UNIVERSITY OF CALIFORNIA, SAN DIEGO

Enabling Fine-grained Network Flow Management in Data Center Networks and Servers

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

in

Computer Science

by

Malveeka Tewari

Committee in charge:

Professor George Porter, Co-Chair
Professor Amin Vahdat, Co-Chair
Professor Shayan Mookherjea
Professor Alex Snoeren
Professor Geoff Voelker

2015

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

Co-Chair

Co-Chair

University of California, San Diego

2015

DEDICATION

To my husband Vivek and our families.

EPIGRAPH

To follow knowledge like a sinking star,
Beyond the utmost bound of human thought.

Lord Alfred Tennyson

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

Looking back at my time spent in graduate school, I feel extremely honored to and blessed to have received an opportunity to work with some of the finest people I've even known, both technically and personally. I have learned a valuable lot during my PhD and have grown as a person and a researcher.

I would like to begin by thanking my advisors Professor Amin Vahdat and Professor George Porter. I am extremely grateful to have received an opportunity to work with them. Amin taught me the importance of thinking critically, working meticulously and not be afraid of failures. Despite moving to Google, he remained in constant touch, providing valuable feedback on the projects and kept us excited and motivated. It is beyond me how he handled two jobs at the same time, I can only aspire to be able to manage my time as efficiently as he does. Thank you Amin for being a constant source of inspiration! It was George's guidance and excitement that kept my graduate careeer afloat while Amin was on leave at Google. He patiently listened to my countless crazy idea, always encouraging me to think big. He taught me how to shape vague ideas into concrete research problems and build effective solutions. He has been a great influence in developing my writing and presentation skills. George your tenacity and dedication is extremely inspiring, thank you for your guidance and patience. I am deeply indebted to my advisors for their support.

I would like to thank the Sysnet faculty at UCSD, Stefan Savage for sharing the most interesting anecdotes and stories, Geoff Voelker for his kindness and for organizing the quarterly Sysnet hikes to ensure the laboring graduate students get to enjoy the San Diego sun, Alex Snoeren for changing the way I would look at graphs and tables for the rest of my life and Kirill for being the most interesting computer scientist. I am also thankful to Geoff, Alex and Shayan for serving on my committee and providing valuable feedback for my thesis.

I would have never applied for graduate school had I not interacted with the amazing folks at the Max Planck Institute of Software Systems, Saarbruecken. I am thankful to Peter Druschel, Krishna Gummadi and Alan Mislove for introducing me to the field of Systems and networking. During my short internship, I made found great friends in Anemome Michel and Ashutosh Gupta. I look back to the summer of 2008 with fondness.

I have been fortunate to have worked with the members of the DCSwitch group at UCSD. Harsha Madhyastha, Nathan Farrington, Radhika Niranjan, Alex Rasmussen, Mohammed Al Fares, Meg Walread-Sullivan were always there for feedback and advice as the more experienced and knowledgeable people in the group. Sivasankar Radhakrishnan, Rishi Kapoor, Mike Conley were the last inductees to the DCSwitch group and without their support, I would've truly felt alone in grad school.

I would like to thank Frank Uyeda, Gjergi Zyba, Bhanu Vattikonda, Arjun Roy, Louis DeKoven, Brown Farinholt, Zhaomo Yang, Vector Lee and Ian Foster who tolerated my (sometimes grumpy) presence as an occupant of room 3142. Thanks to Patrick Verkaik for teaching me how to operate the espresso beast in Chez Bob, Michael Vrable, Dimitar Bounov and Brown for running and keeping Chez Bob stocked for late night deadline crunches. Finally I would like to thank Karyn Benson, Valentine Robert and Neha Chachra for adding the social dimension to my otherwise mundane grad life.

Outside of graduate school, I am thankful to Swati Wagh for being the dependable and supportive roommate and living with me for the most part of my grad school. She and Shalmali Joshi were my first roommates when I moved to San Diego and am thankful for their friendship. Before I met folks in CSE, Aman Bhatia and Mandar Dixit were the only other new PhD students starting with me in 2009. I am thankful that Minu Jacob, Chetan Verma, Bharathan Balaji, Aman Khosa and Manish Gupta joined us in this journey. These people were my support system in San Diego in grad school. I would like to thank Alok Singh, Shweta Purawat, Ashish Cherukuri, Manasa Cherukuri, Sudeep Kamath, Ranjeet Kumar, Prarit Agarwal, Neeraja, Siddhesh for being a part of the fun potlucks and Ashish Tawari for keeping us entertained with his magic tricks.

I am extremely grateful to the Dubey family for making me feel at home and welcomed in their lives. My uncle Abhay Dubey, my aunt Smita Das Dubey and my cousins Mahika, Ambika and Akash for never letting me feel too homesick. I will remain indebted for their love and kindness.

I would like to thank my parents, Manoj and Seema, my sister Mansi, my brother-in-law Ravindra and my brother Manglam for patiently listening to the complaints about my life. My two year old nephew Yudhaan is the sunshine in my life. He teaches me to remain curious and innocent. Finally, I would like to thank my husband Vivek for his encouragement, support and affection. I dedicate this thesis to him and our families and look forward to our future together.

Chapter 3, in full, includes material as it appears in WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. Zhou, Junlan; Tewari, Malveeka; Zhu, Min; Kabbani, Abdul; Poutievski, Leon; Singh, Arjun; Vahdat, Amin. In Proceedings of the Ninth European Conference on Computer Systems (EuroSys), April, 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 4 and 5, in parts, include material that has been submitted for publication as Practical, Centralized Control of Programmable End-Host Data Planes. Tewari, Malveeka; Snoeren, Alex C.; Porter, George. The dissertation author was the primary investigator and author of this paper.

VITA

2009	B.E. (Hons.), Computer Science & Engineering Birla Institute of Technology & Science, Pilani
2011	M.S., Computer Science University of California, San Diego
2015	Ph.D., Computer Science University of California, San Diego

PUBLICATIONS

Tewari, Malveeka; Snoeren, Alex C.; Porter, George. “Practical, Centralized Control of Programmable End-Host Data Planes.” Under submission.

Zhou, Junlan; Tewari, Malveeka; Zhu, Min; Kabbani, Abdul; Poutievski, Leon; Singh, Arjun; Vahdat, Amin. “WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers,” In Proceedings of the Ninth European Conference on Computer Systems (EuroSys), April, 2014.

He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter, “Circuit Switching Under the Radar with REACToR,” In Proceedings of the 11th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), April 2014.

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

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

Meg Walraed-Sullivan, Radhika Niranjan Mysore, Malveeka Tewari, Ying Zhang, Keith Marzullo, and Amin Vahdat, “ALIAS: Scalable, Decentralized Label Assignment for Data Centers,”In Proceedings of the ACM Symposium on Cloud Computing (SoCC), October 2011.

Hamid Hajabdolali Bazzaz, Malveeka Tewari, Guohui Wang, George Porter, T. S. Eugene Ng, David G. Andersen, Michael Kaminsky, Michael A. Kozuch, and Amin Vahdat, “Switching the Optical Divide: Fundamental Challenges for Hybrid Electrical/Optical Datacenter Networks,” In Proceedings of the ACM Symposium on Cloud Computing (SoCC), October 2011.

ABSTRACT OF THE DISSERTATION

Enabling Fine-grained Network Flow Management in Data Center Networks and Servers

by

Malveeka Tewari

Doctor of Philosophy in Computer Science

University of California, San Diego, 2015

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

Modern data centers host hundreds of applications with hundreds of thousands of flows and varied traffic patterns. Efficient management of the data center flows is critical in order to ensure high utilization of the network and also achieving desired application performance. Numerous proposals aim to improve data center and enterprise networks through better management of network traffic. Examples include load balancing using Equal Cost Multipathing (ECMP), traffic engineering, precise per-flow rate limiting, scalable flow prioritization and classification, and support for microsecond-scale TDMA. These proposals rely on (1) support from the switching hardware for in-network flow management and/or (2) require the end host data plane being sufficiently performant and high precision, in order to work in coordination with network switches and centralized controllers.

While these flow management proposals meet the high resource utilization and performance requirement, often they rely on hardware modifications making them hard to deploy or cost-prohibitive, or, inextensible interfaces that make it hard for them to interoperate with the existing software stack in the end hosts. To address these challenges, we evaluate and propose readily deployable, cost-effective mechanisms for flow management both within the data center network fabric, leveraging commodity switch hardware and at the end host stack by leveraging the high performance packet processing libraries with better interfacing with the applications and the network. We propose Weighted Cost Multipathing (WCMP) as an in-network flow management framework that allows network operators to manage flows and load balance traffic with existing commodity switches. WCMP requires no hardware modification and allows fair bandwidth sharing between flows in presence of failures and asymmetry deliver high utilization and application performance. For end host based flow management, we evaluate the existing state-of-art mechanisms for flow management and leverage the high performance packet processing frameworks such as Dataplane development kit (DPDK) to propose a software defined dataplane (SDD) that allows the end host stack to interface with a remote controller and provide fine-grain control over packet transmissions and flow management.

Chapter 1

Introduction

1.1 Data Center Fabric

The recent years have observed an unprecedented increase in the deployment of data center networks all across the world. These data centers are large warehouses with hundreds of thousands of servers running a wide variety of applications. Figure 1.1 shows the key entities impacted by the increase in popularity of the data centers. In the next few sections of this chapter, we discuss how these different entities are the raison d'être for efficient flow management in the data centers and what are the requirements and challenges in implementing flow management across different layers within the data center.

- **Users:** For any system, the target audience or the users of the service and their requirements are key in shaping the design and management of the service. In the data center context, the target users are of the following kinds: (1) end users, surfing the web or requesting content like photos/videos, (2) network operators running services to measure revenue earned/management or diagnostics, or (3) other services that in turn interact with the users directly. These kind of users have different requirements from the underlying application and impose different restrictions on the application behavior and performance. Flow management in the

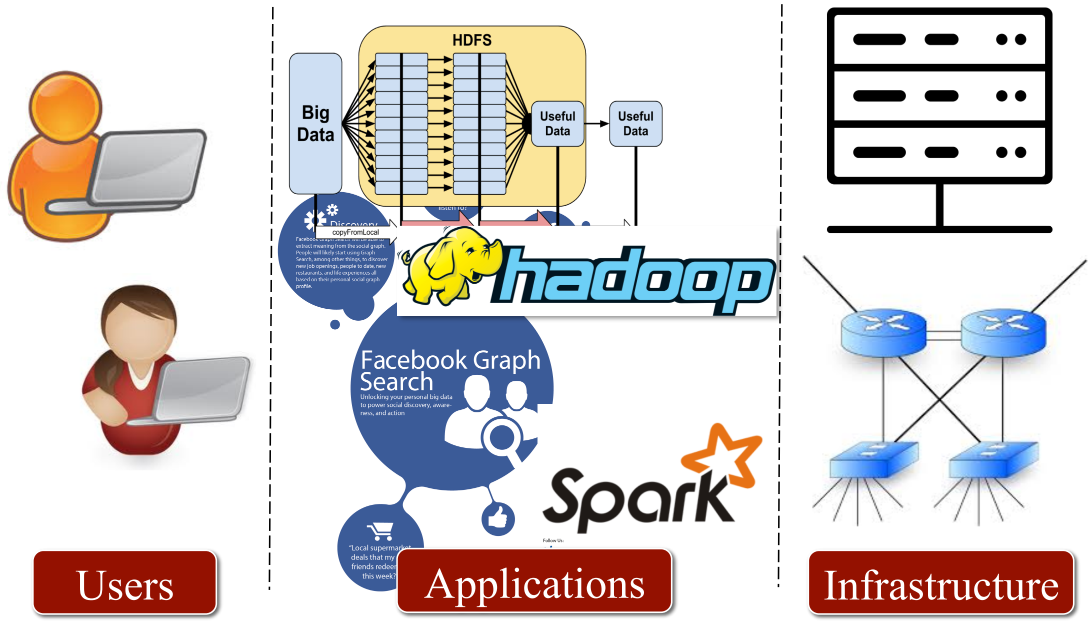


Figure 1.1. Users, applications and infrastructure in data centers

data center provides us with means to translate user expectations into requirements that determine how applications interact with users and among themselves in order to provide the best experience to the user.

- **Applications:** Modern data centers are home to a wide variety of applications including end-user facing applications such as search [35, 16], social networking [30, 79] as well as data intensive applications such as Hadoop [39], Spark [78]. These applications are in-turm made up of other applications such as key-value store [60] and have complex code-paths and logic running over a large number of servers that makes it hard to precisely model the performance of such applications. Finally, the fact that these services run on shared infrastructure further necessitates the need for an efficient flow management network that can allow different applications to utilize the shared infrastructure without hurting each other's performance.

- **Infrastructure:** The data center infrastructure consists of three main entities (1) compute (servers), (2) network (switches, routers, network links) and (3) storage (disk/flash drives). In this dissertation we focus on why efficient flow management is essential to ensure that the servers' compute cycles and the network bandwidth is utilized to maximum. We first look at network based flow management that aims at providing fairness and high throughput utilization for application flows. We then focus on the end hosts and understand how can the end hosts and network be better interfaced for supporting end host based flow management that makes it possible for applications to levarage the visibility into the network state and the network to have a better understanding to application behavior and use that information for better flow management in the data center.

1.2 Flow Management

Efficient flow management is crucial to meet the performance requirements of the varied data center applications and also for ensuring high network utilization and load balancing. We list down the key ideas that motivate the need for efficient and fine-grained flow management in the data centers.

- **Application performance:** Most data center applications run as distributed services running on multiple servers. These servers are constantly communicating within themselves as well as with other applications co-located in the data center to generate meaningful results. Depending on the kind of application, this communication results in hundreds of thousands of interdependent flows for e.g., partition-aggregate pattern [6]. In order to meet the desired application performance, it is necessary to ensure that the different flows receive the desired share of bandwidth without hurting the performance of other co-located applications and without incurring high latency overheads.

- **Quality of service:** Different data center applications have different requirements from the underlying network fabric. For e.g., key-value store based applications such as memcached [60] are latency sensitivie applications and require that the network routes the applications flows with the minimum amount of buffering or delays. On the contrary, applications such as Hadoop [39], Spark [78] are data intensive applications that impose high bandwidth requirements on the underlying network. Some applications such as video streaming services need minimum guaranteed bandwidth guarantees [88] in order to meet the desired service level agreeement (SLA). Given that these applications use the network as a shared resource, it becomes important to efficiently manage the different flows in the network to meet the wide range of application requirements.

- **Fairness:** In addition to meeting the quality of service requirements for different applications, it is also important to ensure that the network, being a shared resouce, is being utlized by the different flows in the desired fair manner. It is possible that even though the minimum bandwidth requirements for the application is met, the bandwidth is shared non-uniformly across the multiple flows for the application. For certain applications that are bottlenecked by the slowest flow, this might result in significant performance degradation. The flow management layer should also ensure that the fairness requirements are met across the different applications using the network [70]. In addition to be fair across the different applications, different applications might also have different priorities that also need to be handled by the flow management framework.

- **Resource utilization:** The network of switches and routers interconnecting the hundreds of thousands of servers within a data center is a major contributor to the capital costs of the data center fabric. To amortize the cost of this investment, it is important to ensure that the network is being utilized to its best. This means load balancing the different flow across the available paths in the network to yield high bandwdith utilization. Flow management is key in ensuring that the flows in the network are routed such that the network resources are utilized to their best.

- **Fault tolerance:** Due to the high number of switches and links in large scale data center topologies, failures of these individual links or switches is more common than rare. As a consequence, these topologies are built with redundant paths in order to gracefully handle failures without impacting network operations or application performance. For such large scale topologies with redundant paths, flow management is essential to route around any failures and also efficiently take advantage of the multiple paths available in the network to ensure high resource utilization and also application performance requirements.

We now discuss two possible approaches for supporting scalable and efficient flow management for data center applications. For each approach we identify the requirements of the flow management framework, the undrelying challenges and a brief description of how we address the challenges.

1.3 Network Based Flow Management

1.3.1 Requirements

- **Low overhead:** In order to meet the application performance requirements and maintain high network resource utilization, it is crucial that the flow management framework performs with low overhead, reacting to changes to data center topology/flow changes with low overhead. Researchers have shown that the churn in data center traffic is quite high, with median flow durations of only 100 ms with 80% of the flows less than 11 seconds long [13]. As such, the flow manage-

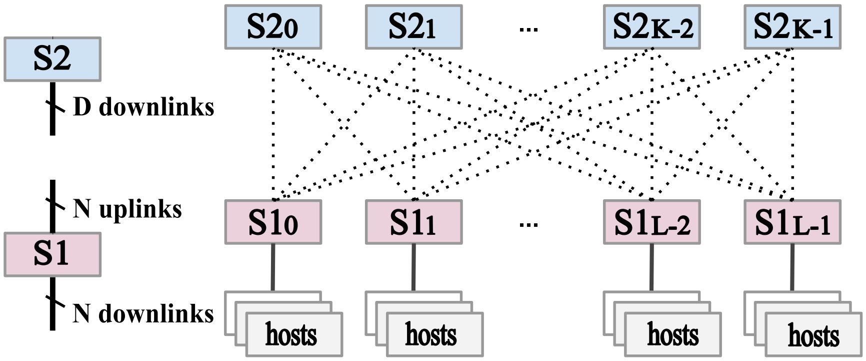


Figure 1.2. Abstract representation of data center topology

ment framework, should be able to react, compute and enforce the desired flow management scheme within few milliseconds.

- **Readily deployable:** The infrastructure investment cost is a significant factor of the CAPEX operations of the network fabric. As such frequent upgrades to hardware is cost-prohibitive and presents a significant barrier to adopting flow management schemes that require hardware changes. As such, we make ease and readiness of deployment a key requirement for the proposed network based flow management. This requires our flow management scheme to be deployable with commodity off the shelf (COTS) hardware without relying on sophisticated hardware features.

- **Efficient state management:** In order to meet the scalability challenges in large data centers, it is necessary to efficiently manage the network state across the large number of network switches and the network control servers. A performant network flow manager will strive to manage as much state as possible locally and disseminating only the necessary state updates in order to maintain high network utilization.

1.3.2 Challenges

- **Asymmetric topologies:** Figure 1.2 shows the abstract model of the data center network topology. While it is ideal to build uniform, symmetric data center network topologies that makes it possible to leverage symmetry properties for managing flows withing the network, it is hard to maintain the symmetric of data center topologies. This is due to two reasons: (1) mismatch in network hardware ports and number of switches that results in improper division of links across the available switches, (2) frequent failures that render certain network links/swiches unusable. Such asymmetric topologies make it challenging to naïvely employ solutions like Equal Cost Multipath (ECMP) that implicitly assume symmetric nature of data center topologies.

- **Limited hardware resources:** The simplest way to meet the deployability and low overhead requirement for flow management is to leverage exisiting switch hardware. One way to do this is to make use of the hardware TCAM entries in the switches for flow managemnt. However, commodity hardware has limited number of these hardware TCAM entries which introduces the challenge of judiciously using the TCAM entries for flow management.

- **Fault tolerance:** Given the large scale of data center networks, the mean time for failure of various entities including network servers, switches and links is quite high [90]. Any network based flow management framework needs to be fault tolerant in order to keep up with the frequent failures in the network fabric and adapt the flow management schemes to the different kinds of failures.

1.3.3 Weghted Cost Multipathing

Currently data centers employ Equal Cost Multipath (ECMP) for efficient flow management. While ECMP meets the scalablity and low state requirements, it fails to ensure fairness and high network utilization in presence of failures or asymmetric topologies. To address the shortcomings of ECMP, we propose Weighted Cost Multipathing (WCMP) that accounts for asymmetry in data center topologies and allows for fair and high utilization of network bandwith in presence of failures. WCMP is a readily deployable solution and requires no hardware modifications in the network switches.

