for each flow individually and perform simulated annealing. However this results in a huge search space limiting the effectiveness of simulated annealing. The diameter of the search space (maximum number of neighbor hops between any two states) with this approach is equal to the number of flows in the system. Our technique of assigning core switches to destination hosts reduces the diameter of the search space to the minimum of the number of flows and the number of hosts in the data center. This heuristic reduces the search space significantly: in a 27k host data center with 27k large flows, the search space size is reduced by a factor of 10^{12000}. Simulated Annealing performs better when the size of the search space and its diameter are reduced [41]. With the straightforward approach, the runtime of the algorithm is proportional to the number of flows and the number of iterations while our technique's runtime depends only on the number of iterations.

We implemented both the baseline and optimized version of Simulated Annealing. Our simulations show that for randomized communication patterns in a 8,192 host data center with 16k flows, our techniques deliver a 20% improvement in bisection bandwidth and a 10-fold reduction in computation time compared to the baseline. These gains increase both with the size of the data center as well as the number of flows.

Initial State: Each pod has some fixed downlink capacity from the core switches which is useful only for traffic destined to that pod. So an important insight here is that we should distribute the core switches among the hosts in a single pod. For a fat-tree, the number of hosts in a pod is equal to the number of core switches, suggesting a one-to-one mapping. We restrict our solution search space to such assignments, i.e. we assign cores not to individual flows, but to destination hosts. Note that this choice of initial state is only used when the Simulated Annealing scheduler is run for the first time. We use an optimization to handle the dynamics of the system which reduces the importance of this initial state over time.

Neighbor Generator: A well-crafted neighbor generator function intrinsically avoids deep local minima. Complying with the idea of restricting the solution search space to mappings with near-uniform mapping of hosts in a pod to core switches, our implementation employs three different neighbor generator functions: (1) swap the assigned core

```
Estimate-Demands()
    for all $i, j$
        $M_{i,j} \leftarrow 0$
    do
        foreach $h \in H$ do Est-Src($h$)
        foreach $h \in H$ do Est-Dst($h$)
    while some $M_{i,j}$.demand changed
    return $M$

Est-Src(src: host)
    $d_F \leftarrow 0$
    $n_U \leftarrow 0$
    foreach $f \in \langle \text{src} \rightarrow \text{dst} \rangle$ do
        if $f$.converged then
            $d_F \leftarrow d_F + f$.demand
        else
            $n_U \leftarrow n_U + 1$
    $e_S \leftarrow \frac{1.0 - d_F}{n_U}$
    foreach ($f \in \langle \text{src} \rightarrow \text{dst} \rangle$ and
            not $f$.converged) do
        $M_{f.\text{src}, f.\text{dst}}$.demand $\leftarrow e_S$
```

```
Est-Dst(dst: host)
    $d_T, d_S, n_R \leftarrow 0$
    foreach $f \in \langle \text{src} \rightarrow \text{dst} \rangle$
        $f$.rl $\leftarrow$ true
        $d_T \leftarrow d_T + f$.demand
        $n_R \leftarrow n_R + 1$
    if $d_T \leq 1.0$ then
        return
    $e_S \leftarrow \frac{1.0}{n_R}$
    do
        $n_R \leftarrow 0$
        foreach $f \in \langle \text{src} \rightarrow \text{dst} \rangle$ and $f$.rl do
            if $f$.demand $< e_S$ then
                $d_S \leftarrow d_S + f$.demand
                $f$.rl $\leftarrow$ false
            else
                $n_R \leftarrow n_R + 1$
        $e_S \leftarrow \frac{1.0 - d_S}{n_R}$
    while some $f$.rl was set to false
    foreach $f \in \langle \text{src} \rightarrow \text{dst} \rangle$ and $f$.rl do
        $M_{f.\text{src}, f.\text{dst}}$.demand $\leftarrow e_S$
        $M_{f.\text{src}, f.\text{dst}}$.converged $\leftarrow$ true
```

Identifier	Description
M	The demand matrix
H	The set of hosts
d_F	"Converged" demand
n_U	The number of unconverged flows
e_S	The computed equal share rate
$\langle \text{src} \rightarrow \text{dst} \rangle$	The set of flows from src to some dst
d_T	The total demand
d_S	Sender limited demand
f.rl	A flag for a receiver limited flow
n_R	The number of receiver limited flows

Figure 4.4: Demand estimator pseudocode for TCP flows, with identifier legend.

switches for any two randomly chosen hosts in a randomly chosen pod, (2) swap the assigned core switches for any two randomly chosen hosts in a randomly chosen edge switch, (3) randomly choose an edge or aggregation switch with equal probability and swap the assigned core switches for a random pair of hosts that use the chosen edge or aggregation switch to reach their currently assigned core switches. Our neighbor generator function randomly chooses between the 3 described techniques with equal probability at runtime for each iteration. Using multiple neighbor generator functions helps us avoid deep local minima in the search spaces of individual neighbor generator functions.

Calculation of Energy Function: The energy function for a neighbor can be calculated incrementally based on the energy in the current state and the cores that were swapped in the neighbor. We need not recalculate exceeded capacities for all links. Swapping assigned cores for a pair of hosts only affects those flows destined to those two hosts. So we need to recalculate the difference in the energy function only for those specific links involved and update the value of the energy based on the energy in the current state. Thus, the time to calculate the energy only depends on the number of large flows destined to the two affected hosts.

Dynamically-Changing Flows: With dynamically-changing flow patterns, in every scheduling phase, a few flows would be newly classified as large flows and a few older ones would have completed their transfers. We have implemented an optimization where we set the initial state to the best state from the previous scheduling phase. This allows the route-placement of existing, continuing flows to be disrupted as little as possible if their current paths can still support their bandwidth requirements. Further, the initial state that is used when the Simulated Annealing scheduler first starts up becomes less relevant over time due to this optimization.

Search Space: The key characteristic of Simulated Annealing is assigning unique core switches based on destination hosts in a pod, crucial to reducing the size of the search space. However, there are communication patterns where an optimal solution necessarily requires a single destination host to receive incoming traffic through multiple core switches. While we omit the details for brevity, we find that, at least for the fat-tree topol-

Table 4.1: Complexity of Global First Fit and Simulated Annealing. k is the number of switch ports, $|F|$ is the total number of large flows, and f_{avg} is the average number of large flows to a host. k^3 is due to link-state structures, and $|F|$ is due to the flows' state.

Algorithm	Time	Space
Global First-Fit	$O((k/2)^2)$	$O(k^3 + |F|)$
Simulated Annealing	$O(f_{avg})$	$O(k^3 + |F|)$

ogy, all communication patterns can be handled if: i) the maximum number of large flows to or from a host is at most $k/2$, where k is the number of ports in the network switches, or ii) the minimum threshold of each large flow is set to $2/k$ of the link capacity. Given that in practice data centers are likely to be built from relatively high-radix switches, e.g., $k \geq 32$, our search space optimization is unlikely to eliminate the potential for locating optimal flow assignments in practice.

4.2.5 Comparison of Placement Algorithms

With Global First Fit, a large flow can be re-routed immediately upon detection and is essentially pinned to its reserved links. Whereas Simulated Annealing waits for the next scheduling tick, uses previously computed flow placements to optimize the current placement, and delivers even better network utilization on average due to its probabilistic search.

We chose the Global First Fit and Simulated Annealing algorithms for their simplicity; we take the view that more complex algorithms can hinder the scalability and efficiency of the scheduler while gaining only incremental bandwidth returns. We believe that they strike the right balance of computational complexity and delivered performance gains. Table 4.1 gives the time and space complexities of both algorithms. Note that the time complexity of Global First Fit is independent of $|F|$, the number of large flows in the network, and that the time complexity of Simulated Annealing is independent of k.

More to the point, the simplicity of our algorithms makes them both well-suited for implementation in hardware, such as in an FPGA, as they consist mainly of simple arithmetic. Such an implementation would substantially reduce the communication overhead of crossing the network stack of a standalone scheduler machine.

Overall, while Simulated Annealing is more conceptually involved, we show in § 4.4 that it almost always outperforms Global First Fit, and delivers close to the optimal bisection bandwidth both for our testbed and in larger simulations. We believe the additional conceptual complexity of Simulated Annealing is justified by the bandwidth gains and tremendous investment in the network infrastructure of modern data centers.

4.2.6 Fault Tolerance

Any scheduler must account for switch and link failures in performing flow assignments. While we omit the details for brevity, our Hedera implementation augments the PortLand routing and fault tolerance protocols [97]. Hence, the Hedera scheduler is aware of failures using the standard PortLand mechanisms and can re-route flows mapped to failed components. Specifically, existing flows assigned to failed links or switches would be re-mapped, and any paths going through them would be skipped during path search on subsequent scheduling rounds.

We also note that all scheduler-assigned flow mappings are soft-state and are optimizations for maximizing bisection bandwidth. They are not needed for connectivity. Therefore, in the case of scheduler machine failure, hotswapping a backup scheduler would be possible after flow entries time out, and would not require a strong consistency model. Furthermore, barring scheduler backups, switches would simply revert to their default ECMP forwarding in the worst-case.

4.3 Implementation

To test our scheduling techniques on a real physical multi-rooted network, we built as an example the fat-tree network described abstractly in prior work [2]. In addition, to understand how our algorithms scale with network size, we implemented a simulator to model the behavior of large networks with many flows under the control of a scheduling algorithm.

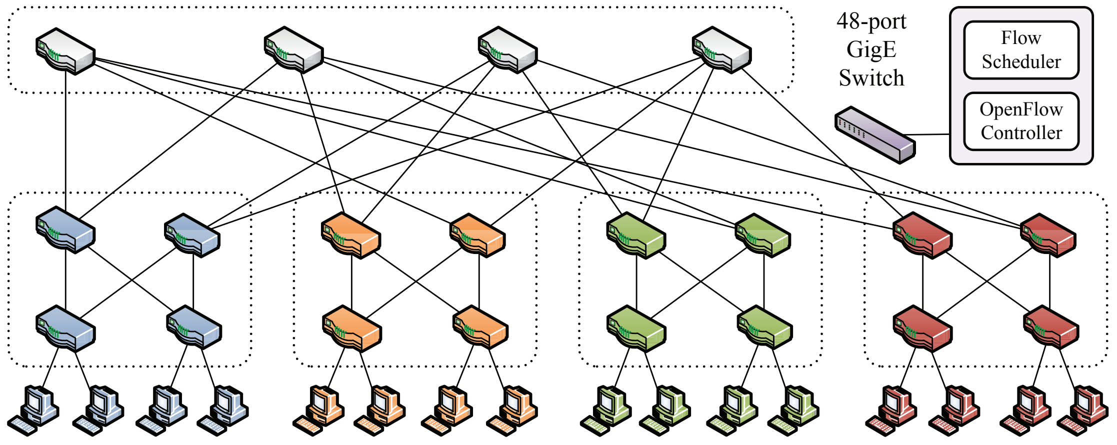


Figure 4.5: Hedera system architecture. The interconnect shows the data-plane network, with GigE links throughout.

4.3.1 Topology

For the rest of the chapter, we adopt the following terminology: for a fat-tree network built from k-port switches, there are k *pods*, each consisting of two layers: lower pod switches (*edge* switches), and the upper pod switches (*aggregation* switches). Each edge switch manages $(k/2)$ hosts. The k pods are interconnected by $(k/2)^2$ *core* switches.

One of the main advantages of this topology is the high degree of available path diversity; between any given source and destination host pair, there are $(k/2)^2$ equal-cost paths, each corresponding to a core switch. Note, however, that these paths are not link-disjoint. To take advantage of this path diversity (to maximize the achievable bisection bandwidth), we must assign flows non-conflicting paths. A key requirement of our work is to perform such scheduling with no modifications to end-host network stacks or operating systems. Our testbed consists of 16 hosts interconnected using a fat-tree of twenty 4-port switches, as shown in Fig. 4.5.

We deploy a parallel control plane connecting all switches to a 48-port non-blocking GigE switch. We emphasize that this control network is not required for the Hedera architecture, but is used in our testbed as a debugging and comparison tool. This network transports only traffic monitoring and management messages to and from the switches; however, these messages could also be transmitted using the data plane. Naturally, for larger networks of thousands of hosts, a control network could be organized as a traditional tree, since control traffic should be only a small fraction of the data traffic. In our deployment, the flow scheduler runs on a separate machine connected to the 48-port switch.

4.3.2 Hardware Description

The switches in the testbed are 1U dual-core 3.2 GHz Intel Xeon machines, with 3GB RAM, and NetFPGA 4-port GigE PCI card switches [86]. The 16 hosts are 1U quad-core 2.13 GHz Intel Xeon machines with 3GB of RAM. These hosts have two GigE ports, the first connected to the control network for testing and debugging, and the other to its NetFPGA edge switch. The control network is organized as a simple star topology. The central switch is a Quanta LB4G 48-port GigE switch. The scheduler machine has a dual-core 2.4 GHz Intel Pentium CPU and 2GB of RAM.

4.3.3 OpenFlow Control

The switches in the tree all run OpenFlow [91], which allows access to the forwarding tables for all switches. OpenFlow implementations have been ported to a variety of commercial switches, including those from Juniper, HP, and Cisco. OpenFlow switches match incoming packets to flow entries that specify a particular action such as duplication, forwarding on a specific port, dropping, and broadcast. The NetFPGA OpenFlow switches have 2 hardware tables: a 32-entry TCAM (that accepts variable-length prefixes) and a 32K entry SRAM that only accepts flow entries with fully qualified 10-tuples.

When OpenFlow switches start, they attempt to open a secure channel to a central controller. The controller can query, insert, modify flow entries, or perform a host of other actions. The switches maintain statistics per flow and per port, such as total byte counts, and flow durations. The default behavior of the switch is as follows: if an incoming packet does not match any of the flow entries in the TCAM or SRAM table, the switch inserts a new flow entry with the appropriate output port (based on ECMP) which allows any subsequent packets to be directly forwarded at line rate in hardware. Once a flow grows beyond the specified threshold, the Hedera scheduler may modify the flow entry for that flow to redirect it along a newly chosen path.

4.3.4 Scheduling Frequency

Our scheduler implementation polls the edge switches for flow statistics (to detect large flows), and performs demand estimation and scheduling once every five seconds. This period is due entirely to a register read-rate limitation of the OpenFlow NetFPGA implementation. However, our scalability measurements in § 4.4 show that a modestly-provisioned machine can schedule tens of thousands of flows in a few milliseconds, and that even at the 5s polling rate, Hedera significantly outperforms the bisection bandwidth of current ECMP methods. In general, we believe that sub-second and potentially sub-100ms scheduling intervals should be possible using straightforward techniques.

4.3.5 Simulator

Since our physical testbed is restricted to 16 hosts, we also developed a simulator that coarsely models the behavior of a network of TCP flows. The simulator accounts for flow arrivals and departures to show the scalability of our system for larger networks with dynamic communication patterns. We examine our different scheduling algorithms using the flow simulator for networks with as many as 8,192 hosts. Existing packet-level simulators, such as *ns-2*, are not suitable for this purpose: e.g. a simulation with 8,192 hosts each sending at 1Gbps would have to process 2.5×10^{11} packets for a 60 second run. If a per-packet simulator were used to model the transmission of 1 million packets per second using TCP, it would take 71 hours to simulate just that one test case.

Our simulator models the data center topology as a network graph with directed edges. Each edge has a fixed capacity. The simulator accepts as input a communication pattern among hosts and uses it, along with a specification of average flow sizes and arrival rates, to generate simulated traffic. The simulator generates new flows with an exponentially distributed length, with start times based on a Poisson arrival process with a given mean. Destinations are based upon the suite in § 4.4.

The simulation proceeds in discrete time ticks. At each tick, the simulation updates the rates of all flows in the network, generates new flows if needed. Periodically it also calls the scheduler to assign (new) routes to flows. When calling the Simulated

Annealing and Global First Fit schedulers, the simulator first calls the demand estimator and passes along its results.