1.4 End Host Based Flow Management

1.4.1 Requirements

- **Unmodified applications:** We require that the end host based flow management framework does not rely on any application changes. Enforcing this requirement provides two benefits (1) compatibility with existing applications that makes it possible for the end host flow management solution to be readily adopted in current data centers and (2) extensible framework that can evolve independently of the changes in the application semantics. Being able to support unmodified applications implies that the framework now needs to provide a flexible abstraction that works with current applications, which is currently missing from the existing network stack. We discuss this in further detail when we talk about challenges in designing and implementing such a framework.

- **Responsive to updates:** For efficient flow management, it is required that the end host stack is able to process and respond to updates in the application traffic pattern and/or network state with as low overhead as possible. Failure to respond to updates within required time will result in the end host lagging behind the network and/or application state change, subsequently causing sub-optimal performance and under-utilization of resources.

- **Precise control over packet transmissions:** Recent proposals for network management in data centers rely on the end host stack to provide functionality that allows the controller to determine precisely at what time the packets are sent out from the end host's network interface [68, 58, 83]. Currently, the existing stack provides no interface to support TDMA based packet transmissions. Achieving scalable rate limiting for a large number of queues with the existing networking infrastructure requires redesinging the NIC firmware and is prohibitive for ready deployment. To that end, we require that the end host based flow management layer supports an interface that allows applications/controllers to specify precisely when the packets should be sent out of the NIC interface.

- **Remotely manageable:** As the data centers move towards centrally managed SDN-style network fabrics, it is important to extend the SDN paradigm to the end host stack as well. Figure 1.3 shows how the existing NIC interface and the current OS stack is a bottleneck that fails to bring the end host stack under central control of the fabric manager. A remotely manageable end host stack supports the separation of control logic and end host data plane functionality and makes it possible for the end host to evolve at the same pace as the SDN based network.

1.4.2 Challenges

- **Too many layers:** The networking stack at the end hosts has been overridden with too many layers of abstraction. The current networking stack was written decades ago with assumptions that are no longer valid. The socket abstraction requires multiple copies of network packets that go through code paths of varying lengths

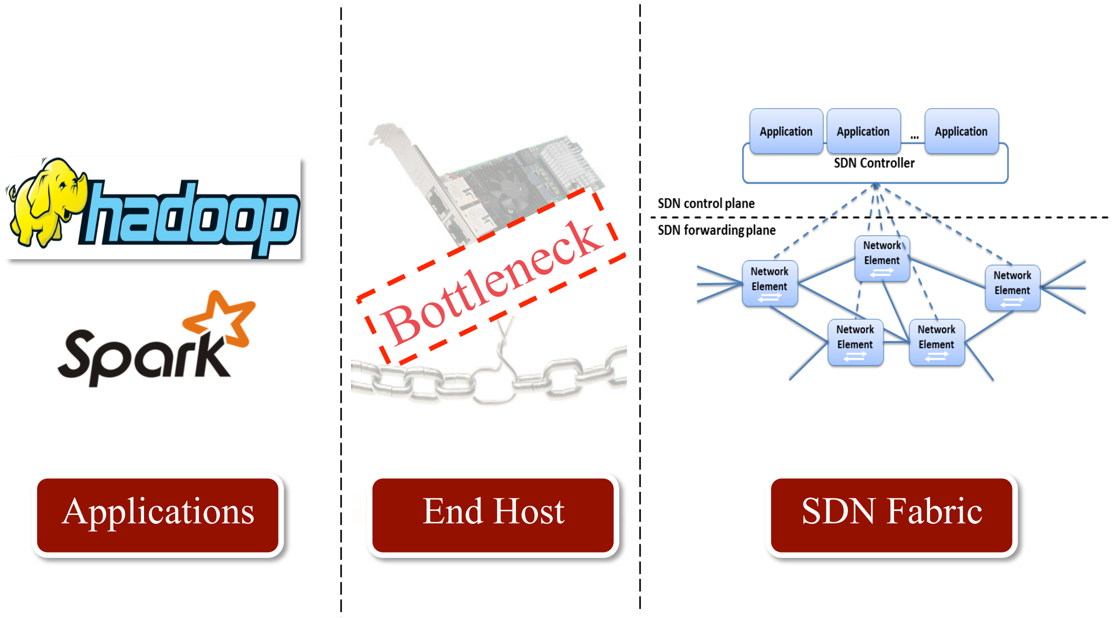


Figure 1.3. End host network stack as a bottleneck

resulting in introducting variable latency delays that makes it hard to synchronize the end host applications to fine-time scales. The layered architecture also adding latency overhead and isolcation concerns for data center applications [53, 75, 12].

- **Synchronizing transfers:** While recent efforts have aimed at improving the design and security of the networking stack, there is still missing support in the networking that makes it possible for a remote controller to precisely control the transmission of the end hosts. There are two parts to this challenge (1) synchronizing the time between the end hosts and the network and, (2) interfaces that allow the remote control to specify when the packets are transmitted. While the first challenge can be addressed by using the IEEE Precision Time Protoco (PTP) [71], the latter still remains a challenge that needs to be addressed.

- **End host isolated from centralized controller:** Software defined networking (SDN) has made it possible for network managers to control the routing and traffic management by separating the control plane and the data plane in the network switches and routers. This separation of control plane logic and data plane functionality makes it possible to network managers to enforce extensible and flexible management policies in the network at a remote controller without being bottlenecked by the switching/routing hardware. However, while the network infrastructure has evolved and is moving towards smarter designs, the end host stack has remained distributed, walled off from enabling remote contorl, only supporting ad-hoc measures to manage the end host stack.

- **Missing interface between applications and network:** The current networking stack lacks appropriate interfacing between the data center applications and the network fabric. While good at abstracting details for packet transmission and application behaviour, the OS network stack lacks appropriate interfaces for better interaction between the applications and the network. As a result, the applications lack knowledge of latest network state that they could potentially leverage for optimizing performance and the networking framework lacks the understanding of application behaviour and has to rely on approximate estimations thay result in sub-optimal network utilization. This lack of appropriate interfacing makes the problem of efficient flow management at the end hosts more challenging.

1.4.3 Software Defined Dataplane: Explicit control

To address the challenges in implementing an end host based flow management framework, we propose “Software defined dataplane (SDD)”, a novel framework that exposes an API to control how packets/flows are managed at the end host network stack. SDD leverages high performant packet processing libraries such as Dataplane development kit (DPDK) [26] and make it possible to bring data center end hosts under network control. Using the simple SDD framework, fabric managers can control packet transmissions at the end host by explicity sending “start” and “stop” control packets to the end hosts. This functionality is critical to implement a time division multiple access (TDMA) based network sharing framework recently proposed by [29, 58, 68].

1.4.4 Software Defined Dataplane: Microprograms

We augment the software defined dataplane by adding a simple programmable interface (SDD API) that allows a remote controller to send "SDD recipes", simple micropgrograms that can be executed at the end host to determine how the packets are transmitted at the end hosts. Using the simpple API, the remote controller encodes conditional statements that are evaluated at the end hosts and govern how packets are sent out of the end host NIC. This obviates the need for explicit control of packet transmissions at the end host and makes it possible to scale the SDD framework to a large number of flows in the network.

1.5 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides an overview of the background and the related work that has been done for improving flow management in data center networks. It also describes some of the latest trends in the networking research that motivate need for more extensible and flexible flow management frameworks. Chapter 3 presents Weighted Cost Multipathing (WCMP), an in-network mechanism for flow management to improve network utilization and ensure fairness across flows in presence of failures and asymmetry in data center topologies. Chapter 4 presents the motivation and strawman design for supporting flow management at the end host in a tightly-coupled manner by sending periodic control packets in order to support optical switching in data centers. Chapter 5 further prposes the Software Defined Dataplne (SDD) framework by proposing SDD API that allows a remote controller to send microprograms that can be executed at the end host for precise, efficient and scalable flow management. Finally, Chapter 6 concludes this dissertation with a summary of contributions.

Chapter 2

Background & Related Work

2.1 Data Center Topologies

A fundamental aspect of any interconnection network is design of the underlying topology. The choice of the network topology not only drives the cost of the network but also determines its performance. This is true for all the different implementations of interconnection networks including transistors on chips, memory, I/O and interfacing in computer architecture, clusters of networks for supercomputing applications and the more recent data center networks. The data center networks are the backbone of the various cloud applications and as such, the data center topology is a crucial factor that impacts the performance of data center and the cloud applications. These applications place strong requirements for routing, bandwidth and quality of service on the network and the topology. In addition to this, the scale at which data center operates makes it even harder to design the network interconnect that addresses the needs of the applications.

The early data center topologies consisted of a tree-like hierarchical structure that used expensive switching technology at the top-tiers of the network. Such a design had several shortcomings including high oversubscription at the higher levels of the tree and average link redundancy. Such topologies were expensive and made it harder to manage resource allocation and good bandwidth guarantees in the network. As the data centers grew in scale, the need for a more scalable, cost-effective and high performance topology also became stronger. The past of couple of years has seen various interesting proposals made for data center networks. With a multiple different possible data center topology proposals available with different trade-offs, it becomes essential to develop a sound understanding of different topology designs and their relative merits.

2.1.1 Topology Design Goals

It is essential to understand the expectations and goals from a system before choosing the right interconnect for the system. Here we list few of the important design goals that need to be kept in mind while designing a data center topology.

- *Scalability*: There are different scales at which a data center topology can be designed. For instance, certain data center topology are designed to cater the needs of a mega data center and there are few that are designed for interconnecting servers in a modular container based data center. In their case the topology should be designed to meet the purpose. As it turns out, for both the two different purposes, the designs differ greatly as the requirements and expectations from the two are quite different. In DCell [38], the authors describe three important aspects of scaling. First, the physical structure has to be scalable. It must physically interconnect hundreds of thousands or even millions of servers at small cost, such as a small number of links at each node and no dependence on high-end switches to scale up. Second, it has to enable incremental expansion by adding more servers into the already operational structure. When new servers are added, the existing running servers should not be affected. Third, the protocol design such as routing also has to scale.

- *High network capacity*: Many online infrastructure services need large amount of network bandwidth to deliver satisfactory runtime performance. Using the distributed Google file system as an example (GFS) [33], a file is typically replicated several times to improve reliability. When a server disk fails, re- replication is performed. File replication and re-replication are two representative, bandwidth-demanding one-to-many and many-to-one operations. Another application example requiring high bandwidth is MapReduce [24]. In its Reduce operation phase, a Reduce worker needs to fetch intermediate files from many servers. The traffic generated by the Reduce workers forms an all-to-all communication pattern, thus requesting for high network capacity from the network.

- *Ease of Manageability*: In addition to being scalalable and high network capacity, the network design should be such that it helps in easier management of the network with easy to define and implement routing and access control policies.

- *Fault Tolerance*: Failures are quite common in current data centers [10]. There are various server, link, switch, rack failures due to hardware, software, and power outage problems. As the network size grows, individual server and switch failures may become the norm rather than excep- tion. Fault tolerance in data center requests for both redundancy in physical connectivity and robust mechanisms in protocol design.

- *Energy Efficient*: Recently, there has been a big movement towards green computing and building energy efficient data centers. Building energy-efficient data centers is important not only because it brings down the operation expenditure (OpEx) of a data center but also because energy is a depleting resource that needs to used judiciously. The concept of *energy proportionality* is gaining popularity in the data center community and has received a lot of attention in research [1].

- *Cost-effective*: Data center costs make up the biggest portion of the infrastructure expenditure for companies like Google, Microsoft, Facebook and Amazon. Designing a cost-effective topology is important because it represents a significant fraction of the initial capital investment while not contributing directly to future revenues.

2.1.2 Wired Topologies

The data centers comprise of tens of thousands of servers in a single large facility. These servers are organized into racks and rows for a space optimizing, efficient layout which are then interconnected to create the physical network. In this section, we describe in detail the properties and popular examples of the topology resulting from the static physical interconnections in a data center.

Clos Topology: Clos topology [20] was first proposed in year 1952 by Charles Clos for interconnection telephone networks. Clos networks are an example of *non-blocking* indirect network. Non-blocking means that a dedicated path can be formed from each input to its selected output without any conflicts for all permutations of the inputs and output. A Clos topology is an n stage network (where n is odd) in which each stage is composed of number of crossbar switches. In a three stage Clos network, the middle stage has one input link from every input switch (which are connected to servers) and one output link to eery output switch(also connected to the servers). One advantage of Clos networks is that it reduces the number of crossbar switches required to achieve a non-blocking interconnect. The high path diversity of a Clos network makes it possible to provide large network capacity and fault-tolerance.

Hypercube Topology: The generalized hypercube structure was proposed in 1984 [15]. The authors describe the motivation for the generalized design comes from the fact that an interconnection structure in general should have a low number of links per node (degree of a node), a small internode distance (diameter), and a large number of alternate paths between a pair of nodes for fault tolerance. Traditional hypercube structures can support N = k^d nodes for some integer values of k and d. In contrast, the generalized hypercube structure supports any number nodes N. Generalized hypercube networks have mixed radix that makes them more flexible and at the same time have smaller latencies as compared to the hypercube topology.

One of the recent architectures proposed for modular data center networks is very closely related to the generalized hypercube structure. In the BCube [37] interconnection structure, each server is equipped with a small number of ports (typically no more than four). Multiple layers of cheap mini-switches are used to connect those servers. BCube provides multiple parallel short paths between any pair of servers. In a BCube network, if we replace each switch and its n links with an n x (n - 1) full mesh that directly connects the servers, we get a generalized hypercube. The multiple layers of switched connections between the servers not only provides high one-to-one bandwidth, but also greatly improves fault tolerance and load balancing. BCube accelerates one-to-several and one-to-all traffic by constructing edge-disjoint complete graphs and multiple edge-disjoint server spanning trees. Moreover, due to its low diameter, BCube provides high network capacity for all-to-all traffic such as MapReduce. BCube runs a source routing protocol called BSR (BCube Source Routing). BSR places routing intelligence solely onto servers. By taking advantage of the multi-path property of BCube and by actively probing the network, BSR balances traffic and handles failures without link-state distribution. With BSR, the capacity of BCube decreases gracefully as the server and/or switch failure increases. BSR is similar to the routing algorithm for Hypercube due to the similarity between the BCube and generalized hypercube architectures.

Random Topology: Apart from traditional network topologies that have well defined connections and properties, researchers have also proposed the use of random topologies in the data center [76]. The motivation for random topologies comes from the graph theoretic result that random graphs have low diameter and high bisection bandwidth. Also these graphs are highly flexible, can be built with any number of nodes and regular or heterogenous degree distributions and can be scaled arbitrarily. The Jellyfish approach is to construct a random graph at the switch layer. Each switch i has some number k_i of ports, of which it uses r_i to connect to other switches, and uses the remaining $k_i - r_i$ ports for servers. In the simplest case, every switch has the same number of ports and servers. If N be the number of switches, so the network supports N(k - r) servers. The resulting the network is a random regular graph. The particular topology is uniform-randomly sampled from the sample space of all such topologies.

2.1.3 Dynamic Topologies

The topologies described so far are statically wired topologies that have certain inherent disadvantage in adapting to evolving services and traffic patterns because of their rigid, static structure. Though researchers have proposed logical overlays and topology management constructs [64] which make it possible for to handle the dynamic work patterns but this comes at the cost of additional complexity and performance overhead. Recently there have been proposals made for dynamically augmenting an existing topology using wireless or optical interconnects in order to adapt to the dynamic bandwidth requirements in a data center. The motivation for using dynamic links comes from the observation that many applications that run in their production data-centers do not require full bisection bandwidth.

Using optical interconnects: Due to the high cost of establishing a circuit because of the setup delay, the setup cost must be amortized by the gains achieved by setting up the circuit. This means that longer lived elephant flows are a better candidate to be switched over the traffic than short-lived bursty flows. Helios and c-Through use single hop optical links to connect an aggregate of servers or *pods* as in Helios terminology as the aggregate traffic from a pod demands from a rack meets the suitable traffic requirements. Even at aggregated levels, all pods do not require high bandwidth at all times and the traffic matrix for inter-pod traffic changes over time. In order to identify which set of pods need to communicate using the optical switches, these systems operate by estimating the traffic demand in the network and use estimation techniques to identify the large elephant flows. The control loop in both these systems constitute of three main steps. 1) Estimate the demand matrix. 2) Run an matching algorithm that to compute a new topology with optical links that will maximize the throughput. 3) Update topology by programming the circuit switch and also updating the flow tables in the pod switches accordingly. The difference between the two systems in the way they measure the demand matrix. Helios collects traffic stats from the pod switches and then uses the Demand Estimation algorithm from Hedera [5] to build the demand matrix whereas c-Though looks at the socket buffer occupancy of TCP flows at the hosts and then aggregates this data in order to create the demand matrix.

For Helios and c-Though, the resulting hybrid topology makes use of single hop links optical links and is still a tree based hierarchical topology. Proteus on the other hand takes advantage of the optical interconnects that's *malleable* to the traffic in the data center where any subset of the server population can be provided with high bandwidth on demand. Their design exploits the reconfigurability of the optical switches and also the ability to change the optical wavelength provisioning for dynamic link capacity at runtime. Proteus begins with connecting N top of rack (ToRs) to one port on an N-port

MEMS switch. The circuit connections in the optical switch can be changed on demand to determine which set of ToRs are connected. Which ToRs are connected by the optical switch is determined on demand by latest traffic in the network. Given a connected ToR graph, proteus uses hop-by-hop links to achieve network connectivity. To reach ToRs not directly connected to it through the optical switch, a ToR uses one of the *k* connections that it has with the optical switch. This means that in order to keep up bandwidth requirements, high volume connections must use a minimal number of hops. Proteus also combines the capability of optical fibers to carry multiple wavelengths at the same time (WDM) with the dynamic recongurability of the wavelength selective switch. Consequently, a ToR is connected to MEMS through a multiplexer and a WSS unit.

Using wireless interconnect: In addition to the use of optical interconnect, there has been work that explores that applicability of wireless flyways in the data center [40, 91]. Like the hybrid circuit and packet switched networks, the motivation for experimenting with wireless in the data center is also motivated by the recent technological trends. The availability of the wireless spectrum between the 57-64 GHz, referred to as the 60Ghz band potentially makes it possible to use wireless to support multi-Gbps rates in the data center. The dynamically configured links - called flyways add extra capacity to the base network to alleviate hotspots - racks with high bandwidth requirements. The basic design of a data center network with 60 GHz flyways includes a base wired network which is provisioned for the average case and can be oversubscribed. Each top-of-rack (ToR) switch is equipped with one or more 60 GHz wireless devices, with electronically steerable directional antennas. A central controller monitors the traffic patterns, and switches the beams of the wireless devices to set up flyways between ToR switches that provide added bandwidth as needed. When the traffic matrix is sparse (i.e. only a few ToR switches are hot), a small number of flyways can significantly improve performance, without the cost of building a fully non-oversubscribed network. Like Helios and c-Though, the system for setting up the flyways involves the three steps of estimating traffic, identifying the flyways that need to be setup and making appropriate routing changes to accommodate for the flyways. Kandula et al propose a *flyway picker* algorithm that requires knowledge of traffic patterns. In some cases (e.g., multi-tenant data centers) traffic patterns may not be predictable, and there may be no cluster-wide scheduler. In such cases, they propose the use of an online traffic engineering approach such as that described in [14], combined with the ability to rapidly steer antennas (every few seconds. For a given data center layout, the number of flyways that may be set up at the same time depends on the antenna used, and the minimum throughput required from each link.

2.2 Network Based Flow Management

Load balancing and traffic engineering have been studied extensively for WAN and Clos networks. Many traffic engineering efforts focus on the wide-area Internet [9, 55, 86, 89] across two main categories: (i) traffic engineering based on measured traffic demands and (ii) oblivious routing based on the “hose” model [55]. Our work differs from the previous work as it addresses unfairness to topology asymmetry and proposes a readily deployable solution. Further more, efficient WAN load balancing relies on traffic matrices which change at the scale of several minutes or hours. However, for data centers highly dynamic traffic matrices, changing at the scale of few seconds motivate the need for load balancing solutions with low reaction times.

While ours is not the first work that advocates weighted traffic distribution, we present a practical, deployable solution for implementing weighted traffic distribution in current hardware. There has been prior work that proposed setting up parallel MPLS-TE tunnels and splitting traffic across them unequally based on tunnel bandwidths [63].

The load balancing ratio between the tunnels is approximated by the number of entries for each tunnel in the hash table. However, the proposal does not address the concern that replicating entries can exhaust switch forwarding table. In this work, we present algorithms that address this challenge explicitly.

Villamizar [81] proposed unequal splitting of traffic across the available paths by associating a hash boundary with each path. An incoming packet is sent out on the path whose hash boundary is less than the hash of the packet header. These hash boundaries serve as path weights and are adjusted based on the current demand and updated repeatedly. Implementing hash boundaries requires switch hardware modification, raising the bar for immediate deployment. Our contributions include proposing a complete, deployable solution that improves load balancing in data centers without hardware modification.

Many heuristic load-balancing algorithms have been proposed [65, 44] for Clos networks. Most of these algorithms are based on models for simultaneous or sequential dynamic scheduling, not practical at the scale of data centers. Geoffray and Hoefler [32] describe a number of strategies to increase the effective bandwidth in multistage interconnection networks, but such approaches focus on source-routed, per-packet dispersive approaches that break the ordering requirement of TCP/IP over Ethernet. In general, we are not aware of any research that addresses building an operational multi-level switch architecture using existing commodity components and designs.

Highly dynamic, reactive traffic engineering techniques like MATE [27] and TeXCP [50] have also been proposed. These schemes are based on explicit utilization or congestion feedback received from the core routers. These algorithms can converge quickly and do not need to collect many traffic samples, but they require specific switch support which is not available on today's commodity switches. FLARE [77] is another technique that proposes multi-path forwarding in the wide-area on the granularity of flowlets (TCP packet bursts). It is not clear though whether the low intra-data center latencies meet the timing requirements of flowlet bursts to prevent packet reordering while achieving high performance.

In the data center context, there have been many proposals which advocate multipath based topologies [4, 36, 37, 59, 85] to provide higher bandwidth and fault tolerance. F10, a proposal for building fault resilient data center fabric [59] advocates the use of weighted ECMP for efficient load balancing in presence of failures. Such network fabrics can leverage WCMP as a readily deployable solution for improving fairness. These topologies with high degree of multipaths have triggered other efforts [5, 22, 23] that evenly spread flows over the data center fabric for higher utilization. Hedera focuses on dynamically allocating elephant flows to paths via a centralized scheduler by repeatedly polling the switches for network utilization and/or flow sizes. Mice flows, however, would still be routed in a static manner using ECMP. Hedera's centralized measurement and software control loop limit its responsiveness to mice flows.Hence, our WCMP algorithms provide baseline short-flow benefits for systems like Hedera, and scale better for large networks with balanced flow sizes (for applications e.g TritonSort/MapReduce).

MPTCP [74], a transport layer solution, proposes creating several sub-flows for each TCP flow. It relies on ECMP to hash the sub-flows on to different paths and thus provide better load balancing. In comparison, WCMP hashes flows across paths according to available capacity based on topology rather than available bandwidth based on communication patterns. We find that for uniform communication patterns, WCMP outperforms MPTCP whereas MPTCP outperforms WCMP in the presence of localized congestion. The two together perform better than either in isolation as they work in a complementary fashion: WCMP makes available additional capacity based on the topology while MPTCP better utilizes the capacity based on dynamic communication patterns as we show in our evaluation section.

MPTCP [74] proposes modifications in the transport layer and provides load balancing by making TCP aware of multiple paths at the transport layer. It improves overall system throughput by initiating several sub-flows for each connection, with the hope that these sub-flows will fall on different paths and thus avoiding hash collisions. MPTCP has the opportunity to dynamically load balance around persistent hot spots, those that arise from imbalanced flow sizes or incast communication patterns [17]. Since MPTCP relies on ECMP at the routing layer for flow hashing, WCMP complements MPTCP performance by enabling more balanced distributions of flows by weighted hashing. While, MPTCP is less suitable for workloads with many short flows and large degrees of multipath, WCMP guarantees fairness for such workloads as well. Also, the additional connection overhead and crypto-authentication for each subflow means that mice flows may be subject to increased latency, an issue for deployments targeting the lowest baseline latency [7, 67].

As importantly, the bandwidth probing overhead of TCP across sub-flows means that a small transfer would likely complete before finding the right balance across multiple paths. With our WCMP approach, flows are weighted across paths according to available capacity, meaning that small flows will statistically have full capacity available to them. Overall, in case of balanced communication patterns or traffic pattern exhibiting ON-OFF characteristics, hashing schemes like WCMP will be more effective than MPTCP as there will be no additional overhead in estimating available capacity along paths for dynamic bandwidth allocation. Our data center communication patterns tend to exhibit ON-OFF traffic similar to earlier studies [13, 52] and achieve fairness improvements with WCMP hashing. Finally, MPTCP is a layer 4 solution, requiring modified kernel at end hosts, while our work is a layer 3 solution requiring software upgrade on router. With in-service software upgrade supported on most routers, WCMP is a more practical solution for data centers that cannot afford service interruption incurred by host kernel upgrade.

2.3 End Host Based Flow Management

Software-defined networking has enabled centralized management of the network control plane. This lets network operators build centralized management applications that determine “how” packets are handled in the network in terms of routing, access control, load balancing, and traffic engineering. However, SDN does not prescribe “when” packets are transmitted, how they are paced, or how end hosts interleave packets across multiple flows headed to different destinations. This packet- and flow-level transmission behavior is important, as it directly impacts bandwidth utilization, latency and burstiness of the resulting traffic, and by extension, the overall performance of data center applications.

In general, network interface cards have been subject to ossification due to long hardware development cycles, preventing new features from being deployed in hardware. At the other extreme, packet processing in the kernel has been subject to latency and latency variation too high to support the above-mentioned functionality [73, 53]. This has resulted in the development of high-throughput packet-processing network stacks based on commodity hardware such as netmap [75] and DPDK [26], processing packets entirely outside of the kernel (DPDK) or partially outside of kernel control (netmap). Alternative approaches include “green field” OS designs [69, 12]. SoftNIC [42] aims to provide an extensible, entirely software-based NIC built using frameworks such as DPDK.

The connection between in-network buffering and end-host synchronization has been long studied in the networking research literature. Buffering in packet switches is what enables end hosts to remain loosely synchronized; it is possible to rely on statistical multiplexing to “smooth out” packet arrivals from multiple hosts across time by storing and forwarding packets through in-switch buffers. Achieving tight synchronization among hosts on the Internet is impossible in general. In cloud and datacenter environments, however, ensembles of servers frequently act as a tightly-coordinated cluster infrastructure, especially in contexts that impose millisecond- or even microsecond-level SLAs on application response time. Thus, by better synchronizing By coordinating packet transmission times across data center hosts, it is possible to greatly reduce—or even eliminate—in-network buffering. This is good for two reasons: first, it lowers end-to-end queueing and latency, and second, it enables cheaper and more energy-efficient network designs. A number of proposals advocate such an approach, including FastPass [68], DCTCP [6], and TDMA-based Ethernet [80]. While these proposals assume buffered packet switches, they aim to reduce the use of those buffers by carefully admitting traffic into the network. A second line of work aims to entirely eliminate the buffering within the network fabric to support end-to-end reconfigurable circuit-switching, either based on optical [29, 28, 82] or wireless [51] circuit switches. Here the need for eliminating in-network buffering comes from the physical requirements of the circuit technology, and so end-hosts and switches must implement strict TDMA, requiring high levels of end-host synchronization (at millisecond or even microsecond timescales).

Chapter 3

WCMP: Weighted Cost Multipathing

3.1 Introduction

There has been significant recent interest in a range of network topologies for large-scale data centers [4, 37, 54]. These topologies promise scalable, cost-effective bandwidth to emerging clusters with tens of thousands of servers by leveraging parallel paths organized across multiple switch stages. Equal Cost Multipath (ECMP) extension to OSPF [36, 45] is the most popular technique for spreading traffic among available paths. Recent efforts [5, 43] propose dynamically monitoring global fabric conditions in the network to adjust forwarding rules, potentially on a per-flow basis. While these efforts promise improved bandwidth efficiency, their additional complexity means that most commercial deployments still use ECMP based forwarding given the simplicity of using strictly local switch state for packet forwarding.

To date, however, the current routing and forwarding protocols employing ECMP-style forwarding focus on regular, symmetric, and fault-free instances of tree-based topologies. At a high-level, these protocols assume that there will be: (i) large number of equal cost paths to a given destination, and (ii) equal amount of bandwidth capacity to the destination downstream among all equal cost paths. The first assumption is violated for non-tree topologies such as HyperCubes and its descendants [3, 37] that require load balancing across non-equal cost paths. The second assumption is violated when a failure downstream reduces capacity through a particular next hop or, more simply, when a tree-based topology inherently demonstrates *imbalanced striping* (defined in Section 3.4). This happens when the number of switches at a particular stage in the topology is not perfectly divisible by the number of uplinks on a switch at the previous stage.

Thus, for realistic network deployments, the bandwidth capacity available among "equal cost" next hops is typically guaranteed to be unequal even in the baseline case where all flows are of identical length. That is to say, ECMP forwarding leads to imbalanced bandwidth distribution simply from static topology imbalance rather than from dynamic communication patterns. It is our position that routing protocols should be able to correct for such imbalance as they typically already collect all the state necessary to do so. Based on our experience with large-scale data center deployments, we have found that substantial variation in per-flow bandwidth for flows that otherwise are not subject to prevailing congestion both reduces application performance and makes the network more difficult to diagnose and maintain.

This paper presents Weighted Cost Multipathing (WCMP) to deliver fair per-flow bandwidth allocation based on the routing protocol's view of the topology. We present the design and implementation of WCMP in a Software Defined Network running commodity switches, which distribute traffic among available next hops in proportion to the available link capacity (not bandwidth) to the destination according to the dynamically changing network topology. We further present topology connectivity guidelines for improving network throughput. Inserting the necessary weighted forwarding entries can consume hundreds of entries for a single destination, aggressively consuming limited on-chip SRAM/TCAM table entries. Hence, we present algorithms that trade a small bandwidth oversubscription for substantial reduction in consumed forwarding table entries. Our results indicate that even a 5% increase in oversubscription can significantly reduce the

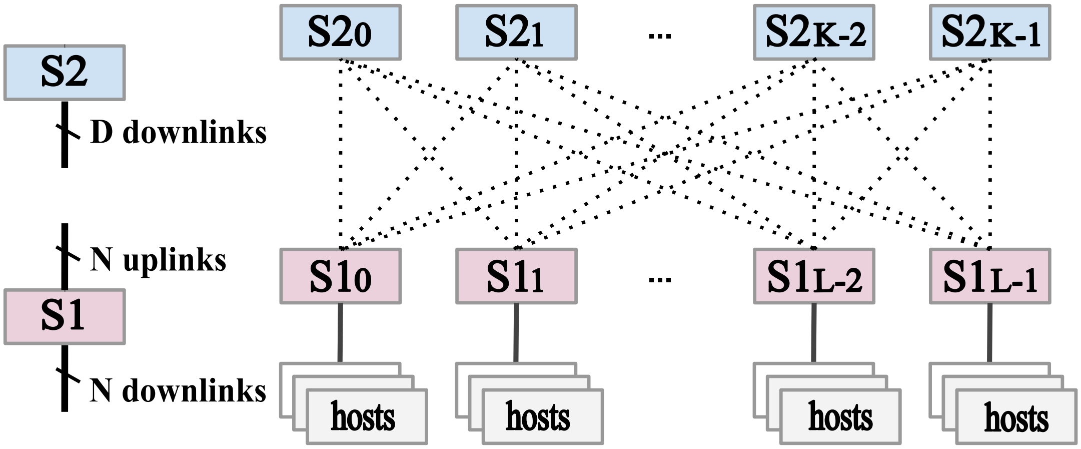


Figure 3.1. Two stage clos network

number of forwarding table entries required. Our algorithms for weight reduction make WCMP readily deployable in current commodity switches with limited forwarding table entries.

We present our evaluation of WCMP in an OpenFlow controlled 10Gb/s data center network testbed for different kinds of traffic patterns including real data center traffic traces from [13]. The WCMP weights are meant to deal with long lived failures (switch/link failures, link flappings etc.) included in the routing updates, and our ability to scale and react is only limited by the reactivity of the routing protocol. Our results show that WCMP reduces the variation in flow bandwidths by as much as 25× compared to ECMP. WCMP is complementary to traffic load balancing schemes at higher layers such as Hedera [5] and MPTCP [74].

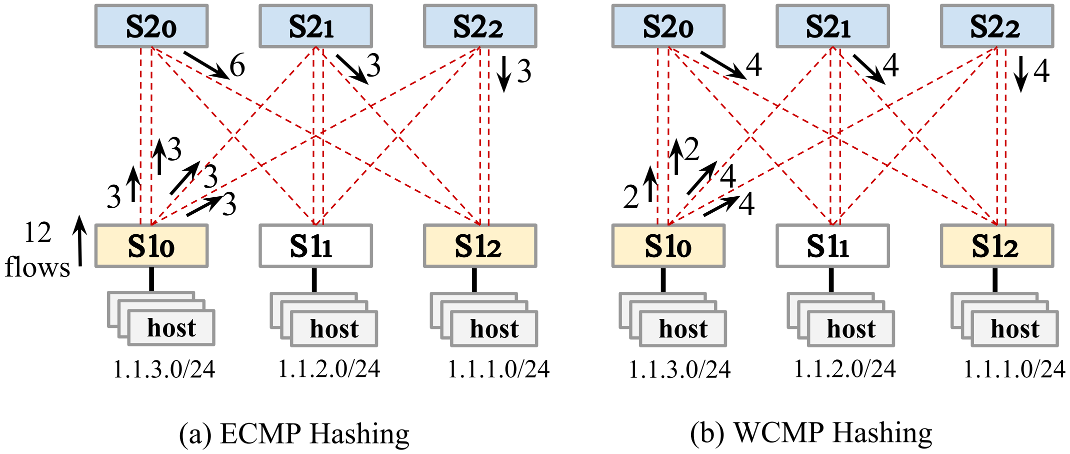
3.2 Background & Motivation

The primary objective of this paper is to propose a deployable solution that addresses the ECMP weakness in handling topology asymmetry, thus improving fairness across flow bandwidths and load balancing in data center networks. While our results apply equally well to a range of topologies, including *direct connect* topologies [3, 37, 85], for concreteness we focus on multi-stage, tree-based topologies that form the basis for current data center deployments [4, 36].

We abstract the data center network fabric as a two-stage Clos topology, as depicted in Figure 3.1. Here, each switch in Stage 1 (S1) stripes its uplinks across all available Stage 2 (S2) switches. Hosts connected to the S1 switches can then communicate by leveraging the multiple paths through the different S2 switches. Typically, ECMP extensions to routing protocols such as OSPF [62] distribute traffic equally by hashing individual flows among the set of available paths. We show how ECMP alone is insufficient to efficiently leverage the multipaths in data center networks, particularly in presence of failures and asymmetry. While we focus on a logical two stage topology for our analysis, our results are recursively applicable to a topology of arbitrary depth. For instance, high-radix S1 switches may internally be made up of a 2-stage topology using lower-degree physical switches; our proposed routing and load balancing techniques would apply equally well at multiple levels of the topology.

3.2.1 Motivating Example

In a multi-stage network, *striping* refers to the distribution of uplinks from lower stage switches to the switches in the successive stage. We define striping in more detail in Section 3.4 but want to point out that a uniformly symmetric or balanced striping requires the number of uplinks at a lower stage switch be equal to (or an integer multiple of) the number of switches in the next stage, to ensure that the uplinks at a lower stage switch can be striped uniformly across the switches in the next stage. In large scale data centers, it is hard to maintain such balanced stripings due to heterogeneity in switch port counts and frequent switch/link failures. We discuss the impact of striping asymmetry in detail in later sections. Here, we motivate how employing ECMP over topologies with uneven striping results in poor load balancing and unfair bandwidth distribution.



(a) ECMP Hashing

(b) WCMP Hashing

Figure 3.2. Multipath flow hashing in an asymmetric topology

Consider the 2-stage Clos topology depicted in Figure 3.2(a). Each link in the topology has the same capacity, 10 Gb/s. Since each S1 switch has four uplinks to connect to the three S2 switches, it results in asymmetric distribution of uplinks or *imbalanced striping* of uplinks across the S2 switches. Assuming ideal ECMP hashing of 12 flows from the source switch $S1_0$ to the destination switch $S1_2$, three flows each are hashed onto each of the four uplinks at $S1_0$. However, the six flows reaching $S2_0$ now have to contend for capacity on the single downlink to $S1_2$. As such, the six flows via $S2_0$ receive *one-sixth* of the link capacity, 1.66 Gb/s each, while the remaining six flows receive *one-third* of the link capacity, 3.33 Gb/s each, resulting in unfair bandwidth distribution even for identical flows being hashed by ECMP.

Note that the effective bandwidth capacity of the two uplinks to $S2_0$ at $S1_0$ is only 10 Gb/s, bottlenecked by the single downlink from $S2_0$ to $S1_2$. Taking this into consideration, if $S1_0$ weighs its uplinks in the ratio 1:1:2:2 for hashing (as opposed to 1:1:1:1 with ECMP) all flows would reach the destination switch with the same throughput, one-fourth of link capacity or 2.5 Gb/s in this example, as shown in Figure

3.2(b). This observation for weighing different paths according to their effective capacity is the premise for *Weighted Cost Multipathing (WCMP)*.

3.2.2 Bandwidth Fairness

Data center applications such as Web search [10], MapReduce [24], and Memcached [60] operate in a bulk synchronous manner where initiators attempt to exchange data with a large number of remote hosts. The operation can only proceed once all of the associated flows complete, with the application running at the speed of the slowest transfer. Similarly, when there are multiple "bins" of possible performance between server pairs, performance debugging becomes even more challenging; for instance, distinguishing between network congestion or incast communication and poor performance resulting from unlucky hashing becomes problematic, motivating fair bandwidth distribution. Orchestra [19], a system for managing data transfers in large clusters, demonstrated how fair flow scheduling at the application layer resulted in a 20% speedup of the MapReduce shuffle phase, motivating the need for fairness in bandwidth distribution across flows.

3.2.3 Reasons for Topology Asymmetry

While it is desirable to build regular, symmetrical topologies in the data centers, it is impractical to do so for the following reasons.

(1) **Heterogeneous Network Components:** Imbalanced striping is inherent to any topology where the number of uplinks on S1 switches is not an integer multiple of the total number of S2 switches. Building the fabric with heterogeneous switches with different port counts and numbers inherently creates imbalance in the topology. Depending on the required number of server ports, such imbalance in striping is typically guaranteed in practice without substantial network overbuild.

(2) **Network Failures:** Even in the case where the baseline topology is balanced, common case network failures will result in imbalanced striping. Consider a data center network consisting of 32 S2 switches with 96-ports each, 48 S1 switches with 128-ports each and 3072 hosts. Each S1 switch uses 64 ports to connect to the S2 switches, with two uplinks to each S2 switch. For this small data center, an average monthly link availability of 99.945 will result in more than 300 link failures each month, creating asymmetric links between S1 and S2 switches and making the topology imbalanced. If any one of the two downlinks from an S2 switch to a destination S1 switch fails, the flows hashed to that particular S2 switch will suffer a 50% reduction in bandwidth, since the destination S1 switch is now reachable by only one downlink. Since ECMP fails to account for capacity reductions due to failures while hashing flows, it leads to unfair bandwidth distribution. This is particularly harmful for the performance of many data center applications with scatter and gather traffic patterns that are bottlenecked by the slowest flow. Tables 3.1 and 3.2 show availability measurements of production data center switches and links (collected by polling SNMP counters) in support of our claim.