When updating flow rates, the simulator models TCP slow start and AIMD, but without performing per-packet computations. Each tick, the simulator shuffles the order of flows and computes the expected rate increase for each flow, constrained by available bandwidth on the flow's path. If a flow is in slow start, its rate is doubled. If it is in congestion avoidance, its rate is additively increased (using an additive increase factor of 15MB/s to simulate a network with an RTT of 100μs). If the flow's path is saturated, the flow's rate is halved and bandwidth is freed along the path. Each tick, we also compute the number of bytes sent by the flow and purge flows that have completed sending all their bytes.

Since our simulator does not model individual packets, it does not capture the variations in performance of different packet sizes. Another consequence of this decision is that our simulation cannot capture inter-flow dynamics or buffer behavior. As a result, it is likely that TCP Reno/New Reno would perform somewhat *worse* than predicted by our simulator. In addition, we model TCP flows as unidirectional although real TCP flows involve ACKs in the reverse direction; however, for 1500B Ethernet frames and delayed ACKs, the bandwidth consumed by ACKs is about 2%. We feel these trade-offs are necessary to study networks of the scale described in this chapter.

We ran each simulation for the equivalent of 60 seconds and measured the average bisection bandwidth during the middle 40 seconds. Since the simulator does not capture inter-flow dynamics and traffic burstiness our results are optimistic (simulator bandwidth exceeds testbed measurements) for ECMP based flow placement because resulting hash collisions would sometimes cause an entire window of data to be lost, resulting in a coarse-grained timeout on the testbed (see § 4.4). For the control network we observed that the performance in the simulator more closely matched the performance on the testbed. Similarly, for Global First Fit and Simulated Annealing, which try to optimize for minimum contention, we observed that the performance from the simulator and testbed matched very well. Across all the results, the simulator indicated better performance than the testbed when there is contention between flows.

4.4 Evaluation

This section describes our evaluation of Hedera using our testbed and simulator. The goal of these tests is to determine the aggregate achieved bisection bandwidth with various traffic patterns.

4.4.1 Benchmark Communication Suite

In the absence of commercial data center network traces, for both the testbed and the simulator evaluation, we first create a group of communication patterns similar to chapter 3 according to the following styles:

1. Stride(i): A host with index x sends to the host with index $(x+i) mod(num\_hosts)$.
2. Staggered Prob *(EdgeP, PodP)*: A host sends to another host in the same edge switch with probability *EdgeP*, and to its same pod with probability *PodP*, and to the rest of the network with probability *1-EdgeP - PodP*.
3. Random: A host sends to any other host in the network with uniform probability. We include bijective mappings and ones where hotspots are present.

We consider these mappings for networks of different sizes: 16 hosts, 1,024 hosts, and 8,192 hosts, corresponding to $k = \{4, 16, 32\}$.

4.4.2 Testbed Benchmark Results

We ran benchmark tests as follows: 16 hosts open socket sinks for incoming traffic and measure the incoming bandwidth constantly. The hosts in succession then start their flows according to the sizes and destinations as described above. Each experiment lasts for 60 seconds and uses TCP flows; we observed the average bisection bandwidth for the middle 40 seconds.

We compare the performance of the scheduler on the fat-tree network to that of the same experiments on the control network. The control network connects all 16 hosts using a non-blocking 48-port gigabit Ethernet switch and represents an ideal network.

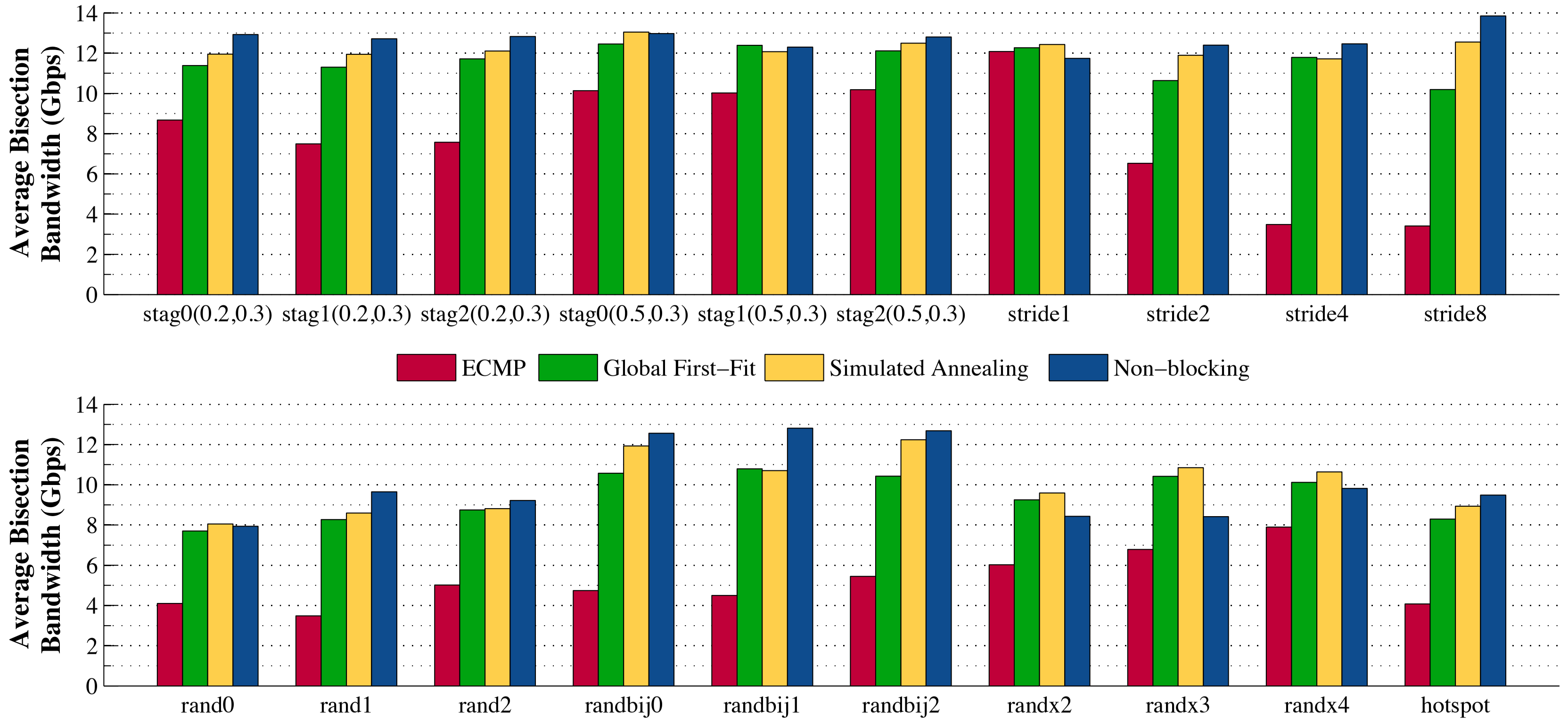


Figure 4.6: Physical testbed benchmark suite results for the three routing methods vs. a non-blocking switch. Figures indicate network bisection bandwidth achieved for staggered, stride, and randomized communication patterns.

In addition, we include a static hash-based ECMP scheme, where the forwarding path is determined by a hash of the destination host IP address.

Fig. 4.6 shows the bisection bandwidth for a variety of randomized, staggered, stride and hotspot communication patterns; our experiments saturate the links using TCP. In virtually all the communication patterns explored, Global First Fit and Simulated Annealing significantly outperform static hashing (ECMP), and achieve near the optimal bisection bandwidth of the network (15.4Gbps goodput). Naturally, the performance of these schemes improves as the level of communication locality increases, as demonstrated by the staggered probability figures. Note that for stride patterns (common to HPC applications), the heuristics consistently compute the correct flow-to-core mappings to efficiently utilize the fat-tree network, whereas the performance of static hash quickly deteriorates as the stride length increases. Furthermore, for certain patterns, these heuristics also marginally outperform the commercial 48-port switch used for our control network. We suspect this is due to different buffers/algorithms of the NetFPGAs vs. the Quanta switch.

Upon closer examination of the performance using packet captures from the testbed, we found that when there was contention between flows, an entire TCP window of packets was often lost. So the TCP connection was idle until the retransmission timer fired (RTO_{min} = 200ms). ECMP hash based flow placement experienced over 5 times the number of retransmission timeouts as the other schemes. This explains the overoptimistic performance of ECMP in the simulator as explained in § 4.3 since our simulator does not model retransmission timeouts and individual packet losses.

4.4.3 Data Shuffle

We also performed an all-to-all in-memory data shuffle in our testbed. A data shuffle is an expensive but necessary operation for many MapReduce/Hadoop operations in which every host transfers a large amount of data to every other host participating in the shuffle. In this experiment, each host sequentially transfers 500MB to every other host using TCP (a 120GB shuffle).

The shuffle results in Table 4.2 show that centralized flow scheduling performs considerably better (39% better bisection bandwidth) than static ECMP hash-based rout-

Table 4.2: A 120GB shuffle for the placement heuristics in our testbed. Shown is total shuffle time, average host-completion time, average bisection bandwidth and average host goodput.

Metric	ECMP	GFF	SA	Control
Total Shuffle Time (s)	438.44	335.50	335.96	306.37
Average Host Completion (s)	358.14	258.70	261.96	226.56
Bisection Bandwidth (Gbps)	2.81	3.89	3.84	4.44
Host Goodput (MB/s)	20.94	28.99	28.63	33.10

ing. Comparing this to the data shuffle performed in VL2 [51], which involved all hosts making *simultaneous* transfers to all other hosts (versus the sequential transfers in our work), we see that static hashing performs better when the number of flows is significantly larger than the number of paths; intuitively a hash collision is less likely to introduce significant degradation when any imbalance is averaged over a large number of flows. For this reason, in addition to the delay of the Hedera observation/route-computation control loop, we believe that traffic workloads characterized by many small, short RPC-like flows would have limited benefit from dynamic scheduling, and Hedera's default ECMP forwarding performs load-balancing efficiently in this case. Hence, by thresholding our scheduler to only operate on larger flows, Hedera performs well for both types of communication patterns.

4.4.4 Simulation Results

OMNeT++ Simulations

Using the OMNeT++ network simulator [100], we show in Table 4.3 the achieved bisection bandwidth and the mean time for each host to transfer a 1MB file simultaneously using TCP with the same topology and benchmark communication suite as our testbed and compare that to a non-blocking switch (labeled *control*). This confirms that oversubscription due to inefficient load-balancing can greatly delay file transfers and overall job completion times.

Table 4.3: Network bisection bandwidth and mean completion time for simultaneous 1MB transfers in OMNeT++.

	ECMP		**GFF**		**SA**		**Control**	
Traffic Pattern	BW (GB/s)	Time (ms)	BW (GB/s)	Time (ms)	BW (GB/s)	Time (ms)	BW (GB/s)	Time (ms)
Stride(1)	1.94	10.90	1.94	10.90	1.94	10.90	1.93	11.00
Stride(2)	0.86	24.70	1.93	11.20	1.93	11.20	1.93	11.00
Stride(4)	0.48	41.90	1.92	11.30	1.92	11.30	1.93	11.00
Stride(8)	0.48	41.90	1.92	11.30	1.92	11.30	1.93	11.00
Stag(1, 0)	1.94	10.90	1.94	10.90	1.94	10.90	1.93	11.00
Stag(0.5, 0.3)	1.38	20.63	1.93	11.20	1.93	11.20	1.93	11.00
Stag(0.2, 0.3)	0.96	32.47	1.25	20.85	1.93	11.20	1.93	11.00
Random	0.66	38.75	1.07	19.28	1.93	11.20	1.93	11.00
Non-bijective	0.82	37.84	1.05	23.24	1.05	23.24	1.09	16.63

Next we evaluate Global First Fit and Simulated Annealing in comparison to static hashing and an idealized non-blocking switch for larger topologies using our simulator.

Communication Patterns

In Fig. 4.7 we show the aggregate bisection bandwidth achieved when running the benchmark suite for a simulated fat-tree network with 8,192 hosts (when k=32). We compare our algorithms against a hypothetical non-blocking switch for the entire data center and against static ECMP hashing. The performance of ECMP worsens as the probability of local communication decreases. This is because even for a completely fair and perfectly uniform hash function, collisions in path assignments *do* happen, either within the same switch or with flows at a downstream switch, wasting a portion of the available bandwidth. A global scheduler makes discrete flow placements that are chosen by design to reduce overlap. In most of these different communication patterns, our dynamic placement algorithms significantly outperform static ECMP hashing. Figure 4.8 shows the variation over time of the bisection bandwidth for the 1,024 host fat-tree network. Global First Fit and Simulated Annealing perform fairly close to optimal for most of the experiment.

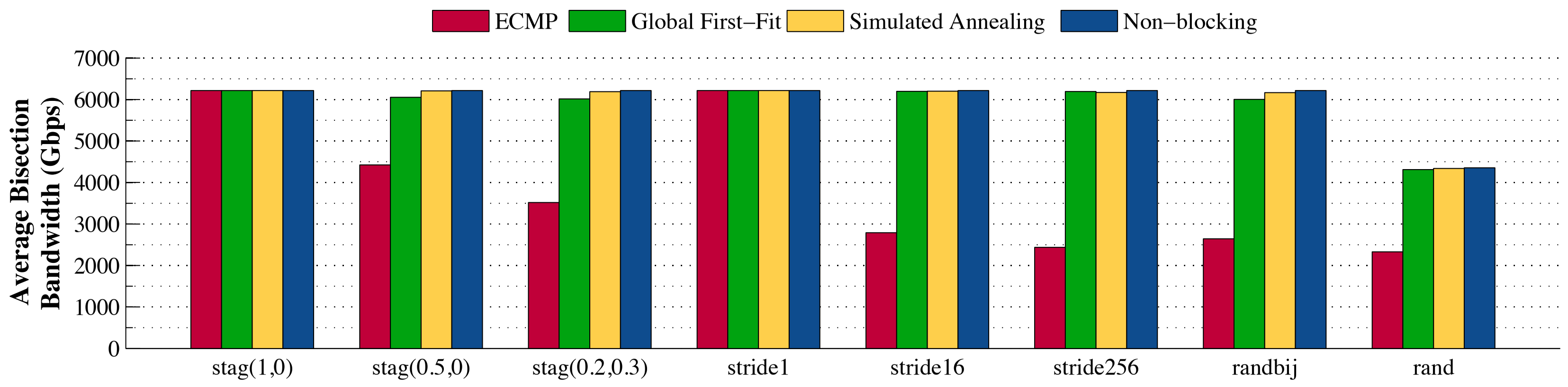


Figure 4.7: Comparison of scheduling algorithms for different traffic patterns on a fat-tree topology of 8,192-hosts.

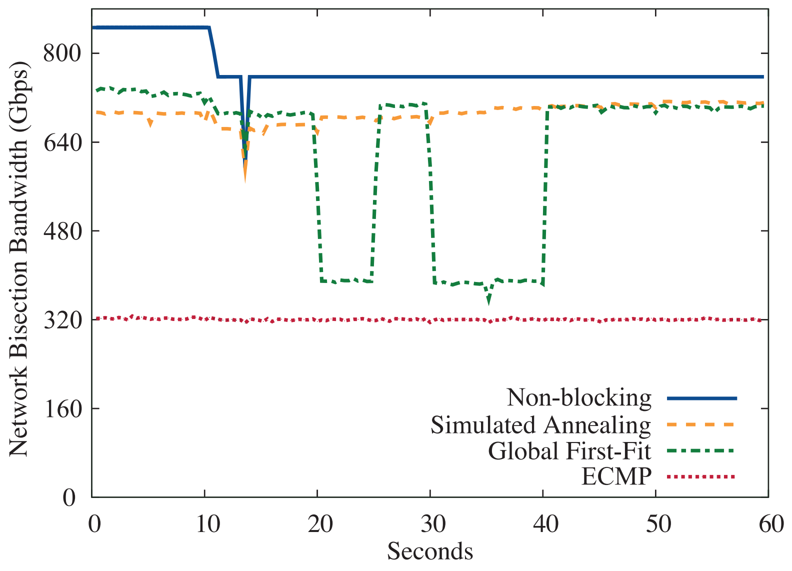


Figure 4.8: Network bisection bandwidth vs. time for a 1,024 host fat-tree and a random bijective traffic pattern.