Table 3.1. Availability of Switches and Links by Month

	Oct	Nov	Dec	Jan
Switch	99.989	99.984	99.993	99.987
Link	99.929	99.932	99.968	99.955

Table 3.2. Availability of Switches and Links by Week

	01/29	02/05	02/12	02/19
Switch	99.996	99.997	99.984	99.993
Link	99.977	99.976	99.978	99.968

3.3 Weighted Cost Multi Pathing

The need for WCMP is motivated by the asymmetric striping in the data center topologies. The right striping is indeed crucial in building a data center that ensures fairness across flows and efficient load balancing. We present our proposals for striping alternatives in Section 3.4. In this section, we discuss the current practices, challenges and our approach in designing a deployable solution for weighted traffic hashing.

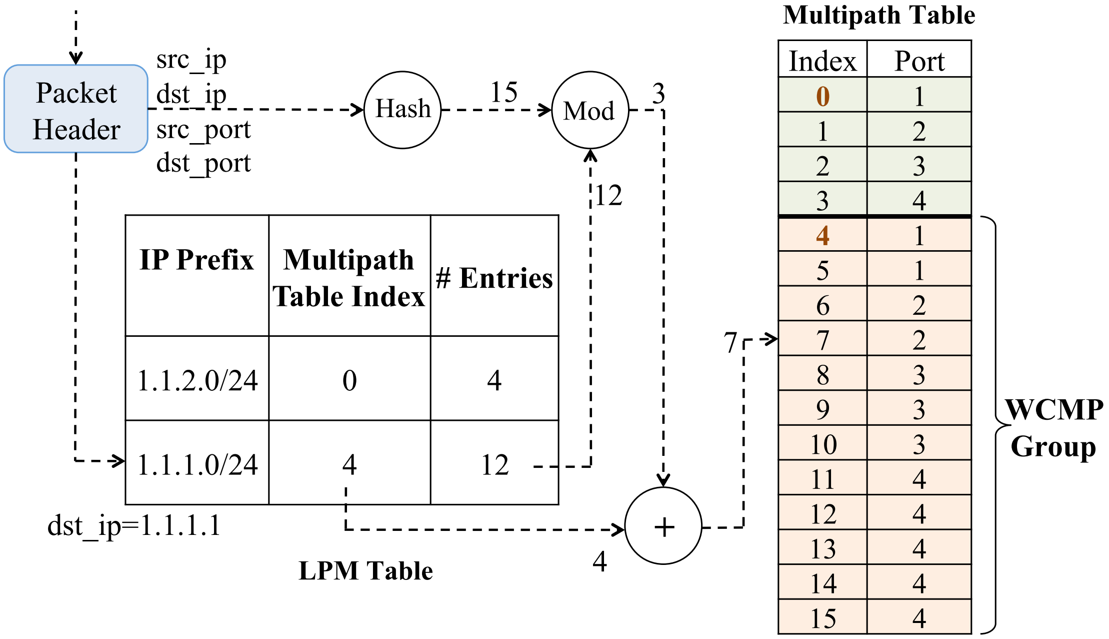
3.3.1 Multipath Forwarding in Switches

Switches implement ECMP based multipath forwarding by creating multipath groups which represent an array of "equal cost" egress ports for a destination prefix. Each egress port corresponds to one of the multiple paths available to reach the destination. The switch hashes arriving packet headers to determine the egress port for each packet in a multipath group. Hashing on specific fields in the packet header ensures that all packets in the same flow follow the same network path, avoiding packet re-ordering.

To implement weighted hashing, we assign weights to each egress port in a multipath group. We refer to the array of egress ports with weights as the *WCMP group*. Each WCMP group distributes flows among a set of egress ports in proportion to the weight for each port. The weight assigned to an egress port is in turn proportional to the capacity of the path(s) associated with that egress port. Currently, many commodity switches offer an OpenFlow compatible interface with their software development kit (SDK) [11, 46]. This allows us to realize weight assignment by replicating a port entry in the multipath table in proportion to its weight.

Figure 3.3 shows the packet processing pipeline for multipath forwarding in a commodity switch. The switch's multipath table contains two groups. The first four entries in the table store an ECMP group for traffic destined to prefix 1.1.2.0/24. The next

12 entries in the table store a WCMP group for weighted distribution of traffic destined to prefix 1.1.1.0/24. Traffic ingressing the switch is first matched against the Longest Prefix Match (LPM) entries. Upon finding a match, the switch consults the multipath group entry to determine the egress port. For example, a packet with destination 1.1.1.1 matches the LPM table entry pointing to the WCMP group with base index of 4 in the multipath table. The switch determines the offset into the multipath table for a particular packet by hashing over header fields e.g., IP addresses, UDP/TCP ports, as inputs. The hash modulo the number of entries for the group added to the group's base index determines the table entry with the egress port for the incoming packet ((15 mod 12) + 4 = 7).



IP Prefix	Multipath Table Index	# Entries
1.1.2.0/24	0	4
1.1.1.0/24	4	12

Index	Port
0	1
1	2
2	3
3	4
4	1
5	1
6	2
7	2
8	3
9	3
10	3
11	4
12	4
13	4
14	4
15	4

Figure 3.3. Flow hashing in hardware among the set of available egress ports

Replicating entries for assigning weights can easily exceed the number of table entries in commodity silicon, typically numbering in the small thousands. To overcome this hardware limitation on table entries, we map the "ideal" WCMP port weights onto a smaller set of integer weights, with the optimization goal of balancing consumed multipath table resources against the impact on flow fairness. In our switch pipeline example, the egress port numbers 1, 2, 3, 4 in the WCMP group have weights 2, 2, 3, 5 respectively (weight ratio 1:1:1.5:2.5) and use 12 entries in the multipath table to provide ideal fairness. If we change these weights to 1, 1, 2, 3 respectively, we reduce the number of table entries required from 12 to 7 with small changes to the relative ratios between the weights. This reduction is extremely useful in implementing weighted hashing as this helps in significantly lowering down the requirement for multipath table entries. This is important for large scale networks as they require many multipath groups for forwarding making hardware table entries a scarce resource in commodity switches. In the next section, we present algorithms that aim at reducing the WCMP weights with limited impact on fairness.

3.3.2 Weight Reduction Algorithms

We begin by formalizing how WCMP egress port weights affect network performance and the amount of traffic served by the different ports in a WCMP group. Consider a WCMP group G with P member egress ports. We denote the weights for each member port in G by the tuple $(X_1, X_2, ..X_P)$, and use the following definitions:

λ_G: maximum traffic demand that can be served by the WCMP group G

$B_i(G)$: maximum traffic demand served by the i^{th} member port in the WCMP group G

$G[i].weight$: weight of i^{th} member port in the WCMP group G (also denoted by X_i)

$G.size$: number of table entries used by the WCMP group G (sum of all X_i)

$$B_i(G) = \lambda_G \cdot \frac{G[i].weight}{\sum_{k=1}^{P} G[k].weight} = \lambda_G \cdot \frac{X_i}{\sum_{k=1}^{P} X_k} \tag{3.1}$$

Let $G\prime$ denote a new WCMP group reduced from G. It has the same set of member egress ports as G but with smaller weights denoted by $(Y_1, Y_2, ..Y_P)$. Given a traffic demand of λ_G, the fraction of traffic demand to the i^{th} member port, we compute $B_i(G\prime)$ by replacing G with $G\prime$ in eq. 3.1. When $B_i(G\prime) > B_i(G)$, the i^{th} port of G receives more traffic than it can serve, becoming oversubscribed. To reduce the number of required table entries, we seek to reduce the weights in G to obtain $G\prime$ while observing a maximum oversubscription of its member ports, denoted as $\Delta(G, G\prime)$:

$$\begin{aligned}\Delta(G,G\prime) &= max_i(B_i(G\prime)/B_i(G)) \\ &= max_i(\frac{G\prime[i].weight \cdot \sum_{k=1}^{P} G[k].weight}{G[i].weight \cdot \sum_{k=1}^{P} G\prime[k].weight}) \\ &= max_i(\frac{Y_i \cdot \sum_{k=1}^{P} X_k}{X_i \cdot \sum_{k=1}^{P} Y_k})\end{aligned} \tag{3.2}$$

While reducing the weights for a WCMP group, we can optimize for one of two different objectives: (i) maximum possible reduction in the group size, given a maximum oversubscription limit as the constraint, or (ii) minimizing the maximum oversubscription with a constraint on the total size of the group.

Weight Reduction with an Oversubscription Limit

Given a WCMP group G with P member ports and a maximum oversubscription limit, denoted by parameter θ_{max}, we want to find a WCMP group $G\prime$ with P member ports where the member weights $(Y_1, Y_2, ..Y_P)$ for $G\prime$ are obtained by solving the following optimization problem.

$$\begin{aligned}\text{minimize} \quad & \sum_{i=1}^{P} Y_i \\ \text{subject to} \quad & \Delta(G,G\prime) \leq \theta_{max} \\ & X_i,\ Y_i \text{ are +ve integers } (1 \leq i \leq P)\end{aligned} \tag{3.3}$$

Algorithm 1. *ReduceWcmpGroup(G, θ_{max}).* Returns a smaller group G' such that $\Delta(G,G') \leq \theta_{max}$

```
1:  for $i = 1$ to $P$ do
2:      $G'[i].weight = 1$
3:  end for
4:  while $\Delta(G,G') > \theta_{max}$ do
5:      $index = ChoosePortToUpdate(G,G')$
6:      $G'[index].weight = G'[index].weight + 1$
7:      if $G'.size \geq G.size$ then
8:          return G
9:      end if
10: end while
11: return $G'$
```

Algorithm 2. *ChoosePortToUpdate(G, G′).* Returns the index of member port whose weight should be incremented to result in least maximum oversubscription.

```
$min\_oversub = INF$
$index = -1$
for $i = 1$ to $P$ do
    $oversub = \frac{(G'[i].weight+1) \cdot G.size}{(G'.size+1) \cdot G[i].weight}$
    if $min\_oversub > oversub$ then
        $min\_oversub = oversub$
        $index = i$
    end if
end for
return $index$
```

This is an *Integer Linear Programming (ILP)* problem in variables Y_is, known to be NP-complete [21]. Though the optimal solution can be obtained by using well known linear programming solutions, finding that optimal solution can be time-intensive, particularly for large topologies. Hence, we propose an efficient greedy algorithm that gives an approximate solution to the optimization problem in Algorithm 1.

We start by initializing each Y_i in G' to 1. G' is the smallest possible WCMP group with the same member ports as G (Lines 1-3). We then increase the weight of one of the member ports by 1 by invoking Algorithm 2 and repeat this process until either: (i) we find weights with maximum oversubscription less than θ_{max} or, (ii) the size of G' is equal to the size of the original group G. In the latter case, we return the original group G, indicating the algorithm failed to find a $G\prime$ with the specified θ_{max}.

Starting from the WCMP group with the smallest size, this algorithm greedily searches for the next smaller WCMP group with the least oversubscription ratio. For a WCMP group G with P ports and original size W, runtime complexity of the greedy algorithm is $O(P \cdot (W - P))$. To compare the results of the greedy algorithm with the optimal solution , we ran Algorithm 1 on more than 30,000 randomly generated WCMP groups and compared the results with the optimal solution obtained by running the GLPK LP solver [34]. The greedy solution was sub-optimal for only 34 cases (0.1%) and within 1% of optimal in all cases.

Weight Reduction with a Group Size Limit

Forwarding table entries are a scarce resource and we want to create a WCMP group that meets the constraint on the table size. Formally, given a WCMP group G with P member ports and multipath group size T, the weights for WCMP group $G\prime$ can be obtained by solving the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \Delta(G, G\prime) \\ \text{subject to} \quad & \sum_{i=1}^{P} Y_i \leq T \\ & X_i,\ Y_i \text{ are +ve integers } (1 \leq i \leq P) \end{aligned} \tag{3.4}$$

Programming forwarding table entries is a bottleneck for route updates [23]. Reducing the number of forwarding table entries has the added benefit of improving routing convergence as it reduces the number of entries to be programmed. We refer to the above optimization problem as *Table Fitting*. This is a *Non-linear Integer Optimization* problem because the objective function $\Delta(G, G\prime)$ is a non-linear function of the variables Y_is, which makes it harder to find an optimal solution to this problem.We present a greedy algorithm for the Table Fitting optimization in Algorithm 3.

We begin by initializing $G\prime$ to G. Since the final weights in $G\prime$ must be positive integers, ports with unit weight are counted towards the *non_reducible_size* as their weight cannot be reduced further. If fractional weights were allowed, reducing weights in the same ratio as that between the size limit T and original size will not result in any oversubscription. However, since weights can only be positive integers, we round the weights as shown in Lines 14-16 of Algorithm 3. It is possible that after rounding, the size of the group exceeds T, since some weights may be rounded up to 1. Hence, we repeatedly reduce weights until the size of the group is less than T. We do not allow zero weights for egress ports because that may lower the maximum throughput of the original WCMP group. In some cases it is possible that the size of the reduced group is strictly less than T, because of rounding down (Line 14). In that case, we increase the weights of the WCMP group up to the limit T, in the same manner as in Algorithm 1 to find the set of weights that offer the minimum $\Delta(G, G\prime)$ (Lines 20-30).

For a WCMP group with size W and P member ports, the runtime complexity of Algorithm 3 is $O(P \cdot (W - T) + P^2)$. If high total bandwidth is not a requirement and zero weights are allowed, we can further optimize this algorithm by rounding fractional weights to zero and eliminate the outer while loop, with final runtime complexity as $O(P^2)$.

Weight Reduction for Multiple WCMP Groups

For reducing weights given a maximum oversubscription limit for a group for WCMP groups H, we can simply run Algorithm 1 independently for each WCMP group in H. However, for reducing weights given a limit on the total table size for H, the exact size limit for individual WCMP groups is not known. One alternative is to set the size limit for each group in H in proportion to the ratio between the total size limit for H and the original size of H and then run Algorithm 3 independently on each group in H. While this approach ensures that groups in H fit the table size S, it does not strive to find the minimum $\Delta(G, G\prime)$ of member groups that could be achieved by changing the individual size constraints on each WCMP group in H. For that, we present Algorithm 4 that achieves weight reduction for a set of WCMP groups by linearly searching for the lowest maximum oversubscription limit.

Algorithm 4 shows the pseudo code for weight reduction across groups in H. We begin with a small threshold for the maximum oversubscription θ_c, and use it with Algorithm 1 to find smaller WCMP groups for each of the member groups in H. We sort the groups in H in descending order of their size and begin by reducing weights for the largest group in H. We repeatedly increase the oversubscription threshold (by a step size ν) for further reduction of WCMP groups until their aggregate size in the multipath table drops below S. Since the algorithm progresses in step size of ν, there is a trade-off between the accuracy of the oversubscription limit and the efficiency of the algorithm, which can be adjusted as desired by changing the value of ν.

3.4 Striping

Striping refers to the distribution of uplinks from lower stage switches to the switches in the successive stage in a multi-stage topology. The striping is crucial in

Algorithm 3. *TableFitting(G, T).* WCMP Weight Reduction for Table Fitting a single WCMP group G into size T.

```
$G' = G$
while $G'.size > T$ do
    $non\_reducible\_size = 0$
    for $i = 1$ to $P$ do
        if $G'[i].weight = 1$ then
            $non\_reducible\_size += G'[i].weight$
        end if
    end for
    if $non\_reducible\_size = P$ then
        $break$
    end if
    $reduction\_ratio = \frac{(T - non\_reducible\_size)}{G.size}$
    for $i = 1$ to $P$ do
        $G'[i].weight = \lfloor (G[i].weight \cdot reduction\_ratio) \rfloor$
        if $G'[i].weight = 0$ then
            $G'[i].weight = 1$
        end if
    end for
end while
$remaining\_size = T - G'.size$
$min\_oversub = \Delta(G, G')$
$G'' = G'$
for $k = 1$ to $remaining\_size$ do
    $index = ChoosePortToUpdate(G, G')$
    $G'[index].weight = G'[index].weight + 1$
    if $min\_oversub > \Delta(G, G')$ then
        $G'' = G'$
        $min\_oversub = \Delta(G, G')$
    end if
end for
return $G''$
```

Algorithm 4. *TableFitting(H, S).* WCMP Weight Reduction for Table Fitting a set of WCMP groups H into size S.

```
$\theta_c = 1.002$ // $\theta_c$: enforced oversubscription
$\nu = 0.001$ // $\nu$: step size for increasing the $\theta_c$
// Sort the groups in $H$ in descending order of size.
$H\prime = H$
while $TotalSize(\{H\prime\}) > S$ do
    for $i = 1 ... NumGroups(\{H\})$ do
        $H\prime[i] = ReduceWcmpGroup(H[i], \theta_c)$
        if $TotalSize(\{H\prime\}) \leq S$ then
            return
        end if
    end for
    $\theta_c + = \nu$
end while
return $H\prime$
```

determining the effective capacity for different paths and hence the relative weights for the different ports in a WCMP group. While weight reduction algorithms reduce weights for a WCMP group, the striping determines the original weights in a WCMP group. Balanced stripings are desirable in data center networks so as to deliver uniform and maximum possible throughput among all communicating switch pairs. However, maintaining balanced striping is impractical due to its strict requirement that the number of upper links in a switch must be an integer multiple of the number of upper layer switches. It is practically unsustainable due to frequent node and link failures. In this section, we attempt to identify the design goals for topology striping and propose striping alternatives that illustrate trade-offs between different goals.

3.4.1 Striping Alternatives

We use the 2-stage network topology depicted in Figure 3.1 in describing striping alternatives. The class of 2-stage Clos networks is characterized by the following parameters, assuming all physical links have unit capacity:

K: Number of stage-2 (S2) switches.

D: Number of downlinks per S2 switch.

L: Number of stage-1 (S1) switches.

N: Number of uplinks per S1 switch.

The striping is imbalanced when N cannot be divided by K. In particular, one S1 switch may be connected to an S2 switch with more uplinks than another S1 switch.While a naïve approach to build fully balanced topologies is to remove the additional uplinks at a switch, this will lower the maximum throughput between pairs of switches that can be connected symmetrically to the upper stage switches. For example in Figure 3.5, even though $S1_0$ and $S1_2$ are asymmetrically connected to the S2 switches, switches $S1_0$ and $S1_1$ are connected symmetrically and can achieve higher throughput as compared to the case where the extra links were removed to build a fully symmetrical topology.

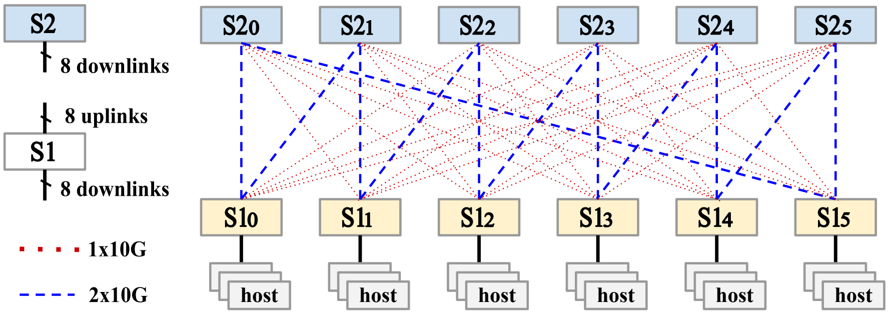


Figure 3.4. Rotation Striping

In order to be as close to balanced striping, we distribute the set of N uplinks from an S1 switch among S2 switches as evenly as possible. We first assign $\lfloor \frac{N}{K} \rfloor$ uplinks to each of the K S2 switches and the remaining N - K.($\lfloor \frac{N}{K} \rfloor$) uplinks at the S1 switch are distributed across a set of N - K.($\lfloor \frac{N}{K} \rfloor$) S2 switches. This strategy ensures that each S1 switch is connected to an S2 switch with at most 1 extra uplink as compared to other S1 switches connected to the same S2 switch. Formally we can represent this striping

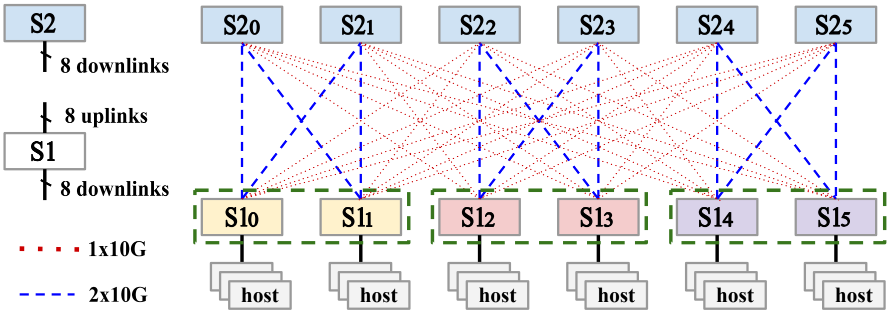


Figure 3.5. Group Striping

strategy using a connectivity matrix R, where each element R_{jk} is the number of uplinks connecting $S1_j$ to $S2_k$. Let p = $\lfloor \frac{N}{K} \rfloor$, then

$$R_{jk} = p \text{ or } p+1, \tag{3.5}$$

The maximum achievable throughput may vary among different pairs of S1 switches based on the striping. The highest throughput between a pair of switches is achieved when each S1 switch has an equal number of uplinks to all S2 switches. The maximum achievable throughput between a pair of switches is lower when the pair have asymmetric striping to the S2 switches. Based on this observation, we derived two alternative striping options: (i) *Rotation Striping* and (ii) *Group Striping*, which illustrate the trade-off between improving the mean throughput versus improving the maximum achievable throughput across the different S1 switch pairs.

Figure 3.4 depicts a 2-stage Clos topology with 6 S1 switches and 6 S2 switches using *rotation striping*. For any pair of S1 switches, there is at least one oversubscribed S2 switch connected asymmetrically to the two S1 switches. Thus the maximum possible throughput for traffic between these switches is less than their total uplink bandwidth, even without competing traffic. The rotation striping can be generated by connecting a switch $S1_i$ with a contiguous set of π_p S2 switches with p uplinks each starting from

$S2_{i(mod)K}$, and the remaining π_{p+1} S2 switches with $p+1$ uplinks.The derivations of π_p and π_{p+1} are shown below.

Algorithm 5. Generating group striping - Phase I

```
$\Omega_{p+1} = D - L \cdot p$
$\Omega_p = L - \Omega_{p+1}$
$\Omega = min(\Omega_{p+1}, \Omega_p)$
$\pi = min(\pi_p, \pi_{p+1})$
if $\Omega = 0$ then
    $Q = 1$
else
    $Q = \lfloor \frac{L}{\Omega} \rfloor - \lceil \frac{L\%\Omega}{\Omega} \rceil$
end if
if $\Omega = \Omega_{p+1}$ then
    $R_{lk} = p$ for $\forall\ l < L, k < K$
else
    $R_{lk} = p + 1$ for $\forall\ l < L, k < K$
end if
for $i = 0$ to $Q - 1$ do
    for $l = i \cdot \Omega$ to $i \cdot \Omega + \Omega - 1$ do
        for $k = i \cdot \pi$ to $i \cdot \pi + \pi - 1$ do
            if $\Omega = \Omega_{p+1}$ then
                $R_{lk} = p + 1$
            else
                $R_{lk} = p$
            end if
        end for
    end for
end for
```

π_p: the number of S2 switches connected to an S1 switch with p downlinks.

π_{p+1}: the number of S2 switches connected to an S1 switch with $p+1$ downlinks.

$$\begin{aligned} \pi_{p+1} &= N - K \cdot p \\ \pi_p &= K - \pi_{p+1} = K \cdot (p+1) - N \end{aligned} \tag{3.6}$$

Algorithm 6. Generating group striping - Phase II

```
$shift = 0$
for $l = Q \cdot \Omega$ to $L - 1$ do
    for $offset = 0$ to $\pi - 1$ do
        $k = \pi \cdot Q + ((offset + shift) \% (K - Q \cdot \pi))$;
        if $\Omega = \Omega_{p+1}$ then
            $R_{lk} = p + 1$
        else
            $R_{lk} = p$
        end if
    end for
    $shift += N/D$;
end for
```

Figure 3.5 depicts the *group striping* for the same topology. Three pairs of S1 switches have identical uplink distributions and can achieve maximum possible throughput, 80 Gb/s. However, compared to the rotation striping, there are also more S1 switch pairs with reduced maximum achievable throughput, 60 Gb/s. Rotation striping reduces the variance of network capacity while group striping improves the probability of achieving ideal capacity among S1 switches.

The algorithm for generating the group striping runs in two phases as shown in Algorithm 5 and 6. We first create Q (Q = 3 in Figure 3.5) sets of S1 switches such that S1 switches within each set can achieve maximum throughput for a destination S1 switch in the same set. In the second phase, the algorithm generates the striping for the remainder of S1 switches. Each of these S1 switches have a unique connection pattern to the S2 switches.

3.4.2 Link Weight Assignment

WCMP is a generic solution that makes no assumption about the underlying topology for assigning and reducing weights for links. For arbitrary topologies, the link weights can be computed by running max-flow min-cut algorithms of polynomial complexity and creating WCMP groups per source-destination pair. The weight reduction algorithm proposed in the section 3.3 would similarly reduce the number of table entries in such settings. With its efficient algorithms for weight assignment and weight reduction, WCMP can react quickly to changes in the topology due to inherent heterogeneity or failures and reprogram updated WCMP groups in just a few milliseconds.

In the topology shown in Figure 3.1, for any switch $S1_s$, its traffic flows destined to a remote switch $S1_d$ can be distributed among its N uplinks. Given: i) the varying effective capacities and ii) delivering uniform throughput among flows as the driving objective, more flows should be scheduled to uplinks with higher capacity than those with lower capacities. Therefore, when installing a WCMP group on $S1_s$ to balance its traffic to $S1_d$ among its N uplinks, we set the weight of each uplink proportional to its effective capacity which is either 1 or $\frac{p}{p+1}$.

3.5 Failure Handling

WCMP relies on the underlying routing protocol to be notified of switch/link failures and change the weights based on the updates. As such, WCMP is limited by the reaction time of the underlying protocol to react to changes in the topology.When a link failure is reported, the only additional overhead for WCMP is in recomputing the weights and programming the multipath table entries with the updated weights in the forwarding table. Since our weight reduction algorithms scale quadratically with number of ports in a group and are very fast for switches with port counts up to 96 ports, this overhead is extremely small.

Our simulations indicate that the weight computation and reduction algorithms can recompute weights in under 10 milliseconds for topologies with 100,000 servers when notified of a topology update. However, failures like link flapping may indeed result in unnecessary computations. In that case, WCMP will disable the flapping port

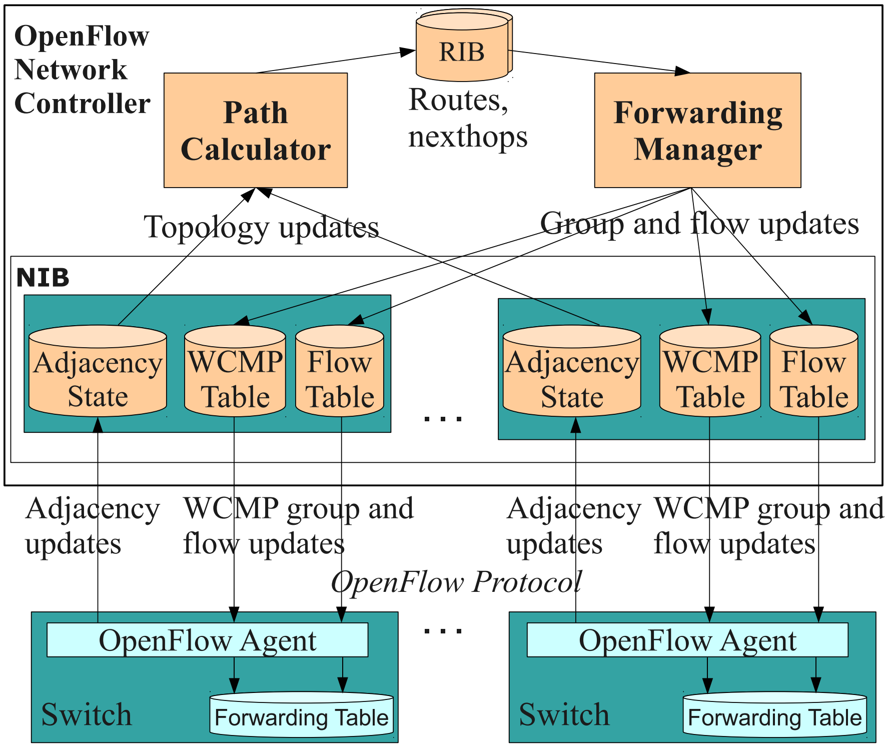


Figure 3.6. WCMP Software Architecture

from the multipath group in order to avoid unnecessary weight computations. We also note that while a series of failures may result in recomputing (and reducing) weights for WCMP groups at a large number of switches, in general most failures will only require updating weights for only a small number of WCMP groups at few switches. The cost of updating weights is incurred only rarely but once the right weights are installed it results in significantly improving the fairness for both short and large flows in the network.

3.6 System Architecture

We base our architecture and implementation for supporting WCMP around the Onix OpenFlow Controller[56] as shown in Figure 3.6. The Network Controller is the central entity that computes the routes between switches (hosts) and the weights for distributing the traffic among these routes based on the latest view of network topology. It also manages the forwarding table entries in the switches. We assume a reliable control channel between the network controller and the switches, e.g., a dedicated physical or logical control network. We implement the Network Controller functionality through three components: Network Information Base (NIB), Path Calculator and Forwarding Manager.

NIB: This component discovers the network topology by subscribing to adjacency updates from switches. Switch updates include discovered neighbors and the list of healthy links connected to each neighbor. The NIB component interacts with the switches to determine the current network topology graph and provides this information to the path calculator. The NIB component also caches switch's flows and WCMP group tables and is responsible for propagating changes to these tables to switches in case of topology changes due to failures or planned changes. In our implementation, all communication between NIB and switches is done using the OpenFlow protocol[66]. An OpenFlow agent running on each switch receives forwarding updates and accordingly programs the hardware forwarding tables.

Path Calculator: This component uses the network topology graph provided by the NIB and computes the available paths and their corresponding weights for traffic distribution between any pair of S1 switches. The component is also responsible for all other routes, e.g., direct routes to hosts connected to an S1 switch.

Forwarding Manager: This component manages the flow and WCMP tables. It converts the routes computed by the path calculator into flow rules, and next hops into WCMP groups to be installed at the switches. Since there are only limited hardware

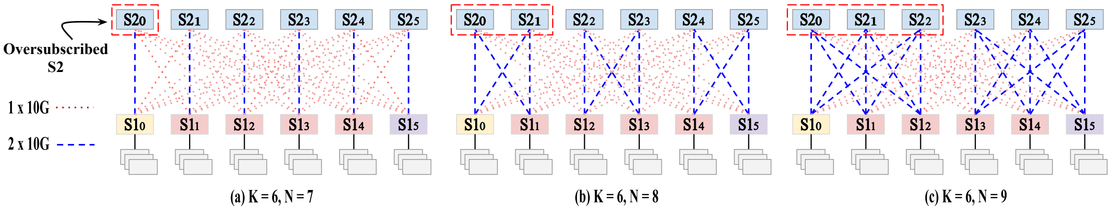


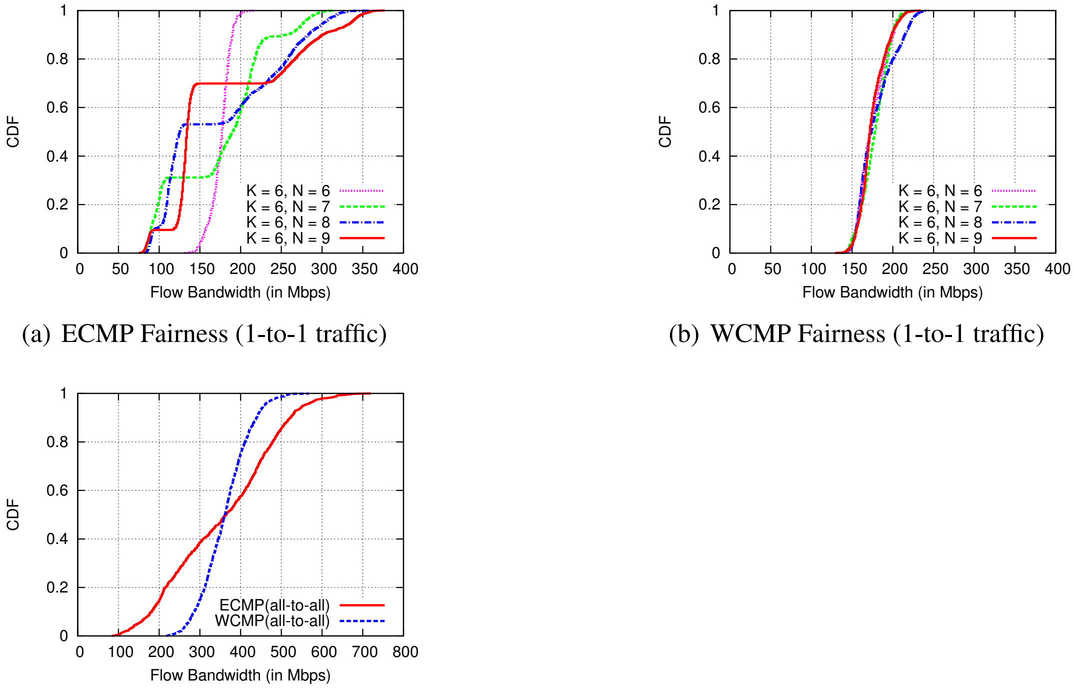
Figure 3.7. Testbed topologies (with group striping) with increasing imbalance for one-to-one traffic pattern

entries at each switch, this component also computes reduced weights for links such that WCMP performance with reduced weights is within tolerable bounds of the performance for the weights computed by the path calculator component. This optimization could also be implemented on the switch side.

Upon start-up, the Path Calculator queries the NIB for the initial topology and computes the routes and next hops. The Forwarding Manager converts the computed routes and next hops to flow rules and WCMP groups. Once the weights are computed, it invokes weight reduction algorithm for different switches in parallel and installs the weights into the switches. As illustrated in Figure 3.6, the Path Calculator further receives topology change events from the NIB (link up /down events), and recomputes next hops upon such events as link/switch failure or topology expansion. The updated next hops are then converted to updated WCMP groups to be installed on individual switches. The Path Calculator and the Forwarding Manager components together consist of 3K lines of C++. We use previously developed Onix and OpenFlow extensions for multipath support (OpenFlow 1.1 specification supports multipath groups).

3.7 Evaluation

We evaluate WCMP using a prototype implementation and simulations. The prototype cluster network consists of six non-blocking 10Gb/s S1 switches interconnected by six nonblocking 10Gb/s S2 switches and 30 hosts with 10G NICs. Each host is running



(a) ECMP Fairness (1-to-1 traffic)

(b) WCMP Fairness (1-to-1 traffic)

(c) ECMP, WCMP Fairness Comparison (all-to-all traffic)

Figure 3.8. Comparing ECMP, WCMP performance on Clos topologies with different imbalance, different traffic patterns

Linux kernel version 2.6.34 with ECN enabled in the TCP/IP stack. Our switches support OpenFlow and ECN, marking packets at 160KB for port buffer occupancy.

We evaluate TCP performance with ECMP and WCMP hashing for topologies with different degrees of imbalance and for different traffic patterns including real data center traffic patterns from [13]. We extended the *htsim* simulator [47] for MPTCP to support WCMP hashing and evaluate WCMP benefits relative to MPTCP. We also measure the effectiveness of our weight reduction algorithms and the impact of weight reduction on flow bandwidth fairness. Overall, our results show:

- WCMP always outperforms ECMP, reducing the variation in the flow bandwidths by as much as 25×.
- WCMP complements MPTCP performance and reduces variation in flow band-

width by 3× relative to the baseline case of TCP with ECMP.

- The weight reduction algorithm can reduce the required WCMP table size by more than half with negligible oversubscription overhead.

3.7.1 TCP Performance

We begin by evaluating TCP performance with ECMP and WCMP for different topologies and for different traffic patterns: one-to-one, all-to-all and real data center traffic.

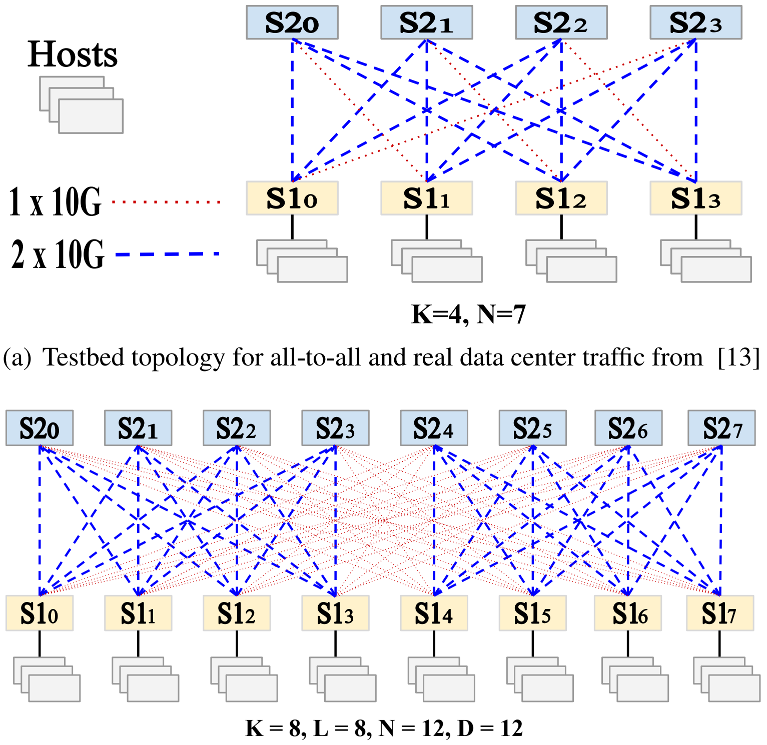
One-to-one Traffic: We first compare the impact of striping imbalance on TCP flow bandwidth distribution for ECMP and WCMP hashing. To vary the striping imbalance, we increase the number of oversubscribed S2 switches, to which a pair of S1 switches is asymmetrically connected. We manually rewire the topology using our group striping algorithm into the three topologies shown in Figure 3.7 and generate traffic between asymmetrically connected switches $S1_0$ and $S1_5$. Each of the nine hosts connected to source switch $S1_0$ transmit data over four parallel TCP connections to a unique host on destination switch $S1_5$ over long flows. We plot the CDF for the flow bandwidths in Figure 3.8.

Figure 3.8(b) shows that WCMP effectively load balances the traffic such that all flows receive almost the same bandwidth despite the striping imbalance. Bandwidth variance for ECMP on the other hand increases with the striping imbalance as shown in Figure 3.8(a). We make the following observations from these CDFs: (1) For ECMP, the number of *slower* flows in the network increases with the striping imbalance in the topology. More importantly, for imbalanced topologies, the minimum ECMP throughput is significantly smaller than the minimum WCMP throughput, which can lead to poor performance for applications bottlenecked by the slowest flow. (2) the variation in the flow bandwidth increases with the imbalance in the topology. WCMP reduces the variation in flow bandwidth by 25× for the topology where K=6, N=9. High variations in flow bandwidth make it harder to identify the right bottlenecks and also limit application performance by introducing unnecessary skew.