Quality of Simulated Annealing

To explore the parameter space of Simulated Annealing, we show in Table 4.4 the effect of varying the number of iterations at each scheduling period for a randomized, non-bijective communication pattern. This table confirms our initial intuition regarding the assignment quality vs. the number of iterations, as most of the improvement takes place in the first few iterations. We observed that the performance of Simulated Annealing asymptotically approaches the best result found by Simulated Annealing after the first few iterations.

The table also shows the percentage of final bisection bandwidth for a random communication pattern as number of hosts and flows increases. This supports our belief that Simulated Annealing can be run with relatively few iterations in each scheduling period and still achieve comparable performance over time. This is aided by remembering core assignments across periods, and by the arrival of only a few new large flows each interval.

Table 4.4: Percentage of full bisection bandwidth vs. the Simulated Annealing iterations, for a case of a random, non-bijective traffic pattern. Also shown is the same load running on a non-blocking topology.

	Number of Hosts		
SA Iterations	16	1,024	8,192
1,000	78.73	74.69	72.83
50,000	78.93	75.79	74.27
100,000	78.62	75.77	75.00
500,000	79.35	75.87	74.94
1,000,000	79.04	75.78	75.03
1,500,000	78.71	75.82	75.13
2,000,000	78.17	75.87	75.05
Non-blocking	81.24	78.34	77.63

Complexity of Demand Estimation

Since the demand estimation is performed once per scheduling period, its runtime must be reasonably small so that the length of the control loop is as small as possible. We studied the runtime of demand estimation for different traffic matrices in data centers of varying sizes.

Table 4.5 shows the runtimes of the demand estimator for different input sizes. The reported runtimes are for runs of the demand estimator using 4 parallel threads of execution on a modest quad-core 2.13GHz machine. Even for a large data center with 27,648 hosts and 250,000 large flows (average of nearly 10 large flows per host), the runtime of the demand estimation algorithm is only 200ms. For more common scenarios, the runtime is approximately 50-100ms in our setup. We expect the scheduler machine to be a fairly high performance machine with more cores, thereby still keeping the runtime well under 100ms even for extreme scenarios.

The memory requirement for the demand estimator in our implementation using a sparse matrix representation is less than 20MB even for the extreme scenario with nearly 250,000 large flows in a data center with 27k hosts. In more common scenarios, with a reasonable number of large flows in the data center, the entire data structure would fit in the L2 cache of a modern CPU.

Table 4.5: Demand estimation runtime for different networks and number of large flows.

k	Hosts	Large Flows	Runtime (ms)
16	1,024	1,024	1.45
16	1,024	5,000	4.14
32	8,192	8,192	2.71
32	8,192	25,000	9.23
32	8,192	50,000	26.31
48	27,648	27,648	6.91
48	27,648	100,000	51.30
48	27,648	250,000	199.43

Table 4.6: Runtime (ms) vs. number of Simulated Annealing iterations for different number of flows f.

	1,024 Hosts		8,192 Hosts	
Iterations	$f = 3{,}215$	$f = 6{,}250$	$f = 25$k	$f = 50$k
1,000	2.997	5.042	6.898	11.573
5,000	12.209	20.848	19.091	32.079
10,000	23.447	40.255	32.912	55.741

Considering the simplicity and number of operations involved, an FPGA implementation can store the sparse matrix in an off-chip SRAM. An FPGA such as the Xilinx Virtex-5 can implement up to 200 parallel processing cores to process this matrix. We estimate that such a configuration would have a computational latency of approximately 5ms to perform demand estimation even for the case of 250,000 large flows.

Complexity of Simulated Annealing

In Table 4.6 we show the runtime of Simulated Annealing for different experimental scenarios. The runtime of Simulated Annealing is asymptotically independent of the number of hosts and only dependent on the number of flows. The main takeaway here is the scalability of our Simulated Annealing implementation and its potential for practical application; for networks of thousands of hosts and a reasonable number of

Table 4.7: Length of control loop (ms).

	Flows per Host		
Hosts	1	5	10
1,024	100.2	100.9	101.7
8,192	101.4	106.8	113.5
27,648	104.6	122.8	145.5

flows per host, the Simulated Annealing runtime is on the order of tens of milliseconds, even for 10,000 iterations.

Control Overhead

To evaluate the total control overhead of the centralized scheduling design, we analyzed the overall communication and computation requirements for scheduling. The control loop includes 3 components—all switches in the network send the details of large flows to the scheduler, the scheduler estimates demands of the flows and computes their routes, and the scheduler transmits the new placement of flows to the switches.

We made some assumptions to analyze the length of the control loop. (1) The control plane is made up of 48-port GigE switches with an average 10μs latency per switch. (2) The format of messages between the switches and the controller are based on the OpenFlow protocol (72B per flow entry) [91]. (3) The total computation time for demand estimation and scheduling of the flows is conservatively assumed to be 100ms. (4) The last hop link to the scheduler is assumed to be a 10GigE link. This higher speed last hop link allows a large number of switches to communicate with the scheduler simultaneously. We assumed that the 10GigE link to the controller can be fully utilized for transfer of scheduling updates.

Table 4.7 shows the length of the control loop for varying number of large flows per host. The values indicate that the length of the control loop is dominated by the computation time, estimated at 100ms. These results show the scalability of the centralized scheduling approach for large data centers.

4.5 Acknowledgement

Chapter 4, in part, contains material as it appears in the Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2010. Al-Fares, Mohammad; Radhakrishnan, Sivasankar; Raghavan, Barath; Huang, Nelson; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 5

User-extensible Active Queue Management

In this chapter, we address the challenge of network extensibility by bypassing the long lead-time and difficulty standardization process for adopting new AQM disciplines by the switch vendors. Instead, we present a model where new and experimental AQM disciplines can be deployed and evaluated directly in production datacenter networks, without modifying existing switches or end-hosts. To this end, we deploy user-programmable "bump on the wire" middleboxes, called *NetBumps*, that augment the existing switching infrastructure.[1] In this model, each NetBump provides a virtual queue primitive that enforces a range of AQM mechanisms at line rate, mechanisms that would normally have to be implemented in the switches themselves. Furthermore, due to its streamlined forwarding from input port to output port with no actual switching, we greatly reduce the latency imposed by NetBump since its functionality is limited to modifications of packets in flight, using a user-level, zero-copy, kernel-bypass network API, with no actual queuing or buffering done within.

By their placement at key points in the network, NetBumps enable AQM techniques to be incrementally deployed and evaluated in real-world traffic scenarios. For example, in a typical data center enviroment, NetBumps would be placed inline with the top-of-rack's (ToR) uplinks (see Fig. 5.1) to enable the traffic monitoring and enforcement

[1]The "bump on the wire" term here is unrelated to previous work about IPsec deployment boxes [74].

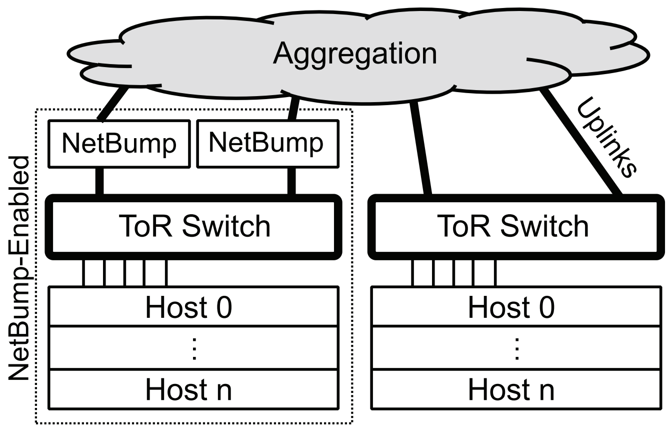


Figure 5.1: Deployment scenario in the data center. "NetBump-enabled racks" include NetBumps in-line with the Top-of-Rack (ToR) switch's uplinks, and monitor output queues at the host-facing ports.

of AQM policies and protocols for that rack. This way, NetBump-enabled swathes of the network could be expanded as needed.

By providing a clear and simple vAQM API, this makes implementing new network functions straightforward. In our experience, new queuing disciplines, congestion control strategies, protocol-specific packet headers (specifically, ones not supported by commercial hardware, e.g. for XCP [72]), and new packets (for a new congestion control protocol we implement in this work) can be built and deployed at line rate into existing networks. Network researchers can quickly develop and experiment with protocol parameters and design by simply modifying their NetBump applications.

5.1 Motivation

In this section we first present an example of NetBump functionality in action, and then motivate our requirements for a low-latency implementation.

5.1.1 NetBump Example

In Fig. 5.2, we show a simple network where two source hosts H_1 and H_2 each send data to a single destination host H_d (in flows F_1 and F_2, respectively). H_1 and H_2 are

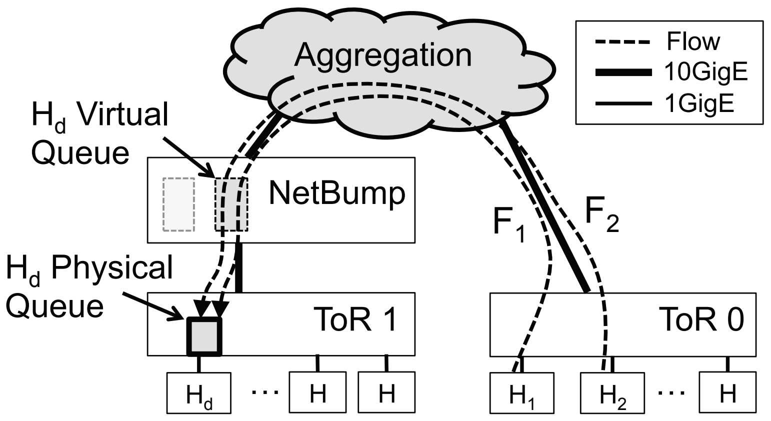


Figure 5.2: An example of NetBump installed on top of a Top-of-Rack switch, monitoring downstream physical queues.

connected to Switch0 at 1Gbps. Switch 0 has a 10Gbps uplink to the NetBump (through the aggregation layer), and on the other side of the NetBump is a second 10Gbps link to Switch1. Destination host H_d is attached to Switch1 at 1Gbps. Flows F_1 and F_2 each have a maximum bandwidth of 1Gbps, and since host H_d has only a single 1Gbps link, congestion will occur on H_d's input or output port in Switch1 if $rate(F_1) + rate(F_2) >$ 1Gbps. Without NetBump, assuming Switch1 implements a drop-tail queuing discipline, packets from F_1 and F_2 will be interleaved in H_d's physical queue until the queue becomes full, at which point Switch1 will drop packets arriving to the full queue. This leads to known problems such as burstiness and lack of fairness.

Instead, as NetBump forwards packets from its input port to its output port, it estimates the occupancy of a virtual queue associated with H_d's output port buffer. When a packet arrives, H_d's virtual queue occupancy is increased by the packet's size. Because the NetBump knows the speed of the link between Switch1 and H_d (§ 5.2.1), it can compute the estimated *drain rate*, or the rate that data leaves H_d's queue. By integrating this drain rate over the time between subsequent packets, it calculates the amount of data that has left the queue since the last packet arrival.

Within NetBump, applications previously requiring new hardware development can instead act on the virtual queue. For example, to implement a Random Early Detection (RED) queuing discipline, the NetBump shown in Fig. 5.2 maintains a virtual output queue for each physical queue in Switch1. This virtual queue maintains two parameters, `MinThreshold` and `MaxThreshold`, as well as a weighted estimate of the current downstream queue length. According to the RED discipline, packets are sent unmodified when the moving average of the queue length is below the `MinThreshold`, the packets are marked (or dropped) probabilistically when the average is between the two thresholds, and packets are unconditionally marked (or dropped) when it is above `MaxThreshold`.

Note that in this example, just as in all the network mechanisms presented in this chapter, packets are never delayed or queued in the NetBump itself. Instead, NetBump marks, modifies, or drops packets at line rate as if the downstream switch directly supported the functionality in question. Note also that NetBump is not limited to a single queuing discipline or application–it is possible to compose multiple applications (e.g. QCN congestion control with Explicit Congestion Notification (ECN) marking [42]). Furthermore, AQM functionality can act only on particular flows transiting a particular end-to-end path if desired.

5.1.2 Design Requirements

The primary goal of NetBump is enabling rapid and easy evaluation of new queue management and congestion control mechanisms in deployed networks with minimal intrusiveness. We next describe the requirements that NetBump must meet to successfully reach this goal.

Deployment with unmodified switches and end-hosts: We seek to enable AQM development and experimentation to take place in the data center or enterprise itself, rather than separate from the network. This means that NetBump works despite leaving switches and end-hosts unmodified. Thus a requirement of NetBump is that it implements a virtual Active Queue Management (vAQM) discipline that tracks the status of neighboring switch buffers. This will differ from previous work that applies this technique within switches [48, 79], as our implementation will be remote to the switch.

Distributed deployment: Modern networks increasingly rely on multipath topologies both for redundancy in the face of link and switch failure, and for improving throughput by utilizing several, parallel links. Left unaddressed, multipath poses a challenge for the NetBump model since a single bump may not be able to monitor all of the flows heading to a given destination. Therefore a requirement for NetBump is that it supports enough throughput to manage a sufficient number of links, and that it supports a distributed deployment model. In a distributed model, multiple bumps deployed throughout the network coordinate with each other to manage flows transiting them. In this way, a set of flows taking separate network paths can still be subjected to a logically centralized, though physically distributed, AQM policy.

Ease of development: Rather than serving as a final deployment strategy, we see NetBump as an experimental platform, albeit one that is deployed directly on the production network. Thus rapid prototyping and reconfiguration are a requirement of its design. Specifically, the platform should export a clear API with which users can quickly develop vAQM applications using C/C++.

Minimizing latency: Many data center and enterprise applications have strict latency deadlines, and any datapath processing elements must likewise have strict performance guarantees, especially given NetBump's target deployment environment of data center networks, whose one-way latency diameters are measured in microseconds. Since the throughput of TCP is in part a function of the network round-trip time [103], any additional latency imposed by NetBump can affect application flows. To show this effect, we measured the completion times of two flows–one in which a single byte is exchanged between a sender-receiver pair, and one where 1MB is exchanged between that same pair. Fig. 5.3 shows the normalized completion times of each flow as a function of one-way middlebox latency. Perhaps not surprising, adding even tens of microseconds of one-way latency has a significant impact on flow completion times when the baseline network RTT is very small.

Forwarding at line rate: Although data center hosts still primarily operate at 1Gbps, 10Gbps has become standard at rack-level aggregation. Deploying a NetBump inline with top-of-rack uplinks and between 10Gbps switches will require an implementation that can support 10Gbps line rates. The challenge then becomes keeping up with packet arrival rates: 10Gbps corresponds to 14.88M 64-byte minimum sized packets per second, including Ethernet overheads.

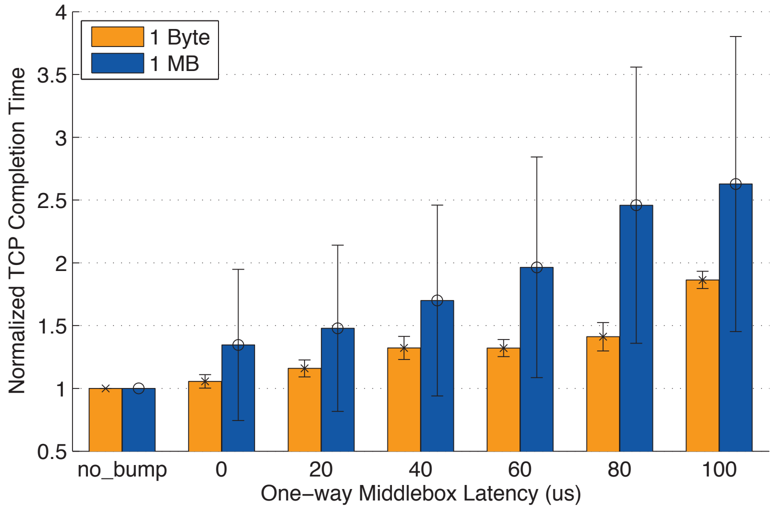


Figure 5.3: Effect of middlebox latency on completion time of short (1 Byte) and medium-sized (1MB) TCP flows. Baseline (direct-connect) transfer time was 213μs (1B), 9.0ms (1MB), others are through a NetBump with configurable added delay.

5.2 Design

In this section we describe the design of the primary NetBump vAQM pipeline, including how this design can scale to support faster links and a distributed deployment for multi-path data center designs. We discuss our implementation choices in § 5.4.

5.2.1 The NetBump Pipeline

The core NetBump pipeline consists of four algorithms: 1) packet classification, 2) virtual queue (VQ) drain estimation, 3) packet marking/dropping, and optionally 4) extensible packet processing.

Virtual Queue Table Data Structure: Each NetBump maintains a set of virtual queues, which differ from physical queues in that they do not store or buffer packets. Instead, as packets pass through a virtual queue, it maintains state on what its occupancy *would* be if it were actually storing packets. Thus each virtual queue must keep track of 1) the number and sizes of packets transiting it, 2) the packet arrival times, and 3) the virtual rate at which they drain from the queue. Note that packets actually drain at line rate (i.e.

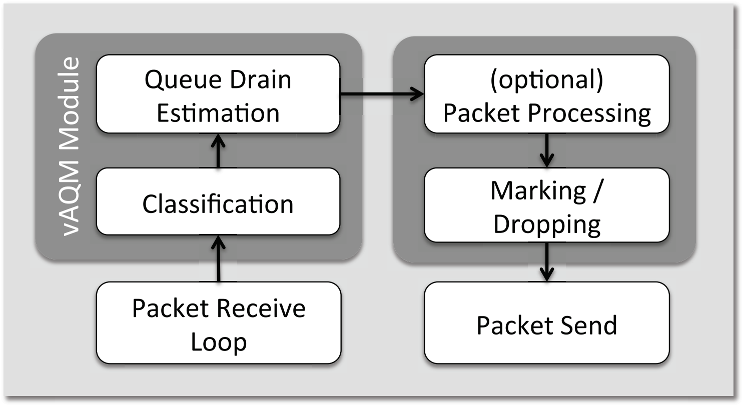


Figure 5.4: The NetBump pipeline.

10Gbps), however a virtual queue could be configured with a virtual drain parameter of 1Gbps or 100Mbps.

The virtual queue table is a simple data structure kept by the NetBump that stores each of these three parameters for each virtual queue supported by that bump. For the AQM functionality we consider, we only need to know the virtual queue occupancy and drain rate, and so each virtual queue keeps 1) the size in bytes of the queue, 2) the time the last packet arrived to the queue, and 3) the virtual queue drain rate. These values are updated when a packet arrives to the virtual queue.

1. Packet Classification: As packets arrive to the NetBump, they must first be classified to determine into which virtual queue they will be enqueued. This classification API is extensible in NetBump, and can be overridden by a user as needed. A reasonable scheme would be to map packets to virtual queues corresponding to the downstream physical switch output buffer that the packet will reside in when it leaves the bump. In this case the virtual queue is emulating the downstream switch port directly. Note that virtual queues do not have to be associated in this way, though they are for most of the applications we consider in this work.

To make this association, NetBump requires two pieces of information: the mapping of packet destinations to downstream output ports, and the speed of the link attached to that port. The mapping is needed to determine the destination virtual queue for a particular packet, and the link speed is necessary for estimating the virtual queue's drain rate. There are a variety of ways of determining these values. The bump could query

Table 5.1: The NetBump API.

Function	Description
`void init(vQueue *vq, int rate);`	Initializes a virtual queue and set the drain rate
`vQueue * classify(Packet *p);`	Classifies a packet to a virtual queue
`void vAQM(Packet *p, vQueue *vq);`	Updates internal vAQM state
`int estimateQlen(vQueue *vq);`	Returns an estimate of a virtual queue's length
`int process(Packet *p, vQueue *vq);`	Packet processing (modify, duplicate, drop, etc.)
`int forward(Packet *p) const;`	Gets the output port (for multi-NIC bumps)

neighboring switches (e.g. using SNMP) for their outgoing link speeds, or those values could be statically configured when the bump is placed in the network. For software-defined networks based on OpenFlow [53,91], the central controller could be queried for host-to-port mappings and link speeds, as well as the network topology. In our evaluation, we statically configure the NetBump with the port-to-host mapping and link speeds.

2. Queue Drain Estimation: The purpose of the queue drain estimation algorithm is to calculate, at the time a packet is received into the bump, the occupancy of the virtual queue associated with the packet (Fig. 5.5). The virtual queue estimator is a leaky bucket that is filled as packets are assigned to it, and drained according to a fixed drain rate determined by the port speed [128].

Lines 1-6 implement the leaky bucket. First, the elapsed time since the last packet arrived to this virtual queue is calculated. This elapsed time is multiplied by a physical port's rate to calculate how many bytes would have left the downstream queue since receiving the last packet. The physical port's drain rate comes from the link speed of the downstream switch or end-host. This amount is then subtracted from the current estimate (or set to zero, if the result would be negative) of queue occupancy to get an updated occupancy. If this is the first packet to be sent to that port, then the default queue occupancy estimate of 0 is used instead. Lastly, the "last packet arrival" field of the virtual queue is updated accordingly.

A key design decision in NetBump is whether to couple the size of the virtual queue inside the bump with the actual size of the physical buffer in the downstream switch. If we knew the size of the downstream queue, then we could set the maximum allowed occupancy of the virtual queue accordingly. This would be challenging in gen-

```
Procedure vAQM(Packet *pkt, vQueue *VQ):
1   if (VQ→lastUpdate > 0) {
2     elapsedTime = pkt→timestamp – VQ→lastUpdate
3     drainAmt = elapsedTime * VQ→rate
4     VQ→tokens –= drainAmt
5     VQ→tokens = max(0, VQ→tokens)
6   }
7   VQ→tokens += pkt→len
8   VQ→lastUpdate = pkt→timestamp

Procedure RED(Packet *pkt, vQueue *VQ):
9   VQ→avg = calculateAvg(v→tokens)
10  if (VQ→avg > VQ→MaxThresh) {
11    VQ→tokens –= pkt→len
12    drop(pkt)
13  } else if (VQ→avg > VQ→MinThresh) {
14    calculate probability ρ
15    with probability ρ:
16      mark(pkt)
17  }
```

Figure 5.5: The queue drain estimation algorithm and the implementation of RED.

eral, since switches do not typically export the maximum queue size programmatically. Furthermore, for shared buffer switches, this quantity might change based on the instantaneous traffic in the network. In fact, by assuming a small buffer size in the virtual queue within NetBump, we can constrain the flow of packets to reduce actual buffer occupancy throughout the network. Thus, assuming small buffers in our virtual queues has beneficial effects on the network, and simplifies NetBump's design.

3. Packet Marking/Dropping: At line 9 in Fig. 5.5, NetBump has an estimate for the virtual queue occupancy. Here a variety of actions can be performed, based on the application implemented in the bump. The example code shows a general random-early drop (RED) application [58]. In this example, there is a "min" limit that results in packet marking, and a "max" limit that results in packet dropping. Packet marking takes the form of setting the ECN bit in the header, and dropping is performed simply in software.

4. Extensible Processing Stage: In addition to the vAQM estimation and packet marking/dropping functionality built into the basic NetBump pipeline, developers can optionally include arbitrary additional packet processing. NetBump developers can include extensions to process packet streams. This API is quite simple, in that the extension is called once per packet, which is represented by a pointer to the packet data and length field. Developers can read, modify, and adjust the packet arbitrarily before re-injecting the packet back into the NetBump pipeline (or dropping it entirely).

Packets destined to particular virtual queues can be forwarded to different extensions, each of which runs in its own thread, coordinating packet reception from the NetBump pipeline through a shared producer-consumer queue. By relying on multi-core processors, each extension can be isolated to run on its own core. This has the advantage that any latency induced by an extension only affects the traffic subject to that extension. Furthermore, correctness or performance bugs in an extension only affects the subset of network traffic enqueued in the virtual queues serving that extension. This enables an incremental "opt-in" experimental platform for introducing new NetBump functionality into the production network.

An advantage of the NetBump architecture is that packets travel a single path from the input port to the output port. Thus, unlike multi-port software routers, here packets can remain entirely on a single core, and stay within a single cache hierarchy. The only point of synchronization is the shared vAQM data structure, and we study the overhead of this synchronization and the resulting lock contention in § 5.5.2.

5.2.2 Scaling NetBump

Managing packet flows in multipath environments requires that NetBump scale with the number of links carrying a particular set of flows. This scaling operates within two distinct regions. First, supporting additional links by adding NICs and CPU cores to a single server, and second, through a distributed deployment model.

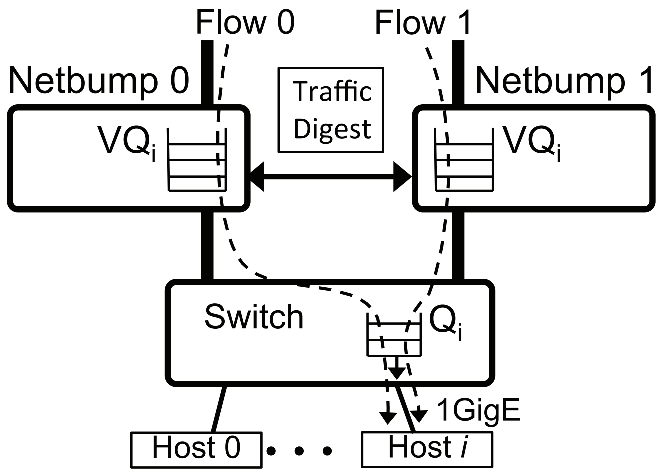


Figure 5.6: Flow0 and Flow1 both destined to $Host_i$, with two NetBumps monitoring the same Q_i buffer.

Multi-link NetBump

For multipath environments in which packets headed to a single destination might travel over multiple paths, it is possible to scale NetBump by simply adding new NICs and CPU cores. For example, a top-of-rack switch with two 10Gbps uplinks would meet these requirements. Here, a single server is only limited in the number of links that it can support by the amount of PCI bandwidth and the number of CPU cores. Each pair of network interfaces supports a single link (10Gbps in, and 10Gbps out), and PCIe gen 2 supports up to three such bi-directional links. In this case, "Multi-link" NetBump is still conceptually simpler than a software-based router, since packets still follow a single-input, single-output path. Each supported link is handled independently inside the bump, and we can assign to it a dedicated CPU core. The only commonality between these links is the vAQM table, which is shared across the links.

Distributed NetBump

For multi-path environments, where NetBumps must be physically separated, or for those with more links than are supported by a single server, we consider a distributed NetBump implementation. Naturally, if multiple NetBumps contribute packets to a shared downstream buffer, they must exchange updates to maintain accurate vAQM estimates. Note that the vAQM table maintains queue estimates for each of neighboring switch's ports (or a monitored subset).

In this case, where we assume the topology (adjacency matrix and link speeds) to be known in advance, NetBumps update their immediate neighbor bumps about the traffic they have processed (Fig. 5.6). Hence, updates are not the queue estimate itself, but tuples of individual packet lengths and physical downstream switch and port IDs, so that forwarding tables need not be distributed. Each *source* NetBump sends an update to its monitoring neighbors at a given tunable frequency (e.g. per packet, or batched), and each *destination* NetBump calculates a new queue estimate by merging its previous estimate with the traffic update from its neighbor, according to the algorithm in Fig. 5.5. In this design, updates are tiny; 4B per monitored flow packet (i.e. 2B for packet size and 2B for the port identifier). This translates to about 3MB/s of control traffic per 10Gbps monitored flow. Note also that updates can be transmitted on a dedicated link, or in-band with the monitored traffic. We chose the latter for our Distributed NetBump implementation.

The above technique introduces two possible sources of queue estimation error: 1) batching updates causes estimates to be slightly stale, and since packet sizes are not uniform, the individual packet components of a virtual queue and their respective order would not necessarily be the same, and 2) the propagation delay of the update. Despite this incremental calculation, the estimation naturally synchronizes whenever the buffer occupancy is near its empty/full boundaries.

5.3 Deployed Applications

In this section, we describe the design and implementation of several vAQM and congestion control applications we developed with NetBump.

5.3.1 Random Early Detection

The first vAQM scheme we implemented is Random Early Detection (RED) [43]. The goal of RED is to keep the average queue length low while achieving high throughput. RED buffer management consists of two parts: estimation of the average queue size using an exponentially-weighed moving average, and a decision on dropping or marking a packet. The packet mark or drop rate increases linearly from zero, when the average queue length is at `MinThresh`, to a maximum probability when the average queue length reaches `MaxThresh`. Our implementation is based on the algorithm proposed in the original RED paper. This vAQM stage maintains `MinThresh` and `MaxThresh` watermarks (with values of 20 and 60 packets, respectively), a w_q setting of 0.1, and varies max_p. We based these values on previous work by Floyd et al. [107].

5.3.2 Data Center TCP

We next implemented Data Center TCP (DCTCP) [5] on NetBump. The purpose of DCTCP is to improve the behavior of TCP in data center environments, specifically by reducing queue buildup, buffer pressure, and incast. It requires changes to the end-hosts as well as network switches. A DCTCP-enabled switch marks the ECN bits of packets when the size of the output buffer in the switch is greater than the marking threshold K. Unlike RED, this marking is based on instantaneous queue size, rather than a smoothed average. The receiver is responsible for signaling back to the sender the particular sequence of marked packets (see [5] for a complete description), and the sender maintains an estimate α of the fraction of marked packets. Unlike a standard sender that cuts the congestion window in half when it receives an ECN-marked acknowledgment, a DCTCP sender reduces its rate according to: $cwnd \leftarrow cwnd * (1 - \alpha/2)$. We support DCTCP in the end-hosts by using a modified Linux TCP stack supplied by Kabbani and Alizadeh [67].

Implementing DCTCP in NetBump was straightforward, and relied on much of the same code as RED. Here, instead of computing a smoothed queue average of the downstream physical queue occupancy, we mark based on the instantaneous queue size. Next, we set both `LowThresh` and `HighThresh` to the supplied K (chosen to be 20 packets, based on the authors' guidelines [5]). We experimented with other values of K, and found that changing this value had little noticeable effect on aggregate throughput or rate convergence.

5.3.3 Quantized Congestion Notification

We also implemented the IEEE 802.1Qau-QCN L2 Quantized Congestion Control (QCN) algorithm [4]. QCN-enabled switches monitor their output queue occupancies and upon sensing congestion (using a combination of queue buildup rate and queue occupancy), they send feedback packets to upstream *Reaction Points*. The Reaction Points are then responsible for adjusting the sending rate according to a prescribed formula. For every QCN-enabled link, there are two basic algorithms:

Congestion Point (CP): For every output queue, the switch calculates a *feedback measure* (F_b) whenever a new frame is queued. This measure captures both the *rate* at which the queue is building up (Q_δ), as well as the *difference* (Q_{off}) between the current queue occupancy and a desired equilibrium threshold (Q_{eq}, assumed to be 20% of the physical buffer). If Q denotes the current queue occupancy, Q_{old} is the previous iteration, and w is the weight controlling rate build-up, then:

$$Q_{off} = Q - Q_{eq} \qquad Q_\delta = Q - Q_{old}$$

$$F_b = -\left(Q_{off} + wQ_\delta\right)$$

Based on F_b, the switch probabilistically generates a congestion notification frame proportional to the severity of the congestion (the probability profile is similar to RED [43], i.e. it starts from 1% and plateaus at 10% when $|F_b| \geq F_{bmax}$). This QCN frame is destined to the upstream reaction point from which the just-added frame was received. If $F_b \geq 0$, then there is no congestion and no notification is generated.