All-to-all Traffic: We next compare ECMP and WCMP hashing for all-to-all communication for the topology shown in Figure 3.9(a) and present the results in Figure 3.8(c). Each host communicates with hosts on all the remote switches over long flows. Again, the load balancing is more effective for WCMP than for ECMP. This graph provides empirical evidence that the weighted hashing of flows provides fairer bandwidth distribution relative to ECMP even when the traffic is spread across the entire topology. In this case, WCMP lowered variation in flow bandwidth by 4× and improved minimum bandwidth by 2×.

Real Data Center Traffic: We also compare ECMP and WCMP hashing for mapreduce style real data center traffic as measured by Benson et. al. [13]. We generate traffic between randomly selected inter-switch source-destination pairs for topology shown in Figure 3.9(a). The flow sizes and flow inter-arrival times have a lognormal distribution as described in [13].

Figure 3.10 shows the standard deviation (std. dev.) in the completion time of flows as a function of the flow size. Though both ECMP and WCMP are quite effective for small flows, for flow sizes greater than 1MB, the variation in the flow completion times is much more for ECMP compared to WCMP, even for flows of the same size. Moreover, this variation increases as the flow size increases.In summary, for real data center traffic, WCMP reduced the std. dev. in bandwidth by 5× on average and, more importantly, 13× at 95%-ile relative to ECMP while average bandwidth improved by



(a) Testbed topology for all-to-all and real data center traffic from [13]

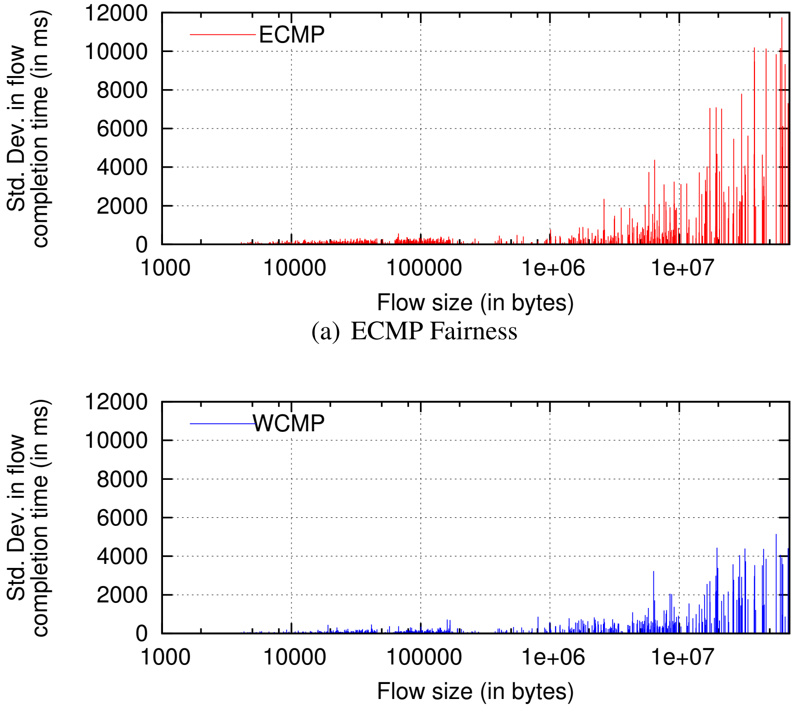
(b) Simulated topology using htsim

Figure 3.9. Evaluation topologies simulated using htsim simulator

20%.

3.7.2 MPTCP Performance

We extended the packet level MPTCP simulator, htsim to support WCMP hashing and evaluate its impact on MPTCP performance. The simulated topology consists of eight 24-port S1 switches (12 uplinks and 12 downlinks) and eight 12-port S2 switches (K=8, L=8, N=12, D=12) as shown in Figure 3.9(b). Each S1 switch is also connected to 12 hosts. We use 1000 byte packets, 1 Gb/s links, 100KB buffers and 100μs as per-hop delay. We use a permutation traffic matrix, where each host sends data to a randomly chosen host on a remote switch. We consider two scenarios (i) all flows start/stop at the same time, (on-off data center traffic pattern Figure 3.11(a)), (ii) flows start at different times, subjected to varying level of congestion in the network (Figure 3.11(b)). We



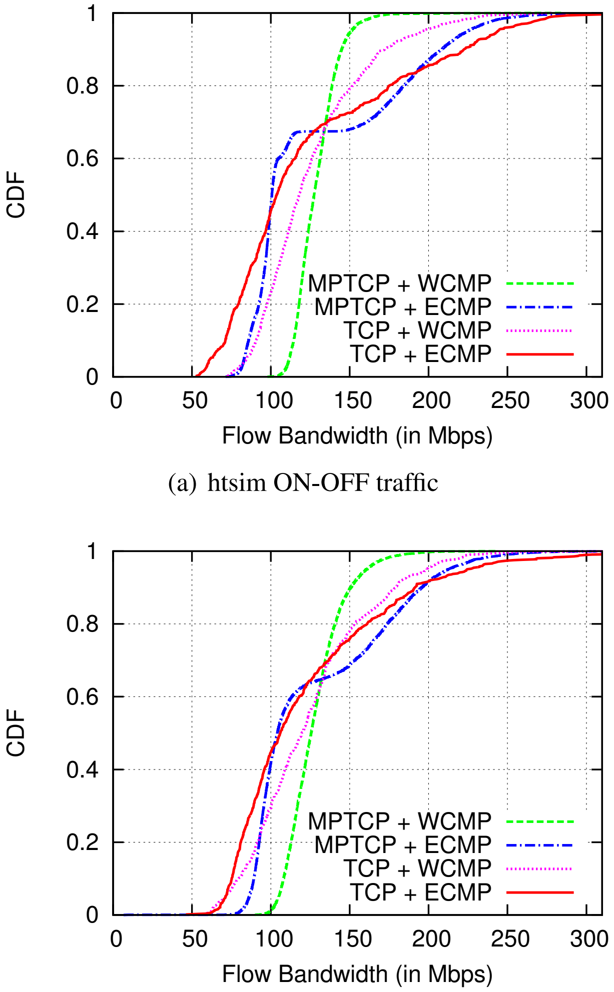
(a) ECMP Fairness

(b) WCMP Fairness

Figure 3.10. Comparing ECMP and WCMP performance for data center traffic measured by Benson et. al. [13]

evaluate all 4 possible combinations of TCP, MPTCP (with 8-subflows per TCP flow) with ECMP and WCMP.

The results in Figure 3.11(a) and 3.11(b) show that MPTCP with WCMP clearly outperforms all other combinations. It improves the minimum flow bandwidth by more than 25% and reduces the variance in flow bandwidth by up to 3× over MPTCP with ECMP. While WCMP with TCP outperforms ECMP with TCP for the on-off communication pattern, it has to leverage MPTCP for significant improvement for the skewed traffic patterns. This is because MPTCP can dynamically adjust the traffic rates of subflows to avoid hot spots while WCMP is useful for addressing the bandwidth variation due to structural imbalance in the topology.



(a) htsim ON-OFF traffic

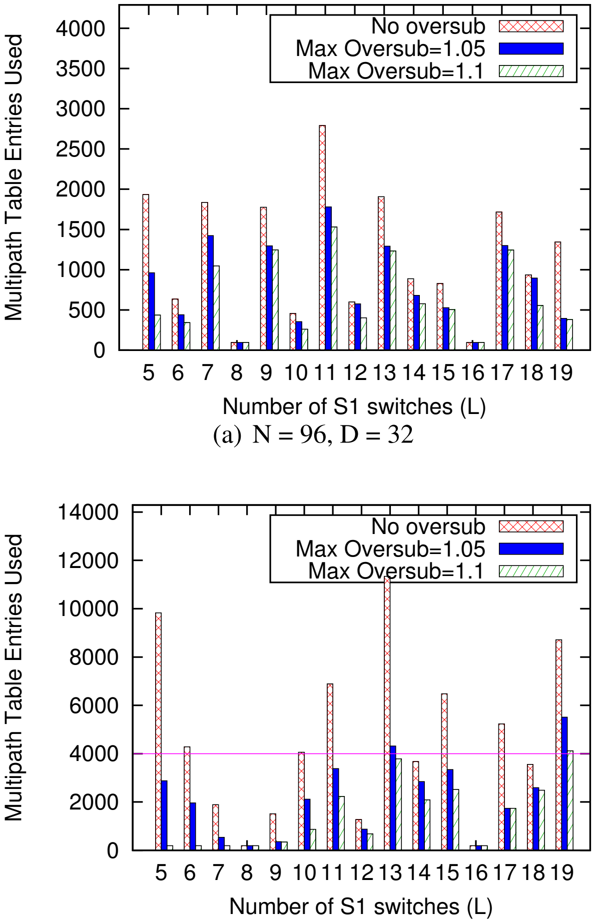
(b) htsim skewed traffic

Figure 3.11. Evaluating MPTCP performance with WCMP

3.7.3 Weight Reduction Effectiveness

Next, we evaluate the effectiveness of our weight reduction algorithms. We analyze two topologies, with the number of S1 uplinks (N) equal to 96 and number of S2 downlinks (D) as 32 and other where N = 192 and D = 64. We vary the number of S1 switches from 5 to 19 and compute the maximum number of table entries required at an S1 switch.

We run Algorithm 1 to reduce this number with different oversubscription limits



(a) N = 96, D = 32

(b) N = 192, D = 64

Figure 3.12. Table entries before and after running Algorithm 1 for reducing WCMP groups at the switch requiring maximum table entries

(1.05 and 1.1) and show the results in Figure 3.12. Without any weight reduction, the maximum number of multipath table entries at the switch is $>$10k for the case where S1 switches have 192 uplinks, while the size of multipath tables on commodity switches is usually only 4K. The reduction algorithm reduces the required multipath table entries by more than 25% while incurring only 1.05 oversubscription (Figure 3.12(b)). It can further fit the entire set of WCMP groups to the table of size 4K at the maximum oversubscription of only 1.1. We also ran the LP solver for reducing the size of the WCMP groups at the S1 switch requiring maximum table entries. In all cases, the results from our weight reduction algorithm were same as the optimal result. Figure 3.12 further shows that without the reduction algorithm, the maximum number of table entries grows by almost 3× when the switch port counts were doubled. The reduction algorithm significantly slows such growth with limited impact on the fairness.

3.7.4 Weight Reduction Impact on Fairness

Our weight reduction algorithms trade-off achieving ideal fairness in order to create WCMP groups of smaller size. Since our evaluation testbed was small and did not require weight reduction, we simulate a large topology for evaluating weight reduction impact on fairness. We instantiate a topology with 19 S1 switches with 96 uplinks each, 57 S2 switches with 32 downlinks each and 1824 hosts using htsim (K=57, L=19, N=96, D=32). With group striping for this topology, we have two groups of six S1 switches, and each S1 switch in a group has identical striping to the S2 switches. The remaining seven S1 switches are asymmetrically connected to all the other S1 switches. We run Algorithm 1 with different oversubscription limits to create WCMP groups with reduced weights. We generate traffic using a random permutation traffic matrix, where each host sends data to another host on a remote switch with long flows. Figure 3.13 shows the results of this experiment. With a oversubscription limit of 1.05, we achieve a 70%

reduction in the maximum number of multipath table entries required (Figure 3.12(a)) with very little impact on the fairness. As we increase the oversubscription limit to 1.1, the fairness reduces further, but WCMP still outperforms ECMP.

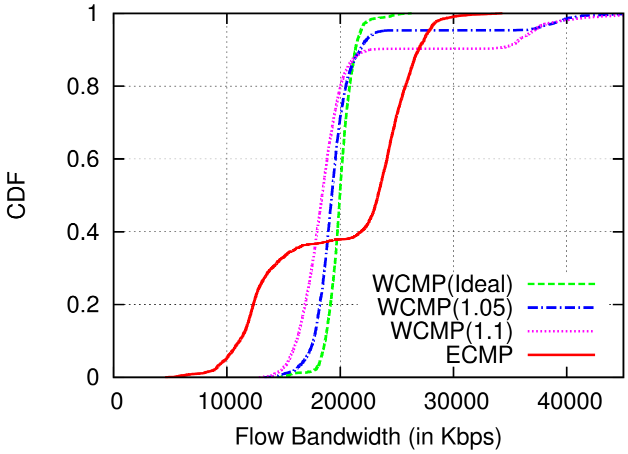


Figure 3.13. Impact of weight reduction on fairness

3.8 Conclusion

Existing techniques for statically distributing traffic across data center networks evenly hash flows across all paths to a particular destination. We show how this approach can lead to significant bandwidth unfairness when the topology is inherently imbalanced or when intermittent failures exacerbate imbalance. Such inherent topology imbalance means that the amount of bandwidth available to a destination varies among the available next hop paths and can result in unfair bandwidth distribution across flows.

We present WCMP for weighted flow hashing across the available next-hops to the destination and show how to deploy our techniques on existing commodity silicon and improve network maintenance/diagnosis. Our performance evaluation on an OpenFlow-controlled network testbed shows that WCMP can substantially reduce the performance difference among flows compared to ECMP, with the potential to improve application performance and network diagnosis; and complements dynamic solutions like MPTCP for better load balancing as it makes balanced capacity available through the fabric as a function of current topology.

Chapter 3, in full, is a reprint of the material from WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. Zhou, Junlan; Tewari, Malveeka; Zhu, Min; Kabbani, Abdul; Poutievski, Leon; Singh, Arjun; Vahdat, Amin. In Proceedings of the Ninth European Conference on Computer Systems (EuroSys), April, 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Tightly Coupled End Host Flow Management

4.1 Introduction

The performance of cloud computing is increasingly dependent on the performance of the underlying network. Numerous recent proposals aim to improve network performance by imposing greater control over data sources: D^3 [83] improves upon TCP by strictly controlling the rate at which end hosts inject traffic into the network. QCN [72] enables congestion control by adjusting end-host sending rates through a low-latency, in-band control plane. pFabric [8] segregates flows within an end host to ensure conflict-free forwarding within the network fabric. FastPass [68] and REACToR [58] enforce TDMA link sharing among sources to support buffer-less electrical or optical interconnects. In each of these cases, it is the source host that is responsible for transmitting packets at the appropriate times and rates to implement the desired data plane functionality.

This level of source-based control is a departure from the traditional network model, where hosts are insulated from the details of the network by strong abstractions. Historically, the operating system (OS) at the source server chooses which packets to send based on an entirely decentralized scheduling discipline. It is then up to the network to buffer packets on their way to their destination. Yet, as mentioned above, further improvements necessitate eroding that abstraction, forcing the end host to be a much more active participant in network-wide scheduling. But for that to occur, the end-host data plane must be meet two requirements. It must be (1) programmable, performant, and precise, and (2) able to work in coordination with network switches and centralized controllers.

4.2 Optics in Data Centers

Electrical packet interconnects switch rapidly, potentially choosing a different destination for each packet. They are well suited for bursty and unpredictable communication. Optical interconnects, on the other hand, can provide higher bandwidth over longer distances (any 10Gbps links longer than about 10 meters must be optical). Optical switches, however, cannot be reconfigured rapidly: the MEMS-based optical circuit switches we use take between 10 and 25ms to reconfigure. As a result of these properties, the two network types offer substantially different advantages, with optics being superior for long-lasting, very high bandwidth point-to-point flows, and electronics winning for bursty, many-to-many local communication.

HyPaC networks aim to achieve the best of both worlds by using these two interconnects together in the same network. Typically, these networks first connect electrically a group of nodes (a “pod”) to a set of “pod switches”. The pod switches connect to a core packet switch at a high degree of over-subscription. The pod switches also connect several uplinks to an optical MEMS switch, which can configure a matching of pod-to-pod links. At any time, nodes in one pod can communicate with limited bandwidth to *any* node in the network using the core packet switch, and with nodes in a few other selected pods at high bandwidth using the optical switch. The optical circuit bandwidth between pods is both less expensive and less flexible than packet-switched bandwidth. A key problem for both c-Through and Helios thus becomes how to schedule these circuits: For example, given four pods, each with one uplink to the circuit switch, should the system connect 1-2 and 3-4, or 1-3 and 2-4?

Both c-Through and Helios estimate the network traffic demand, identify pods that have long-lived, stable aggregated demand between them, and establish circuits between them to amortize the high cost of switching circuits.

4.3 Challenges

In this section we list certain key factors that make adoption of optical switching in data centers challenging.

Synchonizing end host with network controller: In order to efficiently enforce centralized schedules at the end host, it is important that the end host and fabric manager are synchonized. This requires synchronizing times for the network and the end hosts, measuring the demand for different applications running on the end hosts, computing the schedules and finally notifying the servers to follow the desired schedules for packet transfers.

Inaccurate demand information: From the network perspective, it is very hard to have the precise knowledge about application traffic demand and performance requirement. Existing systems have tried to infer application traffic demands from various counters in the network, but as we have shown, these heuristics could result in flipping circuits and sub-optimal network performance. A good circuit allocation mechanism must be able to tolerate inaccurate measurement of application traffic demands.

Application dynamics: Data center applications are highly dynamic. Applications could be started and stopped at different time. Even within a single application, traffic sending to different destinations could be started at any time. In a HyPaC network, the reconfiguration of the network make the system dynamics even worse. When a circuit is setup between certain racks, applications could react to the network bandwidth changes

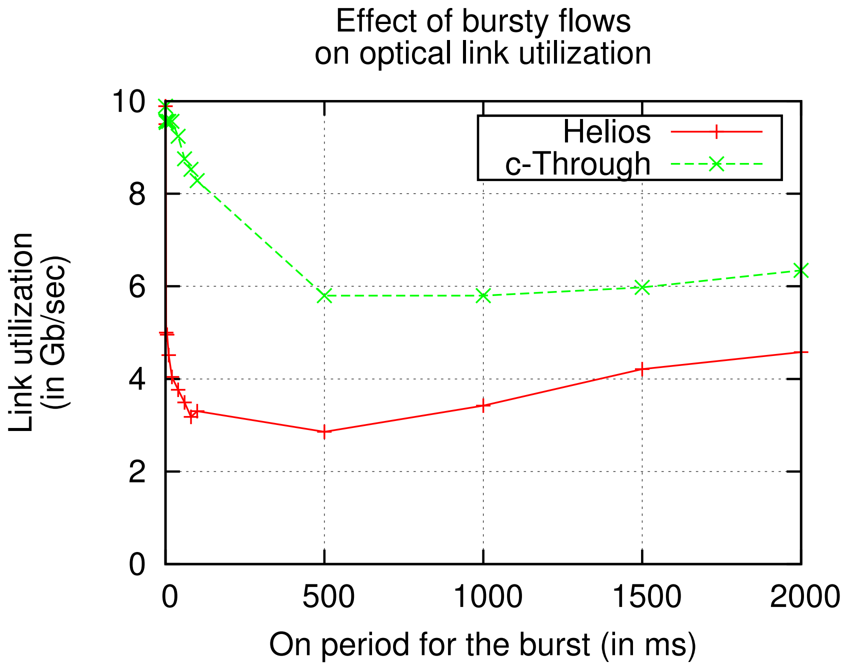


Figure 4.1. The utilization of a circuit in the presence of a bursty flow. By attempting to schedule the bursty flow on the circuit, bandwidth is reduced for the long-lived foreground flow.

and change its traffic pattern. The dynamic behavior is hard to predict due to the complicated policies and dependencies among application components. If we configure the optical circuits based on instaneous reading of traffic demand, the traffic patterns changes could cause sub-optimal circuit utilizaiton. Therefore, the circuit allocator should be able to adapt to unpredictable changes in the workload, but still achieve good utilization of optical circuits.

Interference among heterogeneous applications: Allocating circuit among mixed applications are challenging given the diversity of data center applications. Applications may have very different traffic patterns and performance requirements. Some applications may require fast completion time, but others may require stable and guaranteed bandwidth. In addition, the management policies in data centers could assign applications with different priorities. To share the limited number of optical circuits

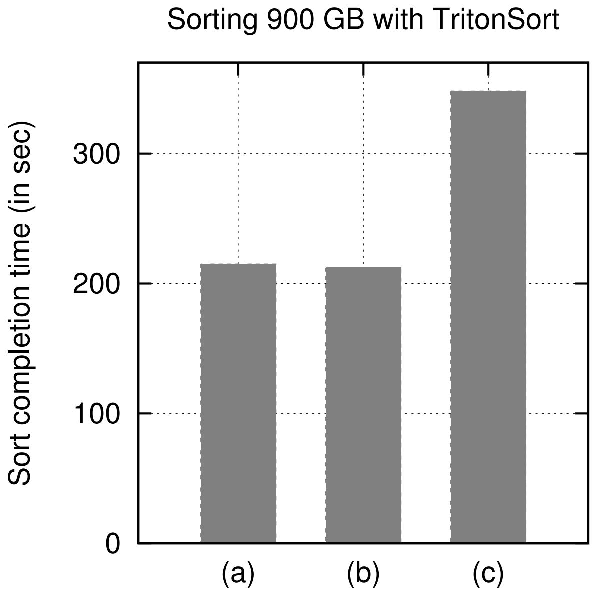


Figure 4.2. TritonSort completion time in three cases: (a) Fully non-blocking electrical switch (baseline case) (b) Helios/c-Through network with no background flows (c) Helios/c-Through network with one competing background flow

among these applications, the circuit allocator must be able to handle the performance interference among applications. Certain definition of fairness and priority supports are desired.

We demonstrate this problem using a topology of three pods (Pod 0, 1 and 2), with two hosts each. Each pod switch has one 5 Gb/s uplink to the core packet switch and one 10 Gb/s uplink to the optical circuit switch. In this topology, only two pods can be connected optically at any given time. For the duration of the experiment, one host in Pod 1 sends data to a host in Pod 2 over a long-lived TCP connection (the “foreground flow”). The second host in Pod 1 sends data to a host in Pod 0 following a bursty ON-OFF pattern; we vary the burst ON-duration with the OFF-duration set to 2 second to observe the circuit scheduling decisions made by the Helios and c-Through circuit schedulers.

Figure 4.1 shows the average utilization of the optical link. In both designs, as the duration of the traffic bursts increases, the utilization of the link initially decreases and then increases as the bursts grow longer than 500ms. With short bursts, the circuit is assigned to the bursty flow between Pod 1 and Pod 0, but after assignment, this flow goes quiescent, under-utilizing the optical capacity. In the next control cycle, the circuit is assigned back to the long-lived foreground flow. The control cycle is hundreds of milliseconds; bursts shorter than the control loop will reduce utilization for part of the control loop cycle. Longer bursts use the optical circuit for a longer fraction of the time it is assigned, improving overall optical utilization. Notably, the optical link capacity is never saturated by the long flow because of the constant flapping of the circuit between the pods.

4.4 Data Plane Development Kit (DPDK)

There is an increasing need for extremely low latency networking solutions and more flexible control over network transmissions as the data center applications grow and become more complex. Historically, traditional core networks used custom hardware based on the need for performance and capabilities that standard off-the-shelf hardware could not provide. This left the intelligence of the network embedded in custom hardware and chipsets that were expensive to buy, difficult to manage, and slow to change. These non-virtualized, legacy solutions also kept businesses locked into dedicated, proprietary hardware, and inflexible legacy applications and architectures. However, the data center infrastructure relies on commodity hardware and as such there is a need to support low latency and flexible control within software in the data center end hosts.

DPDK is a set of libraries and drivers for fast packet processing. It was designed to run on any processors knowing Intel x86 has been the first CPU to be supported. Intel DPDK enables higher levels of packet processing throughput than what is achievable using the standard Linux kernel network stack. This optimized library gives application developers the ability to address challenging data plane processing needs, typically found in Telecom and networking workloads, all in software and on general purpose, Intel architecture-based processors. Figure 4.3 shows the high level DPDK architecture.

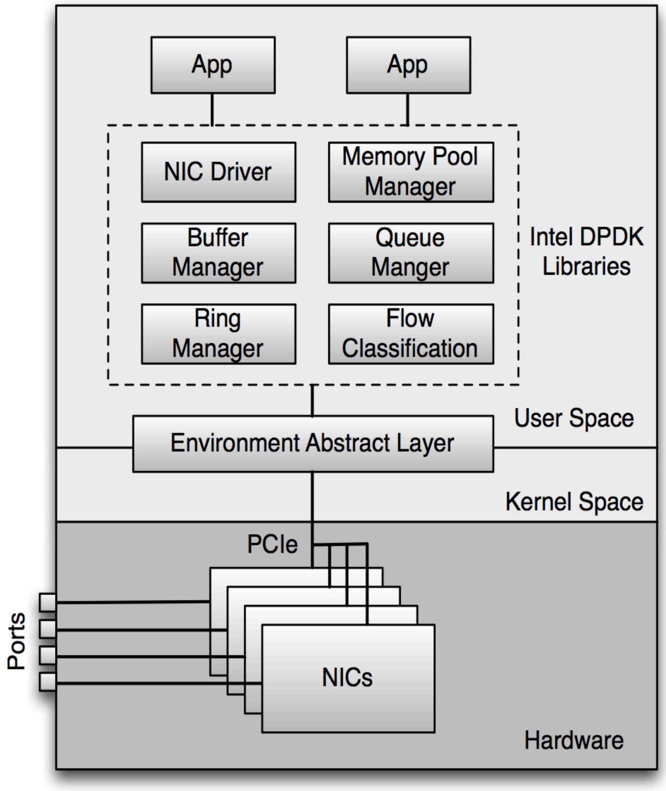


Figure 4.3. DPDK architecture

The development kit reduces a significant amount of overhead when using an out-of-the-box, standard Linux operating system. Significant time is saved by using core affinity, disabling interrupts generated by packet I/O, using cache alignment, implementing huge pages to reduce translation lookaside buffer (TLB) misses, prefetching, new instructions and many other concepts. The DPDK also runs in user space, thus removing the high overhead associated with kernel operations and with copying data between kernel and user memory space. Taking advantage of the breakthroughs enabled by the Intel DPDK, todays Intel architecture-based platforms based on a single Intel Xeon processor E5-2600 series are achieving over 80 million packets per second (Mpps) of L3 forwarding throughput for 64 byte packets [25].

DPDK makes use of huge memory pages and low level driver changes in order to support a new API for sending/recieving packets at high rates. While it deliver high performance, it fails to inter-operate with the traditional network stack requiring changes to existing applications to be able to leverage the DPDK framework. In this work, we use the DPDK framework to support high speed packet processing but also provide an abstraction layer for easy interoperability with existing data center applications.

4.5 Design Requirements

In order to meet the high resource utilization and application performance requirements, a fabric controller for a network with dynamic interconnects should be able to meet four key requirements:

Low-latency response overhead: It is difficult for a controller to have precise knowledge about all application's traffic demands and performance requirements. Existing systems infer these demands from counters in the network, but as we have shown, these heuristics can result in flapping circuits and sub-optimal network performance. A good circuit allocation mechanism must be robust to inaccurate measurement of application traffic demands.

Support correlated flows: To achieve good application layer performance, the circuit scheduling module must be able to accommodate flows whose demand and performance depends on the performance of another flow. The underlying framework supporting the scheduler must provide fine-grained control to handle differently traffic that is on the critical path of an application vs. less important flows. It should also provide an avenue for the controller to gather sufficient information to understand the application's dependence upon a flow's performance.

Adapt to application dynamics: Datacenter applications are highly dynamic.

Applications could be started and stopped at different time. Even within a single application, traffic sending to different destinations could be started at any time. In a HyPaC network, the reconfiguration of the network make the system dynamics even worse. When a circuit is setup between certain racks, applications could react to the network bandwidth changes and change its traffic pattern. The dynamic behavior is hard to predict due to the complicated policies and dependencies among application components. If we configure the optical circuits based on instant reading of traffic demand, the traffic patterns changes could cause sub-optimal circuit utilization. Therefore, the circuit allocator should be able to adapt to unpredictable changes in the workload, but still achieve good utilization of optical circuits.

Support flexible sharing policies among applications: Allocating circuits among mixed applications is challenging given the diversity of data center applications. Particularly in a multi-tenant cloud environment, applications may have very different traffic patterns and performance requirements. In addition, the management policies in data centers could assign applications different priorities. To share the limited number of optical circuits among these applications, the circuit allocator must be able to handle the performance interference among applications and support user-defined sharing policies among applications.

An Observe-Analyze-Act framework To achieve the above design requirements, the circuit controller must be able to obtain a detailed understanding of application semantics and fine-grain control of flows forwarding policies. We propose a three phase approach for managing HyPaC networks framework. To get a better understanding of the network dynamics and application heterogeneity in the cloud ecosystem, the HyPaC scheduler should be able to interact with different components and collect information from them. This information collected in this *Observe* phase would include the link utilization from the switches and application status from the cluster job schedulers (e.g., the Hadoop job Tracker), as well as application priorities and QoS requirements.

The HyPaC manager then analyzes the aggregation of this information to infer the most suitable configuration for the network. The *Analyze* phase is a key step that helps the network controller understand the application semantics of traffic demand, detect ill-behaved applications, discover the correlated flows and therefore make the optimal configurations to support these applications. Finally, in the *Act* phase, it communicates this configuration to the other components in the system in order for the decision to be acted upon. The *Act* phase requires fine-grain control on the flow forwarding to support flexible configuration decisions and sharing policies.

4.6 Design Alternatives

Right now, the end host network stack is optimized to (1) reduce the CPU load required to drive the network stack, and (2) keep the outgoing link high utilized while ensuring fairness between applications running on the host. The goal of this work is to design an end-host network stack that (1) is able to accurately enforce the link arbitration announcements it receives, and (2) use the announcements to somehow improve server efciency and performance. One set of research questions has to do with how precise the link arbitration announcements can be. For example, can the time range (t0; t1) be as small as O(10 μs)? The second, and probably more interesting question, is understanding to what extent we can use this new source of short-term future knowledge to improve system efciency and performance. For example, can we adjust the Linux process scheduler so that processes generate data just-in-time to send that data out according to the announcement? Put another way, to what extent can the performance and efciency of an end host be improved with future knowledge of exclusive access to the network link?

Historically networks have relied on statistical multiplexing, in which end hosts are free to send whatever data they like into the network. To prevent overload, a control loop, either at the transport layer (in the case of TCP) or at the Ethernet level (in the case of 802.1x or 802.1Qbb pause frames), pushes back in a reactive manner in the event of overload. The idea behind this project is to design a network stack for networks that do not rely on statistical multiplexing. Imagine instead that a central link arbitration controller arbitrates access to each link at ne- grained time scales, and transmits a link arbitration announcement to each host. This announcement divides the near-term future into time ranges, and for each time range, species a set of destinations to which the host can send data to (at whatever rate it wants to, e.g., at 100% of the link rate without any congestion control).

In this section, we motivate the need a new framework for supporting flow management by showing that existing OS control planes are not well suited for responding quickly to network-based control for exchanging the ”link arbitration” information.

We consider two alternative strawman approaches to flow management: (1) an OS-based packet handling data plane controlled by an OS-based control plane, and (2) a DPDK-based packet handling data plane controlled by the OS as well. In both cases, the network stack forwards messages from the central controller to the OS, which then adjusts rates, TDMA slots, prioritization, etc. We evaluate this indirect control over the data plane to understand the potential—and limitations—of having the OS “in the loop.”

4.6.1 OS Data Plane + OS Control Plane

We first consider employing the existing OS network stack to process both data traffic and control updates. While the Linux stack supports software rate limiters, there is no native mechanism to enforce TDMA-based sharing of a network link. Instead, one can implement a TDMA discipline by periodically updating rate limits for the different flows based on a schedule. During a flow's assigned time slot it is assigned a rate limit equal to the link rate; the remaining flows are assigned limits of zero. All limits are updated periodically as time slots progress. However, previous work implementing TDMA for data-center Ethernet [80] shows that such an implementation can lead to high variance in time slice duration. Moreover, other researchers have shown [73, 28] that software rate limiting fails to enforce precise rate limits as the number of flows increases or the transmission rate increases. Their experiments demonstrate that software-based rate limiting fails to achieve rates higher than 6 Gb/s. As such it is infeasible to use the existing OS data plane for supporting large number of rate limiters in software.

4.6.2 DPDK Data Plane + OS Control Plane

Next consider a DPDK-based data plane, yet an OS-based control plane implementing TDMA. DPDK can enforce precise TDMA slots, yet the OS is slow to respond to signals indicating the start and end of each slot, as shown in Figure 4.4.

While such an approach allows the end host to deliver high flow rates accurately, the control plane remains slow to respond to control updates and notifications. To demonstrate this, we implement a simple framework that uses a separate interface for receiving control updates which we use to send periodic control packets to indicate the start and end of scheduling period. Figure 4.4 shows the standard deviation in following a TDMA schedule for a single end host which sends traffic in an ON/OFF manner enforced by the TDMA schedule. We have normalized the standard deviation with the size of the TDMA slots. As shown in the figure, even for a single end host the standard deviation for small durations is significant to rule out this alternative.

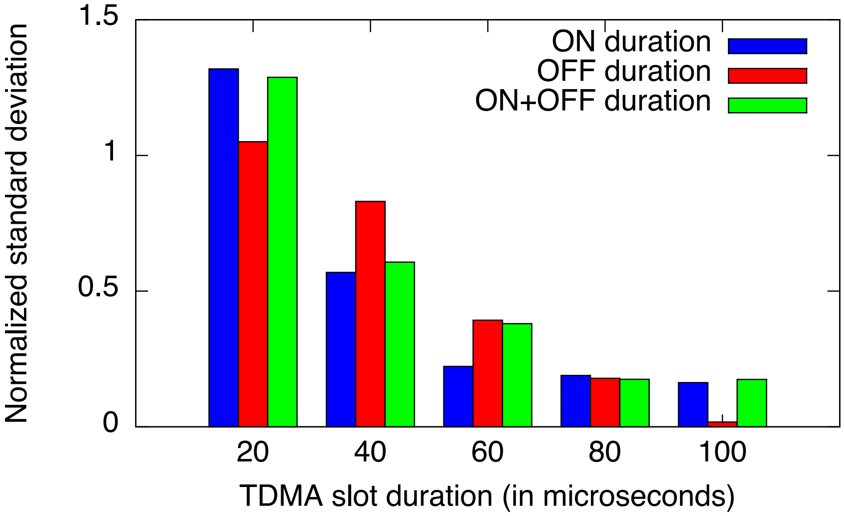


Figure 4.4. The normalized standard deviation in ON, OFF durations increases as the TDMA slot duration is reduced.

4.6.3 DPDK Data Plane + DPDK Control Plane

The obvious alternative is to employ the high-speed packet processing framework for both the control and data plane functions. Unfortunately, existing proposals lack the requisite abstractions, hooks, and API to integrate with data-center applications and proposed network fabrics. To overcome these limitations we describe Software Defined Dataplane SDD in Chapter 5. In this chapter, we discuss a very basic framework to 'start' and 'stop' flows using 'AddDequueCondition' and 'RemoveDequeueCondition' API calls in order to support tightly coupled flow management in data centers.

4.7 Evaluation

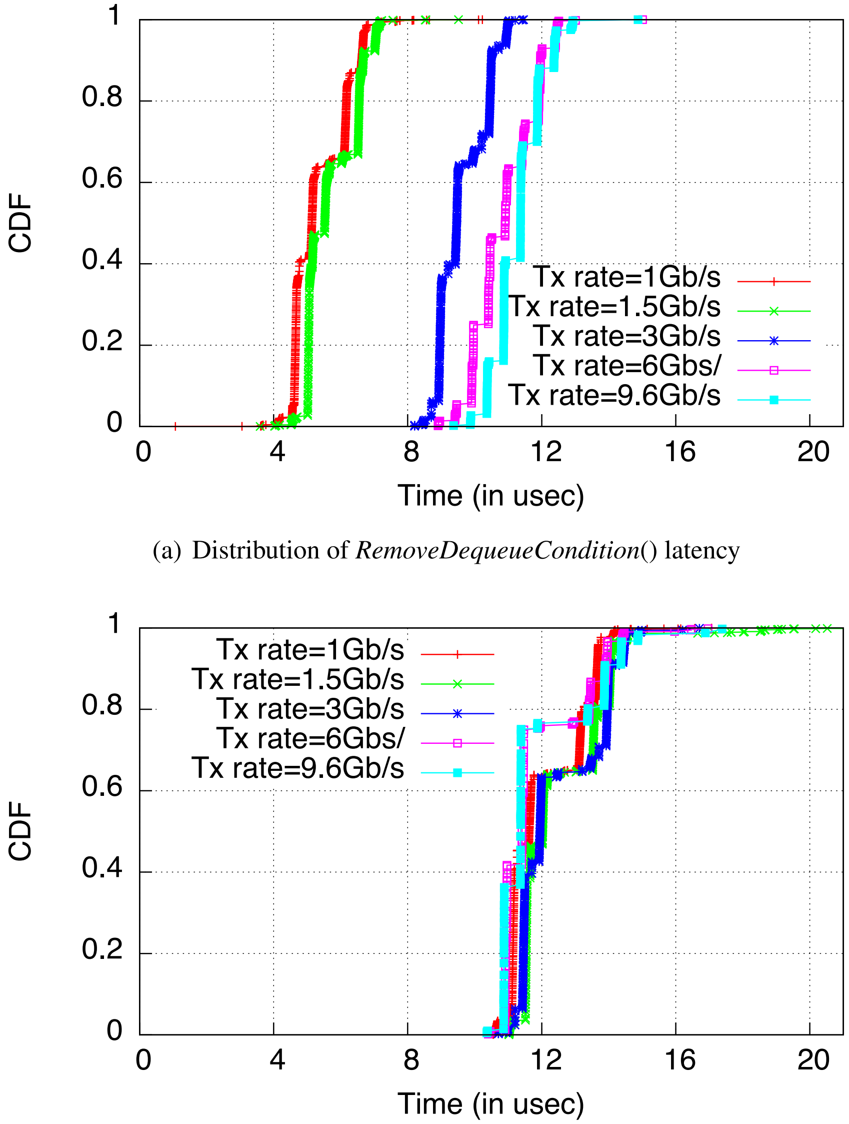
Tightly-coupled TDMA: The key evaluation metric we use to evaluate this timescale is the responsiveness of individual commands from the controller. We measure the response time of adding a new dequeue condition, and the response time of removing a dequeue condition on demand when the central controller sends updates to the end host.

These response times gate the performance of TDMA.

4.7.1 Microbenchmark: Overhead for Processing Control Packets

We start by implementing a tightly-coupled TDMA link sharing discipline. Here the central SDD controller (implemented using the NetFPGA board) begins and ends individual TDMA slots by invoking *AddDequeueCondition*() and *RemoveDequeueCondition*() API calls, to begin and end TDMA slots, respectively. In this experiment, for TDMA slot i, each host sends traffic at stride $i \mod 7$ and the TDMA slots are 200 μs long.

Figure 4.5(a) shows the latency between invoking the *RemoveDequeueCondition*() API call and having the transmission actually stop. We show latency results for five different transmission rates. Across rates, the distribution of latency is similar (about 4 μs), however the mean changes dramatically with the rate. This indicates the presence of in-NIC packet buffering outside the control of DPDK. Figure 4.5(b) shows the distribution of latency between the invocation of the *AddDequeueCondition*() API call and the first packet leaving the NIC (as witnessed by the FPGA-based switch). This latency does not vary as much as the *RemoveDequeueCondition*() call on the transmission (Tx) rate, and has a constant baseline and a constant variation for different transmission rates. This latency includes the overhead of receiving the SDD packet and updating state in the appropriate *SDDQueue*. Overall we find these results encouraging, since even with existing hardware and programmable data plane stacks, centralized, tightly-coupled TDMA is possible at timescales appropriate for proposals such as REACToR [58] and Mordia [28].



(a) Distribution of *RemoveDequeueCondition*() latency

(b) Distribution of *AddDequeueCondition*() latency

Figure 4.5. Characterizing the responsiveness of adding and removing conditional deqeueue commands.