Reaction Point (RP): Since the network generates signals for rate decreases, QCN senders must probe for available bandwidth gradually until another notification is received. The reaction point algorithm has two phases: Fast-Recovery (FR) and Additive-Increase (AI), similar, but independent from, BIC-TCP's dynamic probing.

The RP algorithm keeps track of the sending Target Rate (TR) and Current Rate (CR). When a congestion control frame is received, the RP algorithm immediately enters the Fast Recovery phase; it sets the target rate to the current rate, and reduces the current rate by an amount proportional to the congestion feedback (by at most 1/2). Barring further congestion notifications, it tries to recover the lost bandwidth by setting the current rate to the average of the current and target rates, once every *cycle* (where a cycle is defined in the base byte-counter model as 100 frames). The RP exits the Fast Recovery phase after five cycles, and enters the Additive Increase phase, where the RP continually probes for more bandwidth by adding a constant increase to its target rate (1.5Mbps in our implementation), and again setting the current sending rate to the average of the CR and TR.

5.4 Implementation

NetBump can be implemented using a wide variety of underlying technologies, either in hardware or in software. We evaluated three such choices: 1) the stock Linux-based forwarding path, 2) the RouteBricks software router, and 3) a user-level application relying on kernel-bypass network APIs to read and write packets directly to the network. We call this last implementation *UNetBump*. We show in Fig. 5.7 the latency distributions of these systems when forwarding 1500B packets at 10Gbps (except Linux with 9000B). The baseline for comparison being a simple loopback.

All of our implementations are deployed on HP DL380G6 servers with two Intel E5520 four-core CPUs, each operating at 2.26GHz with 8MB of cache. These servers have 24 GB of DRAM separated into two 12GB banks, operating at a speed of 1066MHz. For the Linux and UNetBump implementations, we use an 8-lane Myricom 10G-PCIE2-8B2-2S+E dual-port 10Gbps NIC which has two SFP+ interfaces, plugged into a PCI-Express Gen 2 bus. For RouteBricks, we used an Intel E10G42AFDA dual-port 10Gbps NIC (using an 82598EB controller) with two SFP+ interfaces.

5.4.1 Linux

The Linux kernel natively supports a complete IP forwarding path, including a configurable set of queuing disciplines that are managed through the "traffic control (tc)" extensions [84]. Linux `tc` supports flow and packet shaping, scheduling, policing, and dropping. While `tc` supports a variety of queuing disciplines, it does not support managing the queues of remote switches. This support would have to be added to the kernel. In our evaluation we used Linux kernel version 2.6.32. We found that the latency

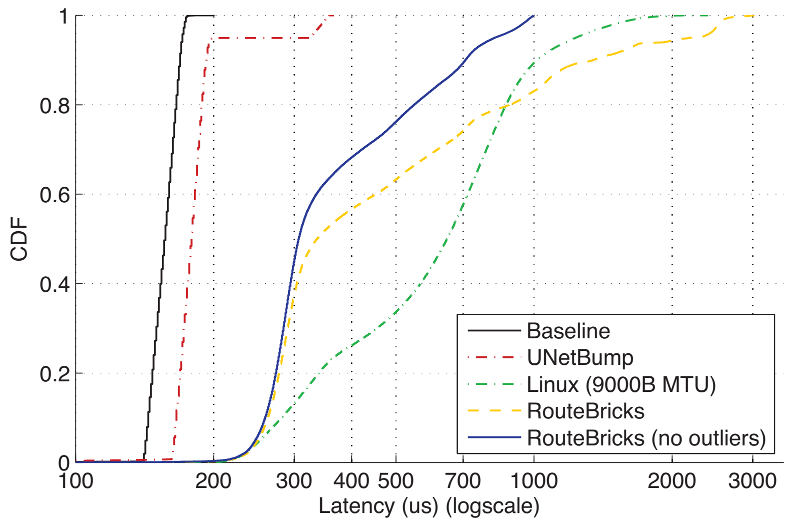


Figure 5.7: Forwarding latency of baseline, UNetBump, Linux, RouteBricks (batching factor of 16, and a Click burst factor of 16), both with and without an outlier queue.

overheads of the Linux forwarding path were very high, with a mean latency above 500μs, and a 99th percentile above 1500μs. Furthermore, our evaluation found that Linux was not able to forward non-Jumbo frames at speeds approaching 10Gbps (and certainly not with minimum-sized packets). Based on these microbenchmarks, we decided not to further consider Linux as an implementation alternative.

5.4.2 RouteBricks

RouteBricks [32] is a high-throughput software router implementation built using Click's core, extensive element library, and specification language. It increases the scalability of Click in two ways–by improving the forwarding rate within a single server, and by federating a set of servers to support throughputs beyond the capabilities of a single server. To improve the scalability within a single server, RouteBricks relies on a re-architected NIC driver that supports multiple queues per physical interface. This enables multiple cores to read and write packets from the NIC without imposing lock contention, which greatly improves performance [31, 35, 90, 137]. Currently, RouteBricks works only with the **`ixgbe`** device driver, which delivers packets out of the driver in fixed-size batches of 16 packets each. We built a single-node RouteBricks server using the same HP server architecture described above, but with the Intel E10G42AFDA NIC (the only available NIC that RouteBricks driver patch still supported). This server used the Intel **`ixgbe`** driver (version 1.3.56.5), with a batching factor of 16.

The use of this batching driver improves throughput by amortizing the overhead of transferring those packets over an entire batch, rather than on a packet-by-packet basis. This enables RouteBricks to support very high line rates, however the use of batching increases latency on an individual packet basis.[2] We also tried a batching factor of 1 and found that the throughput dropped below 10Gbps. Indeed RouteBricks was designed for high throughput, not low-latency. There is nothing in the Click or RouteBricks model that precludes low-latency forwarding, however for this work we chose not to use RouteBricks.

5.4.3 UNetBump

In user-level networking, instead of having the kernel deliver and demultiplex packets, the NIC instead delivers those packets directly to the application. This is typically coupled with kernel-bypass support, which enables the NIC to directly copy packets into a memory region mapped to the application. User-level networking has been further developed to better support virtualization by enabling individual flows to bypass the hypervisor and terminate directly in a guest VM [85].

User-level networking is a well-studied approach that has been implemented in a number of commercially-available products. Myricom offers Ethernet NICs with user-level networking APIs that we use in our evaluation. Intel supports user-level, kernel-bypass networking via the PF_RING/DNA driver [104]. SolarFlare offers a set of "Solarstorm" NICs with this functionality [121], as does SMC [118]. There have been at least two efforts to create an open and cross-vendor API to user-level, kernel-bypass network APIs [102, 108]. We re-evaluate the use of user-level networking to support low-latency applications, especially those requiring low latency variation. Note that it is possible to layer the RouteBricks/Click runtime on top of the user-level, kernel-bypass APIs we use in UNetBump.

[2]Despite extensive debugging with the help of the RouteBricks authors, we could not lower this latency further.

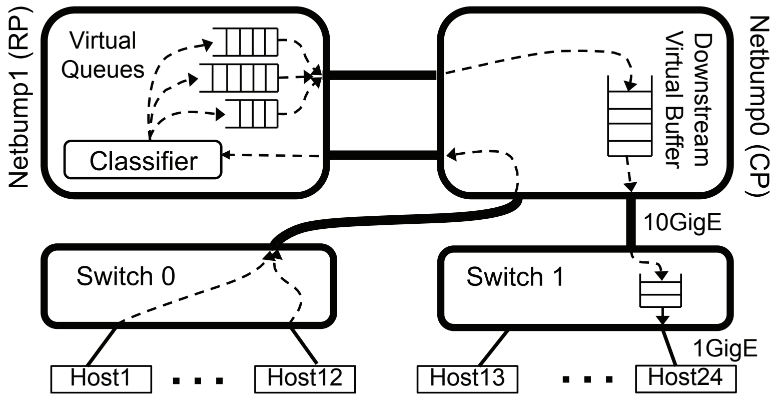


Figure 5.8: Two-rack 802.1Qau-QCN and DCTCP Testbed

5.5 Evaluation

Our evaluation seeks to answer the following questions: 1) How expressive is NetBump? 2) How easy is it to deploy applications? 3) How effective is vAQM estimation in practice? 4) What are the latency overheads and throughput limitations of NetBump?

To answer these, we built and deployed a set of NetBump prototypes in our experimental testbed. We started by evaluating the baseline latency and latency variation of the range of implementation choices. Based on these measurements, we proceeded with construction of UNetBump, a fully-functional prototype based on user-level networking APIs. We then evaluate a range of AQM functionalities with UNetBump.

5.5.1 Testbed Environment

Our experimental testbed consists of a set dual-processor Nehalem server described above, using either Myricom NICs, or in the case of RouteBricks, the Intel NIC. The Myricom NICs use the Sniffer10G driver version 1.1.0b3. We use copper direct-attach SFP+ connectors to interconnect the 10Gbps end-hosts to our NetBumps. Experiments with 1Gbps end-hosts rely on a pair of SMC 8748L2 switches that each have 1.5MB of shared buffering across all ports. Each SMC switch has a 10Gbps uplink that we connect to the appropriate NetBump.

We evaluate NetBump in three different contexts. The first is in microbenchmark, to examine its throughput and latency characteristics. Here we deploy NetBump as a loopback (simply connecting the two ports to the same host) to eliminate the effects of clock skew and synchronization. The second simply puts a NetBump inline between two machines, and tests NetBump's operation at full 10Gbps. Separating the source and destination to different machines enables throughput measurement with real traffic.

The third testbed, Fig. 5.8, evaluates NetBump in a realistic data center environment in which it might be deployed right above the top-of-rack switch. Here, we have two twelve-node racks of end-hosts, each connected to a 1Gbps switch. A 10Gbps uplink connects the two 1Gbps switches and the NetBump is deployed inline with those uplinks. In this case, the NetBump actually has four 10Gbps interfaces–two to the uplinks of each of the two SMC 1Gbps switches, and two that connect to a second NetBump. We use this testbed to evaluate 802.1Qau-QCN, with one NetBump acting as the Congestion Point (CP) and the other as the Reaction Point (RP).

5.5.2 Microbenchmarks

NetBump Latency

A key metric for evaluation is the latency overhead. To measure this, we use a loopback testbed and had a packet generator on the client host send packets onto the wire, through the NetBump, and back to itself. To calibrate, we also replace the NetBump with a simple loopback wire, which gives us the baseline latency overhead of the measurement host itself. We subtract this latency from the observed latency with the NetBump in place, giving us the latency of just the NetBump. We generated a constant stream of 1500-byte packets sent at configurable rates (Fig. 5.9).

For UNetBump, the latency is quite low for the majority of forwarded packets. There is a jump in latency at the tail due to NIC packet batching when they arrive above a certain rate. There is no way to disable this batching in software, even though we were only using a single CPU core which could have serviced a higher packet rate without requiring batching. The forwarding performance of UNetBump was sufficient to keep up with line rate using minimum-sized packets and a single CPU core.

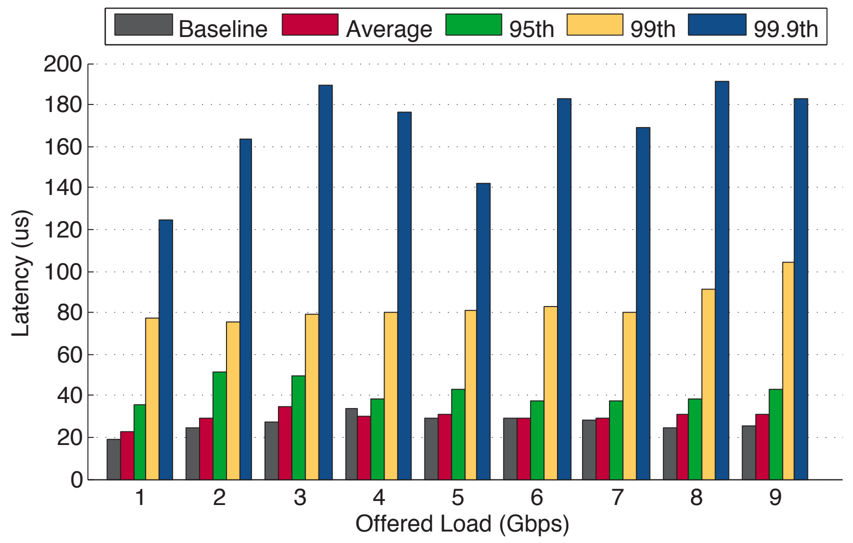


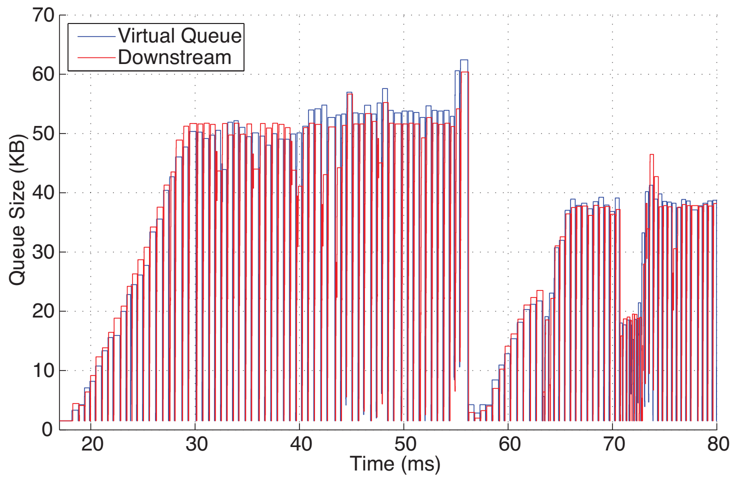
Figure 5.9: Latency percentiles imposed by UNetBump vs. offered load. Baseline is the loopback measurement overhead.

vAQM Estimation Accuracy

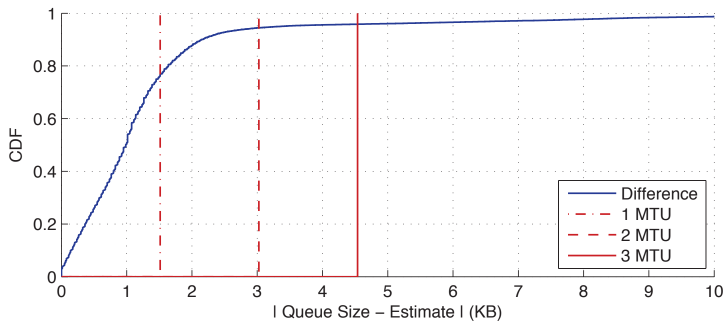
To evaluate the accuracy of the vAQM estimation, we ran **`iperf`** sessions between two hosts, connected in series by a NetBump and another pass-through machine (which records the timestamps of incoming frames). Since we cannot export physical buffer occupancy of commercial switches, we use the frame timestamps and lengths from the downstream pass-through machine to recreate the output buffer size over time, knowing the drain-rate. Fig. 5.10 shows the NetBump virtual queue size vs. the actual downstream queue. The estimate was within two MTUs 95% of the time.

Distributed NetBump

We also measured the accuracy of queue estimation when multiple NetBumps exchange updates to estimate a common downstream queue. In the first experiment, measure the effect of update latency on queue estimate accuracy. We varied the timestamp interleaving of two TCP **`iperf`** flows that share a downstream queue in order to simulate receiving delayed updates from a neighboring NetBump. Fig. 5.11 shows the CDF of the difference between the delayed inter-bump estimation and the in-sync version. Even when update latency was 25μs, the difference was always under 2MTU.



(a) Virtual queue size vs. actual downstream queue. Running an **`iperf`** TCP session between two 10G hosts, rate-limited to 1Gbps with a 40KB buffer downstream to induce congestion.



(b) CDF of the queue size difference. The estimate is within two 1500B MTUs 95% of the time.

Figure 5.10: Downstream vAQM estimation accuracy.

Next, we show the accuracy of NetBump's queue estimating of a downstream queue, based solely on updates from its neighbor. In our implementation, the updates are transmitted in-band with the monitored traffic. Fig. 5.12 gives the CDF of the difference between the actual queue size and the distributed NetBump estimate. We observe that the estimate is within 3MTUs 90% of the time. Note, however, the effect of update batching: estimates quickly drift when updates are delayed. Fig. 5.13 shows a typical relative difference CDF when background elephant flows are present (i.e. some flows are observed directly, and others indirectly through updates).

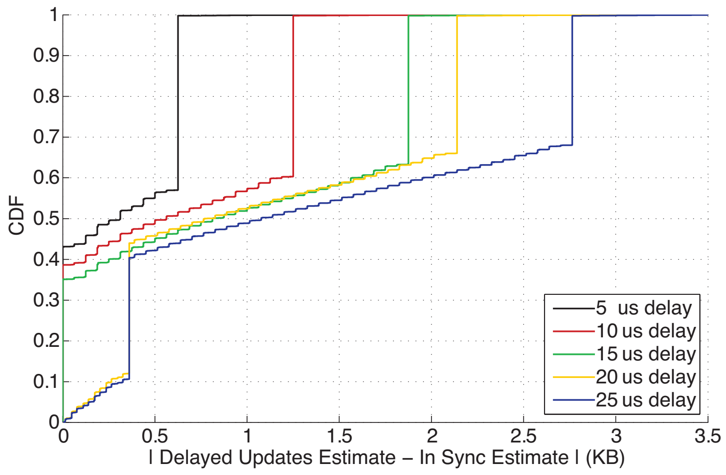


Figure 5.11: CDF of the absolute difference between the queue estimate with delayed updates and the in-sync version. The combined throughput is rate-limited to 5Gbps, and the downstream buffer is 40KB.

Effect of assigning CPU affinity

One of the challenges of designing NetBump was not only maintaining a low average latency, but also reducing variance. Modern CPU architectures provide separate cores on the same die and physically separate memory across multiple Non-Uniform Memory Access (NUMA) banks. This means that access time to memory banks changes based on which core issues a given request. To reduce latency outliers, we allocated memory to each UNetBump thread from the same NUMA domain as the CPU core it was scheduled to.

Given the significant additional latency that may be introduced by the unmodified Linux kernel scheduler, we compare latency of NetBump with and without CPU-affinity and scheduler modifications. Our control experiment uses default scheduling. To improve on this, we exclude all but one of the CPU cores from the default scheduler, and ensure that the UNetBump user-space programs execute on the reserved cores. We then examined the average, 95th, 99th, 99.9th, and maximum latencies through NetBump compared to the baseline (Table 5.2). CPU-affinity had a minor effect on latency on average, but was most pronounced on outlier packets. The maximum observed latency was 17 times smaller with CPU-affinity at the 99.9th percentile, showing the importance of explicit resource isolation in low-latency deployments.

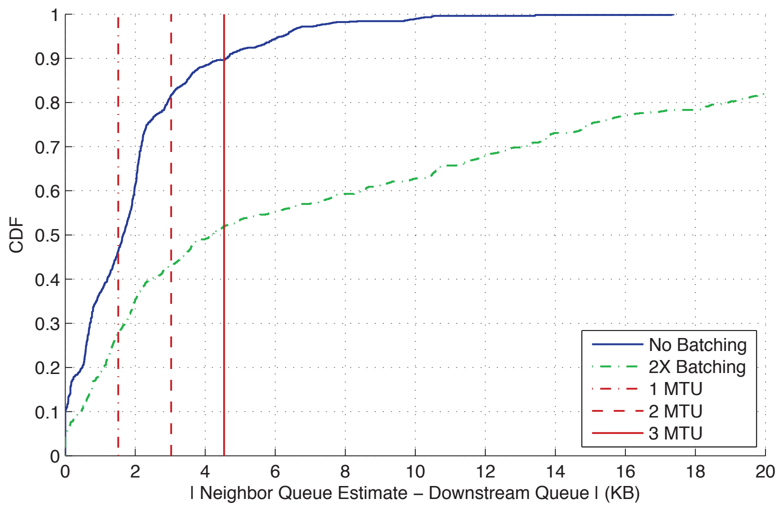


Figure 5.12: CDF of the difference between actual queue size and the Distributed NetBump estimate using a 1Gbps rate-limited TCP flow and a 40KB buffer. The estimating NetBump does not observe the monitored traffic directly.

Table 5.2: UNetBump latency percentiles vs. CPU core affinity.

Latency (μs)	**Avg**	**95th**	**99th**	**99.9th**	**Max**
No Affinity	32	39	76	1,322	3,630
With Affinity	30	42	83	169	208

Multicore performance

In UNetBump, basic vAQM estimation can be done at 10Gbps using only a single CPU core. However, to support higher link rates, additional cores might be necessary. The NIC itself will partition flows across CPU cores using a hardware hash function. In this scenario, a user-space thread would be responsible for handling each ring pair, and the only time these threads must synchronize would be when updating the vAQM state table. To evaluate the effect of this synchronization on the latency of NetBump in a multi-threaded implementation, we examined the effect of vAQM table lock overhead. As a baseline, a single-threaded forwarding pipeline (FP) has a latency of 29.16μs. Running NetBump with two FPs (two ring pairs in the NIC and each FP running on its own core) increased that latency by 17.9% to 35.5μs. Further running NetBump with four FPs on

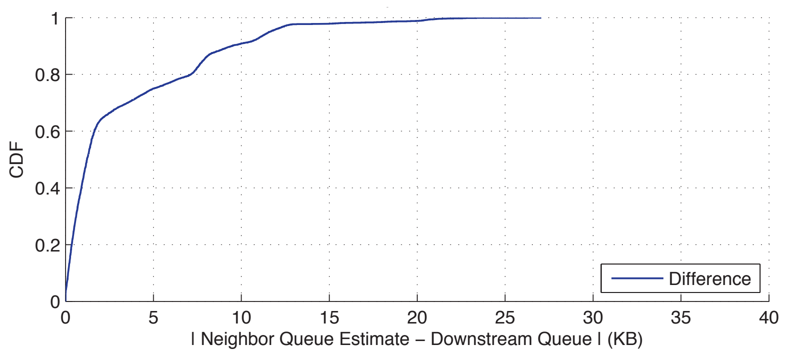


Figure 5.13: Typical distributed NetBump relative error with background elephant flows.

Table 5.3: Coding effort for NetBump and its applications.

Application	Lines of Code
NetBump core	940
RED	29
DCTCP	29
QCN	464

four cores increased the latency by an additional 1.95% to 36.8μs. Thus we find that the synchronization overhead is minimal to gain back a four-fold increase in computation per packet, or alternatively, a four-fold increase in supported line rate. A key observation is that NetBump avoids some of the required synchronization overheads found in software routers [31, 35, 90] with multiple ports, since in NetBump each input port only forwards to a single output port, preventing packets from spanning cores or causing contention on shared output ports.

5.5.3 Deployed Applications

One metric highlighting the ease of writing new applications with NetBump is shown in Table 5.3. Most of our applications took only 10s of lines of code, and QCN, which is much more complex, was written in less than 500 lines of code. The time commitment ranged from hours to a couple of days in the case of QCN. We now examine each application in detail.

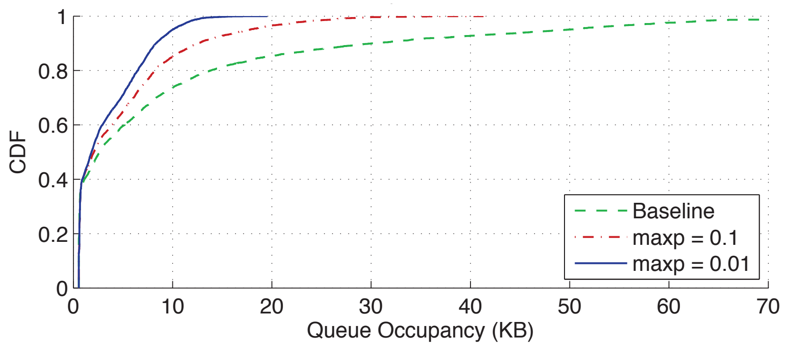


Figure 5.14: Evaluation of RED with two max_p parameter settings, showing the effect of reduced buffering given a higher marking probability.

Random Early Detection

Fig. 5.14 shows how deploying RED lowers the downstream buffer occupancy for a set of flows. Here three alternatives are compared: RED with two max_p parameter values, as well as the baseline drop-tail queuing discipline. One observation was that experimenting with different RED parameters was very easy with NetBump, and thus we could rapidly explore its parameter space by simply providing different arguments to our NetBump application's command line.

Data Center TCP

The next experiment represents a recreation of the DCTCP convergence test presented by Alizadeh et al. [5] performed in our two-rack testbed (see Fig. 5.8). Five source nodes each open a TCP connection to one of five destination nodes in 25 second intervals. In the baseline TCP case (Fig. 5.16(a)), due to buffer pressure and a drop-tail queuing discipline, the bandwidth is shared unfairly, resulting in a wide oscillation of throughput and unfair sending rate among the flows. Fig. 5.16(b) shows the throughput of DCTCP-enabled endpoints and a DCTCP vAQM strategy in the NetBump. Like in the original DCTCP work, here the fair sharing of network bandwidth results from the lower queue utilization afforded by senders appropriately backing off in response to NetBump-set ECN signals.

Another contribution of reduced queue buildup is better support for mixtures of latency-sensitive and long-lived flows. Fig. 5.15 shows the CDF of response time for

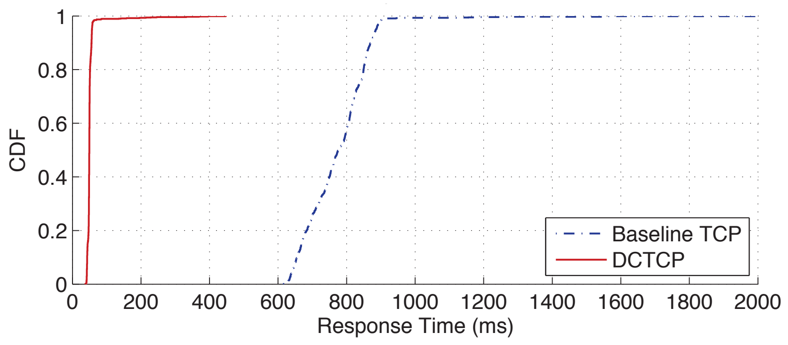


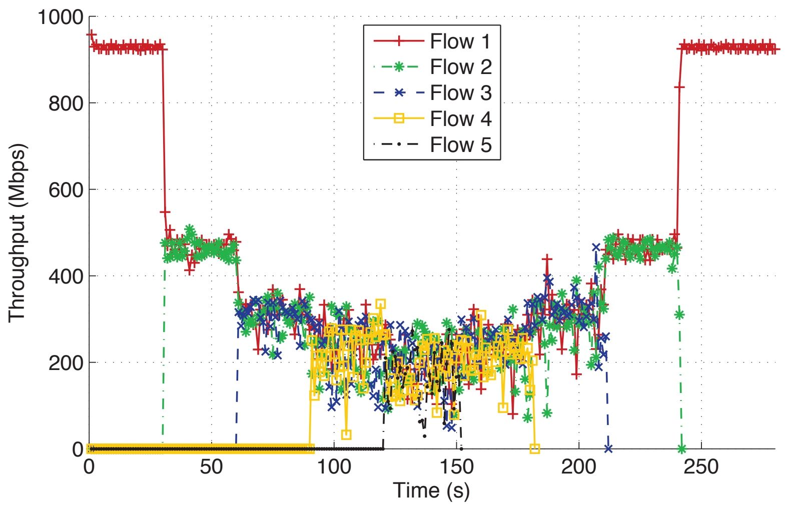
Figure 5.15: Baseline TCP (CUBIC) and DCTCP response times for short RPC-type flows in the presence of background elephant flows.

10,000 RPC-type requests in the presence of two large elephant flows, comparing stock TCP endpoints without NetBump DCTCP support. This figure recreates a key DCTCP result: signaling the long flows to reduce their rates results in smaller queues, lower RTT, and in the end, shorter response times.

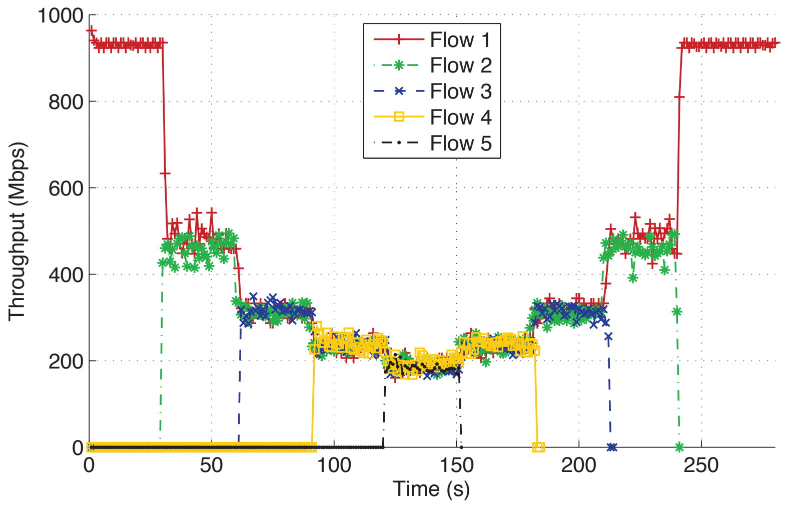
Quantized Congestion Notification

Another example of how the NetBump programming model enabled easy and rapid prototyping and evaluation of new protocols was deploying 802.1Qau-QCN. Our implementation of QCN is 464 lines of code, and it took around 2-3 days to write and debug. Developing QCN within NetBump enabled us to easily tune parameters and evaluate their effect. This was especially important given QCN's novelty, and the lack of other tools or simulations we could have used to study it. Using the testbed topology of Fig. 5.8, we use NetBump0 as the CP, and NetBump1 as the RP. In our RP, we chose a virtual queue size of 100KB (and Q_{eq} at 20KB).

Through our evaluation, we found that the feedback control loop tends to be more stable when the frequency of feedback messages is higher and their effect smaller. For this reason, we use $F_{bmax} = 32$, and plateau the probability profile at 20%. We also found that due to the burstiness of the packet arrival rate, we had to decrease w to 1 to avoid unnecessary rate drops. Our implementation also needed to consider the relative flow weights in the entire queue when choosing which flow to rate limit, rather than just using the current packet. We use the byte counter-only model of RP in our implementation.



(a) Baseline TCP (CUBIC)



(b) DCTCP

Figure 5.16: The effect on fairness and convergence of DCTCP on five flows sharing a bottleneck link.

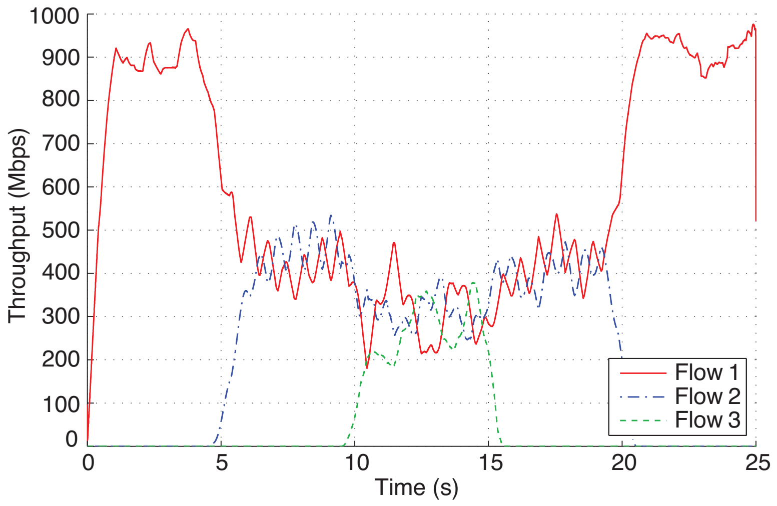


Figure 5.17: QCN with three 1Gbps UDP flows. With QCN enabled, the RP virtual queue occupancy never exceeded 40%, as opposed to persistent drops downstream without.

For the Additive Increase phase, we use cycles of 100 packets, and an increase of 1.5Mbps (to exaggerate and show the convergence of the virtual port current rates), and 600 packet cycles for the Fast Recovery phase.

The throughput of three 1Gbps UDP flows sharing the same bottleneck link is shown in Fig. 5.17. Without QCN, the downstream buffer would be persistently overwhelmed by the three UDP flows from 5-20s, but with QCN enabled, congestion is pushed upstream and the virtual queue occupancy never exceeded 40%, thereby preventing drops for potential mice flows.

5.5.4 Evaluation Summary

In this section we have described several vAQM and congestion control applications built with NetBump. We found that even though some had been extensively studied in the literature (e.g. RED), finding the particular parameters to make them work well in our network required several attempts. NetBump simplified this design–deploy–evaluate loop. Furthermore, in the case of QCN, there was little experience available due to its novelty and lack of deployments. The ability to develop our implementation in software, while testing it with real traffic, proved extremely useful.

5.6 Acknowledgement

Chapter 5, in part, contains material submitted for publication. Al-Fares, Mohammad; Porter, George; Kapoor, Rishi; Das, Sambit; Weatherspoon, Hakim; Prabhakar, Balaji; Vahdat, Amin. The dissertation author was the lead graduate student on this paper.

Chapter 6

Conclusions and Future Work

We here provide a summary of the contributions of this dissertation and a few concluding remarks, and finally give some important directions for future research beyond this work.

6.1 Summary

In chapter 1, we set out to prove the following: That by combining commodity, off-the-shelf networking equipment with the power of network programmability using open standards, it is possible to overcome all of the following three challenges:

1. Building scalable data center topologies. Specifically, those that can accommodate tens of thousands of servers, and support all-to-all communication at the full speed of the edge.
2. Overcoming forwarding scalability limitations. Namely, the ability to actually achieve the full bisection bandwidth possible for multipath topologies.
3. Support continually and rapidly extensible networking functionality.

We here summarize how we tackled each of these goals. First, in chapter 3, we discussed how bandwidth is increasingly becoming major a scalability bottleneck in large-scale clusters. Existing solutions for addressing this bottleneck center around hierarchies of switches, with expensive, non-commodity switches at the top of the hierarchy. At any given point in time, the port density of high-end switches limits overall cluster size while at the same time incurring tremendous cost. To overcome this limitation, we presented a data center communication architecture that leverages commodity Ethernet switches to deliver scalable bandwidth for large-scale clusters. We base our topology around the fat-tree and then present techniques to perform switch-local load balancing while remaining backward compatible with Ethernet, IP, and TCP.

Overall, we find that we are able to deliver scalable bandwidth at a fraction of the per-port cost of existing techniques. We believe that larger numbers of commodity switches have the potential to displace high-end switches in data centers in the same way that clusters of commodity PCs have displaced supercomputers for high-end computing environments.

Next, in chapter 4, we address the issue of forwarding adaptability to dynamic traffic patterns. The most important finding of our work is that in the pursuit of efficient use of multipath network topologies, a central scheduler with global knowledge of active flows can significantly outperform the hash-based ECMP load-balancing method that is currently deployed in commercial switches. We limit the overhead of our approach by focusing our scheduling decisions on the large flows responsible for much of the bytes sent across the network. We find that Hedera's performance gains are dependent on the rates and durations of the flows in the network; the benefits are more evident when the network is stressed with many large data transfers both within pods and across the diameter of the network.

We have demonstrated the feasibility of building a working prototype of our scheduling system for multi-rooted trees, and have shown that Simulated Annealing almost always outperforms Global First Fit and is capable of delivering near-optimal bisection bandwidth for a wide range of communication patterns both in our physical testbed and in simulated data center networks consisting of thousands of nodes. Given the low computational and latency overheads of our flow placement algorithms, the large investment in network infrastructure associated with data centers (many millions of dollars), and the incremental cost of Hedera's deployment (e.g., one or two servers), we show that dynamic flow scheduling has the potential to deliver substantial bandwidth gains with moderate additional cost.

Finally, in chapter 5, we presented NetBump, a platform for developing, experimenting with, and deploying alternative packet buffering and queuing disciplines with minimal intrusiveness and at low latency. NetBump leaves existing switches and end-hosts unmodified. It acts as a "bump on the wire", examining, optionally modifying, and forwarding packets at line rate in tens of microseconds to implement a variety of virtual active queuing disciplines and congestion control protocols implemented in user-space. We built and deployed several applications with NetBump, including DCTCP and 802.1Qau-QCN. These applications were quickly developed in hours or days, and required only tens or hundreds of lines of code in total.

A major barrier to developing and deploying new network functionality is the difficulty of programming the network datapath. In this work we evaluate Net-Bump as deployed on a variety of software-programmable systems, including Linux and Click/RouteBricks, and a user-level, kernel-bypass networking API. We found this latter implementation choice, despite being the oldest, provided the lowest-latency performance, supporting line-rate forwarding of minimum-sized packets at 9.5Gbps across each of these applications. The adoption of multi-core processors, along with kernel-bypass commodity NICs, provides a feasible platform to deploy data modifications written in user-space at line rate. Our experience has shown that NetBump is a useful and practical platform for prototyping and deploying new network functionality in real data center environments.

In conclusion, and revisiting the hypothesis statement, we have shown how all constituent projects of this dissertation leveraged commodity components and open standards to achieve their respective goals. Furthermore, these projects were highly congruent, in spirit, with the philosophy of Software-Defined Networking (SDN); specifically as it relates to (1) providing a clear networking API to enable experimentation and deployment of new protocols (e.g. NetBump), and (2) exposing network events and management to applications (e.g. flow scheduling with Hedera). This research certainly would not have been possible without this opening up of monolithic, vertically-integrated commercial switches into open, general purpose forwarding machines with accessible, centralized control. Going forward into the next decade, we strongly believe in this model's disruptive and transformative power over the networking world in general.

6.2 Future Work

The data center networking field is in its infancy and rapidly evolving, from an operations engineering perspective in general, and a network interconnect design in particular. Even with the deluge of data center networking research and innovation by academia, small start-ups, and big industry players in the past decade, we feel that the community has barely scratched the surface, and that there is an almost infinite amount of work to be done.

We here mention some exciting related on-going research and a few open problems. This list is by no means exhaustive.

6.2.1 Network Operating System

The network operating system concept envisioned by software-defined networking stems from a great body of previous work [17, 18, 19, 50], and has seen great strides in recent projects like NOX [53] and Onix [78]. While these last two systems are advanced control platforms currently in use in production networks of thousands of hosts, this field is evolving and there is still significant work to be done. Onix, for example, currently only allows one running management application at any given time. To support an analog of the multiprogramming model from ancient systems like the IBM S/360 [7] and MULTICS [110], several problems such as protection and concurrent resource management would have to be solved in the networking domain.

One specific example where such a network operating system would affect and drastically improve the performance of Hedera is the large-flow notification mechanism. Instead of Hedera polling the edge switches for flow statistics and performing the flow detection (possibly redundant functionality re-implemented unnecessarily by other network applications), the OS would provide these triggers explicitly, thereby reducing the detection latency overhead. The OS would also handle the requisite flow table entry insertion once paths are computed. On the whole, the possibilities for network management application composition are endless.

6.2.2 Cabling Complexity

Naturally, one of the major drawbacks of using fat-tree topologies is the large overhead of cabling involved. For a k-port fat-tree, there are $(3/4) * k^3$ cables to install and debug. To overcome this limitation, Farrington et al. [40] suggest leveraging commodity merchant silicon switching chips to package such a fat-tree topology onto a distributed multi-stage switch. Such an approach would offer great savings in overall cost, rack space, and cabling effort.

Another issue is that of incremental deployment. As an operator builds pods incrementally as the data center expands, the operator is faced with two options: either have the core switching layer fully populated with switches (and have a fraction of ports empty in the interim), or build the core layer incrementally as well (and multiplex cores accordingly). While the latter is more efficient in terms of core layer utilization, it does require re-wiring the core layer whenever new pods are added.

A compromise between these two extremes would be to use an intelligent patch panel (e.g. using an optical circuit switch), that would allow pods and core switches to be plugged in once, and would reconfigure the pod-core wiring scheme whenever topology changes are needed. Implementing the glue logic behind this mechanism would be fairly straightforward, and would simplify the incremental deployment scenario for multi-rooted tree topologies in general.

6.2.3 Hybrid Interconnects

Several recent research efforts have proposed augmenting the electrical switches in the data center with either optical switches (e.g. Helios [39]), or with short-range wireless links (e.g. flyways [56]). At a high-level, these approaches supplement an oversubscribed core network with dynamically added connectivity between pods with sufficiently stable traffic demands. These approaches have the benefit of lowering cost, power, and cabling complexity.

Besides augmenting traditional electrical switches, optical circuit switches in the data center have another exciting application. When used as an patch-panel onto which all pod and core switches are connected, optical switches can be used to dynamically alter the physical topology of the network on-the-fly to implement dynamically-defined topologies with different levels of oversubscription. This could be based on communication demands, for example, or as a mechanism for physical isolation in multi-tenant environments over time. This would have the potential to deliver high-bandwidth dynamically where and when needed, and in situations where full 1:1 bisection may not be required at all times.

6.2.4 Balanced Systems

Efficient network utilization is only one piece of the overall data center utilization puzzle. While the challenge of building large balanced systems is a vast field of research, there remains an important challenge of system provisioning (in terms of CPU, memory, I/O bandwidth, disk, etc.) in the data center. Since these clusters serve a rich mix of applications and tenants, what server configuration would be considered acceptable, in terms of performance and cost, for a given set of software requirements and latency deadlines?

The current state-of-the-art is to simply follow various system design rules of thumb accumulated over decades, as well as testing sample configurations' performance in small deployments. The BICMIC [120] and *scc* [89] projects take a more disciplined approach by using resource characterization, modeling the intended workload, and a required SLA contract to find the most cost-effective cluster design.

An example of this research would be the TritonSort project [106], which attempted to architect a sorting system and provision computing nodes such that overall utilization of system resources is maximized. To show the potential impact of this effort, TritonSort achieved a six-fold improvement in per-node efficiency over the previous sorting record holder.

These efforts show that there is substantial room for improvement in terms of balanced software/hardware system design, and that much work needs to be done to bring those gains to general distributed computing frameworks like MapReduce/Hadoop.

6.3 Acknowledgement

Chapter 6, in part, contains material as it appears in the Proceedings of the ACM Conference on Data Communication (SIGCOMM) 2008. Al-Fares, Mohammad; Loukissas, Alexander; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, contains material as it appears in the Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI) 2010. Al-Fares, Mohammad; Radhakrishnan, Sivasankar; Raghavan, Barath; Huang, Nelson; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, contains material submitted for publication. Al-Fares, Mohammad; Porter, George; Kapoor, Rishi; Das, Sambit; Weatherspoon, Hakim; Prabhakar, Balaji; Vahdat, Amin. The dissertation author was the lead graduate student on this paper.

Bibliography

[1] 10G QCN Implementation on Hardware. http://fif.kr/AsiaNetFPGAws/slide/1-4_slide.pdf, 2010.

[2] Al-Fares, M., Loukissas, A., and Vahdat, A. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of ACM SIGCOMM* (2008).

[3] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of USENIX NSDI* (2010).

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

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

[6] Amazon EC2. http://aws.amazon.com/ec2/, 2011.

[7] Amdahl, G. M., Blaauw, G. A., and Brooks, Jr., F. P. Architecture of the IBM System/360. *IBM Journal of Research and Development 44*, 1 (2000), 21–36.

[8] Announcing the Windows Azure HPC Scheduler and HPC Pack 2008 R2 Service Pack 3 releases. http://blogs.technet.com/b/windowshpc/archive/2011/11/11/hpc-pack-2008-r2-sp3-and-windows-azure-hpc-scheduler-released.aspx, 2011.

[9] Anwer, M. B., Motiwala, M., Tariq, M. b., and Feamster, N. SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. In *Proceedings of ACM SIGCOMM* (2010).

[10] Apache Hadoop Project. http://hadoop.apache.org/, 2008.

[11] Aweya, J., Ouellette, M., Montuno, D. Y., and Felske, K. Rate-based proportional-integral control scheme for active queue management. *Int. J. Netw. Manag. 16* (May 2006), 203–231.

[12] Benson, T., Anand, A., Akella, A., and Zhang, M. Understanding Data Center Traffic Characteristics. In *Proceedings of ACM WREN* (2009).

[13] Blumrich, M., Chen, D., Coteus, P., Gara, A., Giampapa, M., Heidelberger, P., Singh, S., Steinmacher-Burow, B., Takken, T., and Vranas, P. Design and Analysis of the BlueGene/L Torus Interconnection Network. *IBM Research Report RC23025 (W0312-022) 3* (2003).

[14] Boden, N., Cohen, D., Felderman, R., Kulawik, A., Seitz, C., and Seizovic, J. Myrinet: A Gigabit-per-second Local Area Network. *Micro, IEEE 15*, 1 (1995).

[15] Brin, S., and Page, L. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of WWW7* (1998).

[16] Buonadonna, P., Geweke, A., and Culler, D. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of ACM/IEEE CDROM* (1998).

[17] Caesar, M., Caldwell, D., Feamster, N., Rexford, J., Shaikh, A., and van der Merwe, J. Design and Implementation of a Routing Control Platform. In *Proceedings of USENIX NSDI* (2005).

[18] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: Taking Control of the Enterprise. In *Proceedings of ACM SIGCOMM* (2007).

[19] Casado, M., Garfinkel, T., Akella, A., Freedman, M. J., Boneh, D., McKeown, N., and Shenker, S. SANE: A Protection Architecture for Enterprise Networks. In *Proceedings USENIX Security* (2006).

[20] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of USENIX OSDI* (2006).

[21] Chelsio Network Interface. http://www.chelsio.com, 2011.

[22] Chen, Y., Griffith, R., Liu, J., Katz, R. H., and Joseph, A. D. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proceedings of ACM WREN* (2009).

[23] Cheveresan, R., Ramsay, M., Feucht, C., and Sharapov, I. Characteristics of Workloads used in High Performance and Technical Computing. In *Proceedings of ICS* (2007).

[24] Chisvin, L., and Duckworth, R. J. Content-Addressable and Associative Memory: Alternatives to the Ubiquitous RAM. *Computer 22*, 7 (1989), 51–64.

[25] Cisco and OpenFlow. http://blogs.cisco.com/datacenter/whats-new-with-cisco-and-openflow/, 2011.

[26] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf, 2008.

[27] Claise, B. Cisco Systems NetFlow Services Export Version 9. RFC 3954, Internet Engineering Task Force, 2004.

[28] Clos, C. A Study of Non-blocking Switching Networks. *Bell System Technical Journal 32*, 2 (1953).

[29] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of USENIX OSDI* (2004).

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

[31] Dobrescu, M., Argyraki, K., Iannaccone, G., Manesh, M., and Ratnasamy, S. Controlling Parallelism in a Multicore Software Router. In *Proceedings of ACM Presto* (2010).

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

[33] Downey, A. B. Evidence for Long-tailed Distributions in the Internet. In *Proceedings of ACM SIGCOMM IMW* (2001).

[34] Eatherton, W., Varghese, G., and Dittia, Z. Tree Bitmap : Hardware/Software IP Lookups with Incremental Updates. *ACM SIGCOMM CCR 34*, 2 (2004), 97–122.

[35] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., Mathy, L., and Papadimitriou, P. Forwarding Path Architectures for Multicore Software Routers. In *Proceedings of ACM Presto* (2010).

[36] Elwalid, A., Jin, C., Low, S., and Widjaja, I. MATE: MPLS Adaptive Traffic Engineering. *Proceedings of IEEE INFOCOM* (2001).

[37] Ely, D., Savage, S., and Wetherall, D. Alpine: A User-level Infrastructure for Network Protocol Development. In *Proceedings of USITS* (2001).

[38] Even, S., Itai, A., and Shamir, A. On the Complexity of Timetable and Multicommodity Flow Problems. *SIAM Journal on Computing 5*, 4 (1976), 691–703.

[39] Farrington, N., Porter, G., Radhakrishnan, S., Bazzaz, H. H., Subramanya, V., Fainman, Y., Papen, G., and Vahdat, A. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proceedings of ACM SIGCOMM* (2010).

[40] Farrington, N., Rubow, E., and Vahdat, A. Data Center Switch Architecture in the Age of Merchant Silicon. In *Proceedings of IEEE Hot Interconnects* (2009).

[41] Fleischer, M. Simulated Annealing: Past, Present, and Future. In *Proceedings of IEEE WSC* (1995).

[42] Floyd, S. TCP and Explicit Congestion Notification. *ACM SIGCOMM CCR 24* (October 1994), 8–23.

[43] Floyd, S., and Jacobson, V. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM ToN 1* (August 1993), 397–413.

[44] Fred, S. B., Bonald, T., Proutiere, A., Régnié, G., and Roberts, J. W. Statistical Bandwidth Sharing: A Study of Congestion at Flow Level. *ACM SIGCOMM CCR* (2001).

[45] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[46] Geoffray, P., and Hoefler, T. Adaptive Routing Strategies for Modern High Performance Networks. In *Proceedings of IEEE Hot Interconnects* (2008).

[47] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. *ACM SIGOPS Operating Systems Review 37*, 5 (2003).

[48] Gibbens, R. J., and Kelly, F. Distributed Connection Acceptance Control for a Connectionless Network. In *Teletraffic Engineering in a Competitive World* (1999), Elsevier.

[49] Gibbens, R. J., and Kelly, F. Resource Pricing and the Evolution of Congestion Control. In *Automatica 35* (1999).

[50] Greenberg, A., Hjalmtysson, G., Maltz, D. A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR* (2005).

[51] Greenberg, A., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D., Patel, P., and Sengupta, S. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of ACM SIGCOMM* (2009).

[52] Greenberg, A., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *Proceedings of ACM PRESTO* (2008).

[53] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. NOX: Towards an Operating System for Networks. *ACM SIGCOMM CCR* (2008).

[54] Guo, C., Lu, G., Li, D., Wu, H., Zhang, X., Shi, Y., Tian, C., Zhang, Y., and Lu, S. BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers. In *Proceedings of ACM SIGCOMM* (2009).

[55] Guo, C., Wu, H., Tan, K., Shi, L., Zhang, Y., and Lu, S. DCell: A Scalable and Fault-Tolerant Network Structure for Data Centers. In *Proceedings of ACM SIGCOMM* (2008).

[56] Halperin, D., Kandula, S., Padhye, J., Bahl, P., and Wetherall, D. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *Proceedings of ACM SIGCOMM* (2011).

[57] Heller, B., Seetharaman, S., Mahadevan, P., Yiakoumis, Y., Sharma, P., Banerjee, S., and McKeown, N. ElasticTree: Saving Energy in Data Center Networks. In *Proceedings of USENIX NSDI* (2010).

[58] Hollot, C., Misra, V., Towsley, D., and Gong, W.-B. A Control Theoretic Analysis of RED. In *Proceedings of IEEE INFOCOM* (2001).

[59] Hollot, C., Misra, V., Towsley, D., and Gong, W.-B. On Designing Improved Controllers for AQM Routers Supporting TCP Flows. In *Proceedings of IEEE INFOCOM* (2001).

[60] Holmberg, K. Optimization Models for Routing in Switching Networks of Clos Type with Many Stages. *Advanced Modeling and Optimization 10*, 1 (2008).

[61] Hölzle, U., and Barroso, L. A. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. Morgan and Claypool Publishers, 2009.

[62] Hopps, C. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, Internet Engineering Task Force, 2000.

[63] IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Force. http://www.ieee802.org/3/ba/, 2010.

[64] InfiniBand Architecture Specification Volume 1, Release 1.0. http://www.infinibandta.org/specs, 2001.

[65] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of ACM EuroSys* (2007).

[66] Juniper J-Flow. http://www.juniper.net/techpubs/software/erx/junose61/swconfig-routing-vol1/html/ip-jflow-stats-config2.html, 2008.

[67] Kabbani, A., and Alizadeh, M. Personal communication, 2011.

[68] Kabbani, A., Alizadeh, M., Yasuda, M., Pan, R., and Prabhakar, B. AF-QCN: Approximate Fairness with Quantized Congestion Notification for Multi-tenanted Data Centers. In *Proceedings of IEEE Hot Interconnects* (2010).

[69] Kandula, S., Katabi, D., Davie, B., and Charny, A. Walking the Tightrope: Responsive yet Stable Traffic Engineering. In *Proceedings of ACM SIGCOMM* (2005).

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

[71] Karandikar, S., Kalyanaraman, S., Bagal, P., and Packer, B. TCP Rate Control. In *Proceedings of ACM SIGCOMM* (2000).

[72] Katabi, D., Handley, M., and Rohrs, C. Congestion Control for High Bandwidth-delay Product Networks. In *Proceedings of ACM SIGCOMM* (2002).

[73] Katz, D. Ward, D. BFD for IPv4 and IPv6 (Single Hop) (Draft). Tech. rep., Internet Engineering Task Force, 2008.

[74] Keromytis, A. D., and Wright, J. L. Transparent network security policy enforcement. In *Proceedings of USENIX ATC* (2000).

[75] Kim, C., Caesar, M., and Rexford, J. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proceedings of ACM SIGCOMM* (2008).

[76] Kohler, E. Personal communication, 2011.

[77] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. *ACM ToCS 18* (August 2000), 263–297.

[78] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., and Shenker, S. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of USENIX OSDI* (2010).

[79] Kunniyur, S., and Srikant, R. An Adaptive Virtual Queue (AVQ) Algorithm for Active Queue Management. *IEEE/ACM ToN 12* (April 2004), 286–299.

[80] Kuzmanovic, A. The Power of Explicit Congestion Notification. In *Proceedings of ACM SIGCOMM* (2005).

[81] Lantz, B., Heller, B., and McKeown, N. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of ACM Hotnets* (2010).

[82] Leiserson, C., Abuhamdeh, Z., Douglas, D., Feynman, C., Ganmukhi, M., Hill, J., Hillis, D., Kuszmaul, B., Pierre, M., Wells, D., et al. The Network Architecture of the Connection Machine CM-5 (Extended Abstract). In *Proceedings of ACM SPAA* (1992).

[83] Leiserson, C. E. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers 34*, 10 (1985), 892–901.

[84] Linux Traffic Control howto. http://tldp.org/HOWTO/Traffic-Control-HOWTO/, 2011.

[85] Liu, J., Huang, W., Abali, B., and Panda, D. High Performance VMM-Bypass I/O in Virtual Machines. In *Proceedings of USENIX ATC* (2006).

[86] Lockwood, J., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., and Luo, J. NetFPGA–An Open Platform for Gigabit-rate Network Switching and Routing. In *Proceedings of IEEE MSE* (2007).

[87] Lu, G., Guo, C., Li, Y., Zhou, Z., Yuan, T., Wu, H., Xiong, Y., Gao, R., and Zhang, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of USENIX NSDI* (2011).

[88] Lu, Y., Pan, R., Prabhakar, B., Bergamasco, D., Alaria, V., and Baldini, A. Congestion Control in Networks with no Congestion Drops. In *Proceedings of Allerton CCC* (2006).

[89] Madhyastha, H. V., McCullough, J. C., Porter, G., Kapoor, R., Savage, S., Snoeren, A. C., and Vahdat, A. scc: Cluster Storage Provisioning Informed by Application Characteristics and SLAs. In *Proceedings of USENIX FAST* (2012).

[90] Manesh, M., Argyraki, K., Dobrescu, M., Egi, N., Fall, K., Iannaccone, G., Kohler, E., and Ratnasamy, S. Evaluating the Suitability of Server Network Cards for Software Routers. In *Proceedings of ACM PRESTO* (2010).

[91] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR* (2008).

[92] Misra, V., Gong, W.-B., and Towsley, D. Fluid-based Analysis of a Network of AQM Routers Supporting TCP Flows with an Application to RED. In *Proceedings of ACM SIGCOMM* (2000).

[93] Miura, S., Boku, T., Okamoto, T., and Hanawa, T. A Dynamic Routing Control System for High-Performance PC Cluster with Multi-path Ethernet Connection. In *Proceedings of IEEE IPDPS* (2008).

[94] Moy, J. OSPF Version 2. RFC 2328, Internet Engineering Task Force, 1998.

[95] Mudigonda, J., Yalagandula, P., Al-Fares, M., and Mogul, J. C. SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In *Proceedings USENIX NSDI* (2010).

[96] Myricom Sniffer10G. http://www.myricom.com/support/downloads/sniffer.html, 2011.

[97] Mysore, R. N., Pamporis, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Subramanya, V., and Vahdat, A. PortLand: A Scalable, Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proceedings of ACM SIGCOMM* (2009).

[98] NEC/Stanford: 10G QCN Implementation on Hardware. http://www.ieee802.org/1/files/public/docs2009/au-yasuda-10G-QCN-Implementation-1109.pdf, 2009.

[99] Oki, E., Jing, Z., Rojas-Cessa, R., and Chao, H. J. Concurrent Round-robin-based Dispatching Schemes for Clos-network Switches. *IEEE/ACM ToN 10* (December 2002), 830–844.

[100] OMNeT++ Simulator. http://www.omnetpp.org/, 2008.

[101] Open Networking Foundation. https://www.opennetworking.org/, 2011.

[102] OpenOnload. http://www.openonload.org/, 2011.

[103] Padhye, J., Firoiu, V., Towsley, D., and Kurose, J. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proceedings of ACM SIGCOMM* (1998).

[104] PF_RING Direct NIC Access. http://www.ntop.org/products/pf_ring/dna/, 2011.

[105] Raiciu, C., Pluntke, C., Barre, S., Greenhalgh, A., Wischik, D., and Handley, M. Data Center Networking with Multipath TCP. In *Proceedings of ACM Hotnets* (2010).

[106] Rasmussen, A., Porter, G., Conley, M., Madhyastha, H. V., Mysore, R. N., Pucher, A., and Vahdat, A. TritonSort: A Balanced Large-scale Sorting System. In *Proceedings of USENIX NSDI* (2011).

[107] RED: Discussions of Setting Parameters. http://icir.org/floyd/REDparameters.txt, 1997.

[108] Rizzo, L., and Landi, M. netmap: Memory Mapped Access to Network Devices. In *Proceedings of ACM SIGCOMM* (2011).

[109] Rubow, E., McGeer, R., Mogul, J., and Vahdat, A. Chimpp: A Click-based Programming and Simulation Environment for Reconfigurable Networking Hardware. In *Proceedings of ACM/IEEE ANCS* (2010).

[110] Saltzer, J. H. Protection and the Control of Information Sharing in MULTICS. *Commun. ACM 17* (July 1974), 388–402.

[111] Schmuck, F., and Haskin, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of USENIX FAST* (2002).

[112] Scott, L. R., Clark, T., and Bagheri, B. *Scientific Parallel Computing*. Princeton University Press, 2005.

[113] Shah, N. Understanding Network Processors. Master's thesis, University of California, Berkeley, Calif., 2001.

[114] Sharma, S., Gopalan, K., Nanda, S., and Chiueh, T. Viking: A Multi-spanning-tree Ethernet Architecture for Metropolitan Area and Cluster Networks. In *Proceedings of IEEE INFOCOM* (2004).

[115] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. Can the Production Network be the Testbed? In *Proceedings of USENIX OSDI* (2010).

[116] Shieh, A., Kandula, S., and Sirer, E. G. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *Proceedings of ACM Hotnets* (2010).

[117] Sinha, S., Kandula, S., and Katabi, D. Harnessing TCPs Burstiness using Flowlet Switching. In *Proceedings of ACM HotNets* (2004).

[118] SMC SMC10GPCIe-10BT Network Adapter. http://www.smc.com/files/AY/DS_SMC10GPCIe-10BT.pdf, 2011.

[119] Smiljanić, A. Rate and Delay Guarantees Provided by Clos Packet Switches With Load Balancing. *IEEE/ACM ToN 16*, 1 (February 2008), 170 –181.

[120] Snoeren, A. C., Madhyastha, H., McCullough, J., Kapoor, R., Porter, G., Savage, S., and Vahdat, A. BICMIC: Matching Cluster Configurations to Application Demands. Presented at CloudS, 2010.

[121] SolarFlare Solarstorm Network Adapters. http://www.solarflare.com/Enterprise-10GbE-Adapters, 2011.

[122] Srinivasan, V., and Varghese, G. Faster IP Lookups using Controlled Prefix Expansion. *ACM ToCS 17*, 1 (1999), 1–40.

[123] Sun Datacenter Switch 3456 Architecture White Paper. http://www.sun.com/products/networking/datacenter/ds3456/ds3456_wp.pdf, 2008.

[124] Thaler, D., and Hopps, C. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, Internet Engineering Task Force, 2000.

[125] Top 500 Supercomputing Sites. http://www.top500.org/, 2011.

[126] TR10: Software-Defined Networking. http://www.technologyreview.com/biotech/22120/, 2009.

[127] Tucker, L., and Robertson, G. Architecture and Applications of the Connection Machine. *Computer 21*, 8 (1988).

[128] Turner, J. New directions in communications (or which way to the information age?). *Communications Magazine, IEEE* (2002).

[129] Turner, J. S. An Optimal Nonblocking Multicast Virtual Circuit Switch. In *Proceedings of IEEE INFOCOM* (1994).

[130] Valiant, L. G., and Brebner, G. J. Universal Schemes for Parallel Communication. In *Proceedings of ACM STOC* (1981).

[131] Vetter, J., Alam, S., Dunigan, T.H., J., Fahey, M., Roth, P., and Worley, P. Early Evaluation of the Cray XT3. In *Proceedings of IEEE IPDPS* (2006).

[132] von Eicken, T., Basu, A., Buch, V., and Vogels, W. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of ACM SOSP* (1995).

[133] Wang, Z. *Internet QoS: Architectures and Mechanisms for Quality of Service*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[134] Welsh, M., Basu, A., and von Eicken, T. ATM and Fast Ethernet Network Interfaces for User-level Communication. In *Proceedings of IEEE HPCA* (1997).

[135] Woodacre, M., Robb, D., Roe, D., and Feind, K. The SGI Altix 3000 Global Shared-Memory Architecture. *SGI White Paper* (2003).

[136] World's Largest Data Center: 350 E. Cermak. http://www.datacenterknowledge.com/special-report-the-worlds-largest-data-centers/worlds-largest-data-center-350-e-cermak/, 2010.

[137] Wu, Q., Mampilly, D. J., and Wolf, T. Distributed Runtime Load-balancing for Software Routers on Homogeneous Many-core Processors. In *Proceedings of ACM Presto* (2010).

[138] XFS: A High-performance Journaling Filesystem. http://oss.sgi.com/projects/xfs/, 2008.

[139] Yan, H., Maltz, D. A., Ng, T. S. E., Gogineni, H., Zhang, H., and Cai, Z. Tesseract: A 4D Network Control Plane. In *Proceedings of USENIX NSDI* (2007).

[140] Yu, M., Rexford, J., Freedman, M. J., and Wang, J. Scalable Flow-based Networking with DIFANE. In *Proceedings of ACM SIGCOMM* (2010).