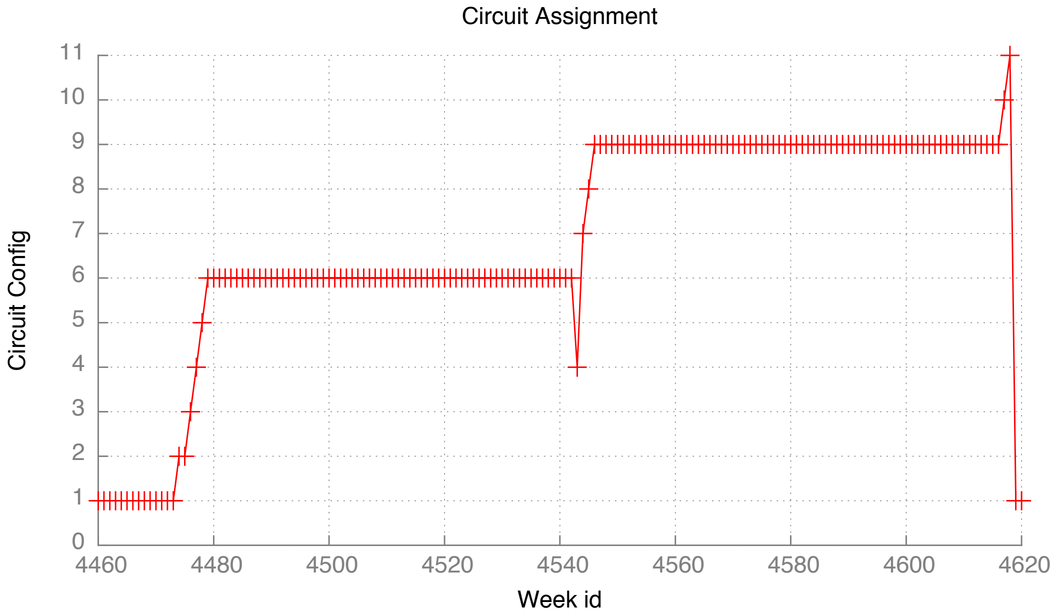


Figure 4.6. Changes in circuit configurations as demand is varied.

4.7.2 Microbenchmark: Circuit Reconfigurations

In addition to supporting functionality for starting and stopping flows, we also proive a basic framework for 'Demand Estimation' at the end hosts. The abstraction layers reports the buffer occupancy for DPDK abstraction and reports the buffer occupancy to the remote controller. Based on the esitmated demand, the remote scheduler computes the new circuit configuration and programs the optical switch to enforce the desired circuit connections.

In this microbenchmark, we measure if the reported demand results in the optimal circuit assignment or not and how stable is the circuit assignment with periodic demand notifications to the remote controller. In this experiment we report time as 'weeks' where each week is 1.5ms long. We have seven hosts R1, R2, R3, R4, R5, R6 and R7 connected using a hybrid REACToR switch. Initially, we have R3 sending traffic to R4, R5 and R6. After week 4500, the traffic pattern changes and R3 starts sending traffic to R0, R1, R2 and R7. R3 stops sending traffic after week 4600.

Figure 4.6 shows the different circuit configurations seen during the run of the experiment. Circuit config = 6, indicates the optimal configuration where R3 has circuit setup to R4, R5 and R6 and circuit config = 9 corresponds to the configuration where R3 has circuits to R0, R1, R2 and R7. As we can see the demand estimation framework does result in optimal circuit assignment in this experiment. Also, the system is able to converge to the optimal configuration within 3 weeks.

4.7.3 Microbenchmark: Circuit Link Utilization

In the previous experiment, we also measure the circuit link utilization for the different circuit configurations in order to ensure the circuits are being utilized to the full extent once the circuits are assigned. This ensures that in addition to reporting the correct demand, the end host (R3) is fully utilizing the circuit link once it assigned to it. This is governed by correctly invoking the ’AddDequeueCondition’ call to send traffic to the desired end host.

Figure 4.7 shows the utilization of the different links once the circuits are established by the remote scheduler. With REACToR the ideal circuit link utilization can only be 9 Gb/s as circuit link is rate limited to the 90% of the link capacity so that it does not overdrive the receiver. As we can see the measures circuilt link utilization is very close to the ideal circuit link utilization for all the different circuit link assignments as the traffic pattern changes.

4.7.4 Macrobenchmark: End to End Performance

We have also implemented a userspace network stack using Intel DPDK [26]. At present this stack supports only UDP, however it permits transmitting packets at microsecond timescales, enabling us to synchronize flows with Solstice [57] without being limited to the number of hardware queues in the NIC. Using this stack, we repeat

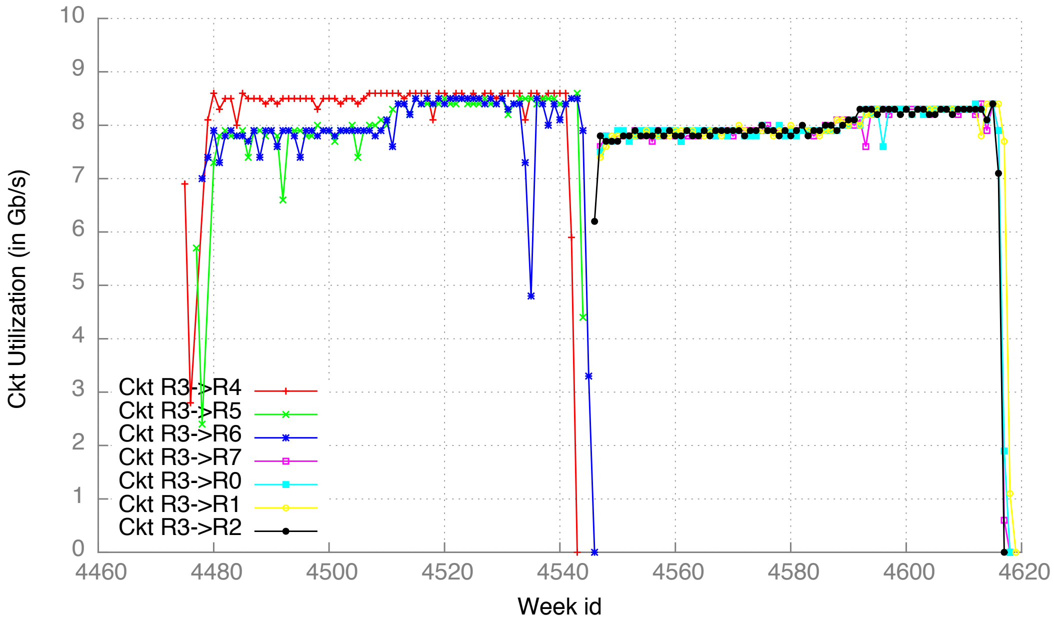


Figure 4.7. Circuit link utilization as demand is varied.

one of the experiments from Liu et al. (Figure 8 in [58]), but instead of precomputing circuit schedules, we use an implementation of Solsticecompute circuit schedules in real-time. We divide the eight hosts into two groups of four. Initially all hosts in a group stream data to the other hosts in the same group, and then the workload changes so that each host streams data to the hosts in the other group. The accumulation period is 1.5 ms, and at least that often each endhost sends an estimate of its demand to a centralized controller which invokes Solstice to obtain the circuit configuration schedule for the next period. For the first workload, we expect a schedule of three configurations where each configuration lasts 500 μs. For the second workload, we expect a schedule of four configurations where each configuration lasts 375 μs.

Figure 4.8 shows outgoing packets from host 3 at the workload transition time. This host periodically sends an estimate of the expected demand for the next scheduling period to the controller running Solstice. At the transition time (t=2100 μs), senders are halfway through their scheduling period. Packets already queued at the first three hosts continue to be sent via circuits, and the new flows are sent over the packet switch. At the end of its committed scheduling period, Solstice schedules a new set of configurations to react to the workload transition based on the demand inputs from the endhosts. Boxes in the figure indicate circuit assignments, and at the transition there are some partially filled boxes from the old workload, and a set of dashed lines for the new workload being sent on the packet switch which operates at much lower bandwidth.

The demonstration from this hardware deployment is that Solstice is fast enough to be embedded in a closed-loop controller that can react to changes in application demand within a single accumulation period (i.e., 1.5 ms).

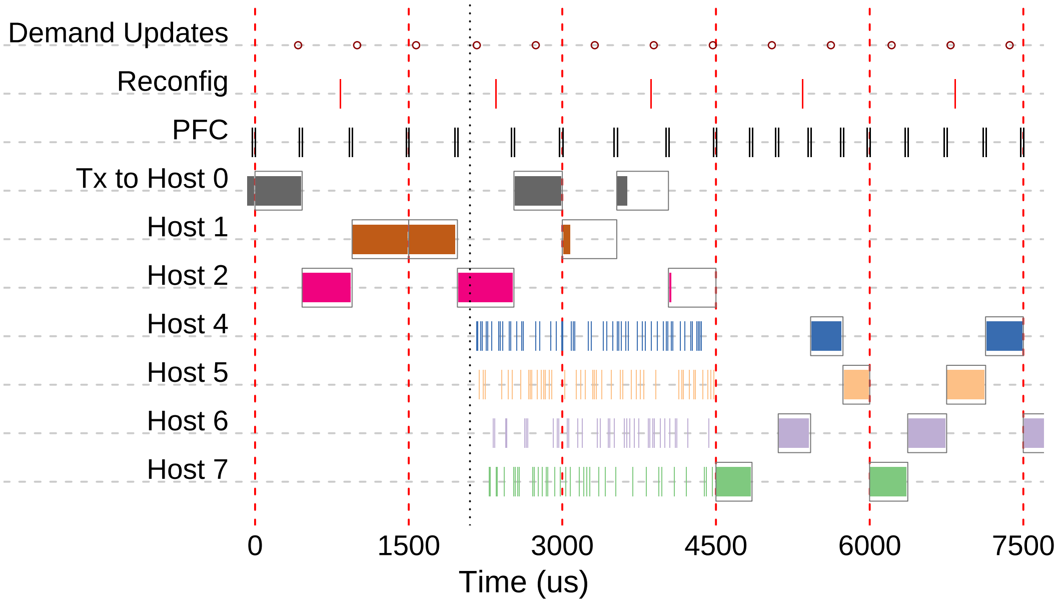


Figure 4.8. Real-time, closed-loop control plane invoking Solstice.

4.8 Conclusion

Hybrid electrical/optical networks show significant promise for supporting the ever increasing demand placed on data center networks. Unfortunately, several unsolved challenges could prevent their adoption. This chapter enumerated a set of challenges encountered during a year of experience with hybrid networks deployed on real hardware. Although the complete picture of how to build these networks is still unknown, we propose a flexible and framework based on OpenFlow for that we believe will enable a variety of future solutions to the remaining challenges. We are optimistic that solving these problems will lead to increased adoption of optical circuit switching into data center networks, leading to lower data center costs, complexity, and energy use.

In this chapter, we also described a basic framework for supporting link arbitration or TDMA scheduling of traffic for supporting optical switching within the data center. We leverage DPDK for enabling high speed packet processing and provide an abstraction on top to provide ’demand estimation’ functionality and ability to ’start’ and ’stop’ flows when control packets are received from a remote controller.

By adding support for tighly coupling the end host stack with a remote controller through microsecond timescale TDMA scheduling, we are moving towards a data center framework where both the end host and the network are managed by a centralized controller. This is in contrast to the existing architectures where the end host stack has remained largely decentralized and isolated from network management. By bringing the end host under central control we have achieved two goals (1) demand estimation, where the network is made aware of the state at the end host which enables the centralized controller to make better decisions to manage network traffic and (2) tight control over packet transmissions as a result of which the remote controller can precisely manage how the packets enter the network in order to maximize the network utilization. However, this is only a first step towards a completely software managed data center framework.

In order to fully leverage the potential of this software defined framework a future possibility is to make the interaction between data center applications and centralized controller more transparent and efficient. With that framework, applications will have the visibility in the network state and manage the transfers and communications to best suit the application requirements. We leave the interface between applications and the network fabric as future work.

Chapter 4, in part, has been submitted for publication of the material as Practical, Centralized Control of Programmable End-Host Data Planes. Tewari, Malveeka; Snoeren, Alex C.; Porter, George. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Software Defined Dataplane

5.1 Introduction

The performance of cloud computing is increasingly dependent on the performance of the underlying network. Numerous recent proposals aim to improve network performance by imposing greater control over data sources: D^3 [2, 83, 84, 87] improves upon TCP by strictly controlling the rate at which end hosts inject traffic into the network. QCN [72] enables congestion control by adjusting end-host sending rates through a low-latency, in-band control plane. pFabric [8] segregates flows within an end host to ensure conflict-free forwarding within the network fabric. FastPass [68] and REACToR [58] enforce TDMA link sharing among sources to support buffer-less electrical or optical interconnects. In each of these cases, it is the source host that is responsible for transmitting packets at the appropriate times and rates to implement the desired data plane functionality.

This level of source-based control is a departure from the traditional network model, where hosts are insulated from the details of the network by strong abstractions. Historically, the operating system (OS) at the source server chooses which packets to send based on an entirely decentralized scheduling discipline. It is then up to the network to buffer packets on their way to their destination. Yet, as mentioned above, further improvements necessitate eroding that abstraction, forcing the end host to be a much more active participant in network-wide scheduling. But for that to occur, the end-host data plane must be meet two requirements. It must be (1) programmable, performant, and precise, and (2) able to work in coordination with network switches and centralized controllers.

Significant progress has been made towards the first requirement, yet the second has remained elusive. In this paper, we argue for a *software-defined data plane (SDD)* API that can support configurable, source-based flow and packet processing functionality. In particular, our proposed design is simple enough to be implemented entirely within the context of programmable network stacks such as DPDK, avoiding the need for OS intervention. As a result, a centralized network controller will be able to remotely program the end-host data plane. As an initial exploration of SDD, we implement microsecond-scale TDMA, and show that it can be programmed from centralized network controllers.

5.2 Motivation

We first begin by identifying several recent trends which enable and motivate the need for a software-defined data plane (SDD).

5.2.1 Network Centralization

Software-defined networking has enabled centralized management of the network control plane. This lets network operators build centralized management applications that determine “how” packets are handled in the network in terms of routing, access control, load balancing, and traffic engineering. However, SDN does not prescribe “when” packets are transmitted, how they are paced, or how end hosts interleave packets across multiple flows headed to different destinations. This packet- and flow-level transmission behavior is important, as it directly impacts bandwidth utilization, latency and burstiness of the resulting traffic, and by extension, the overall performance of data center applications.

5.2.2 Software Programmable Data Planes

In general, network interface cards have been subject to ossification due to long hardware development cycles, preventing new features from being deployed in hardware. At the other extreme, packet processing in the kernel has been subject to latency and latency variation too high to support the above-mentioned functionality [73, 53]. This has resulted in the development of high-throughput packet-processing network stacks based on commodity hardware such as netmap [75] and DPDK [26], processing packets entirely outside of the kernel (DPDK) or partially outside of kernel control (netmap). Alternative approaches include “green field” OS designs [69, 12]. SoftNIC [42] aims to provide an extensible, entirely software-based NIC built using frameworks such as DPDK.

5.2.3 Server and Network Synchronization

The connection between in-network buffering and end-host synchronization has been long studied in the networking research literature. Buffering in packet switches is what enables end hosts to remain loosely synchronized; it is possible to rely on statistical multiplexing to “smooth out” packet arrivals from multiple hosts across time by storing and forwarding packets through in-switch buffers. Achieving tight synchronization among hosts on the Internet is impossible in general. In cloud and datacenter environments, however, ensembles of servers frequently act as a tightly-coordinated cluster infrastructure, especially in contexts that impose millisecond- or even microsecond-level SLAs on application response time. By coordinating packet transmission times across data center hosts, it is possible to greatly reduce—or even eliminate—in-network buffering. This is good for two reasons: first, it lowers end-to-end queueing and latency, and second, it enables cheaper and more energy-efficient network designs. A number of proposals advocate such an approach, including FastPass [68], DCTCP [6], and TDMA-based Ethernet [80]. While these proposals assume buffered packet switches, they aim to reduce the use of those buffers by carefully admitting traffic into the network. A second line of work aims to entirely eliminate the buffering within the network fabric to support end-to-end reconfigurable circuit-switching, either based on optical [29, 28, 82] or wireless [51] circuit switches. Here the need for eliminating in-network buffering comes from the physical requirements of the circuit technology, and so end-hosts and switches must implement strict TDMA, requiring high levels of end-host synchronization (at millisecond or even microsecond timescales).

5.2.4 Server and Network Virtualization

Many data centers and cloud environments are heavily virtualized, with end hosts hosting numerous VMs. Operators need to virtualize the network fabric to provide tenant isolation as well as weighted or proportional bandwidth sharing. While NICs typically support a small number of hardware rate limiters (e.g., 8–64), a key requirement for large network is scalable rate limiting across hundreds or thousands of flows and destinations. Numerous projects propose to implement such scalable isolation and bandwidth sharing, including EyeQ [48] and Senic [73]. However, these proposals are not under centralized network control.

5.2.5 Disaggregation and Rack-scale Computing

One challenge to implementing resource-efficient computing in cloud environments is choosing the right mixture of CPU, memory, storage, and network bandwidth. Conventional servers fix the ratio of these resources in a static binding, making provisioning and scheduling of jobs a considerable challenge. Recently proposed disaggregated

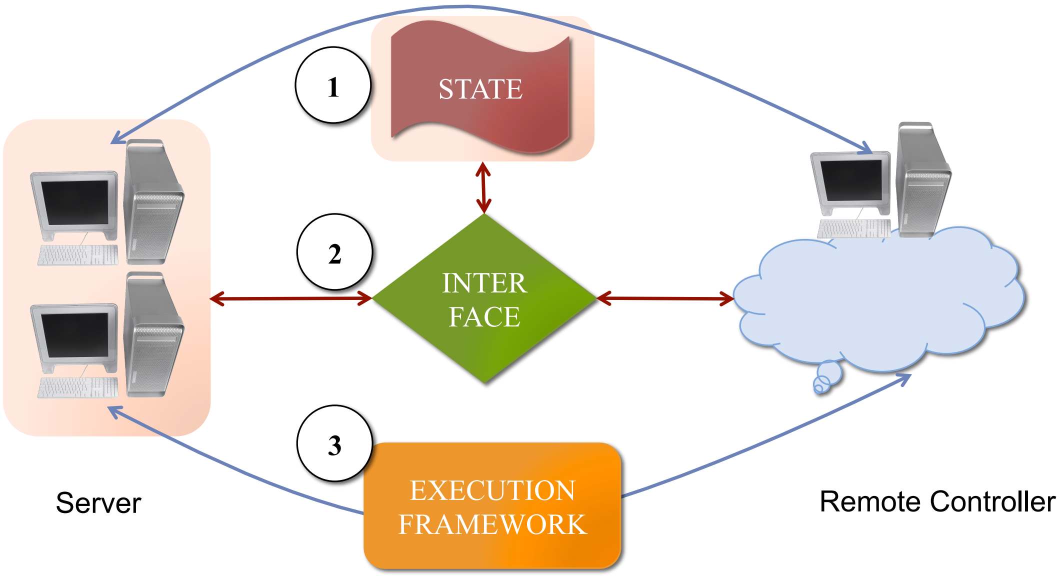


Figure 5.1. Key components for defining SDD.

data center designs [41, 31] argue for separating out underlying resources such as CPU and flash storage and putting them more directly on the network fabric. In this way, users can provision "on-demand" servers from these disaggregated resources that more closely fit their resource needs. However, achieving this vision requires tight synchronization of flows from individual components across the network fabric, placing stringent demands on the data plane itself.

A more conservative proposal in this direction is to eliminate the software overheads that exist between resources like memory and the network. An example is the use of RDMA to support very low-latency key-value stores [61, 49].

5.3 Design

We now describe the SDD framework and the SDD API.

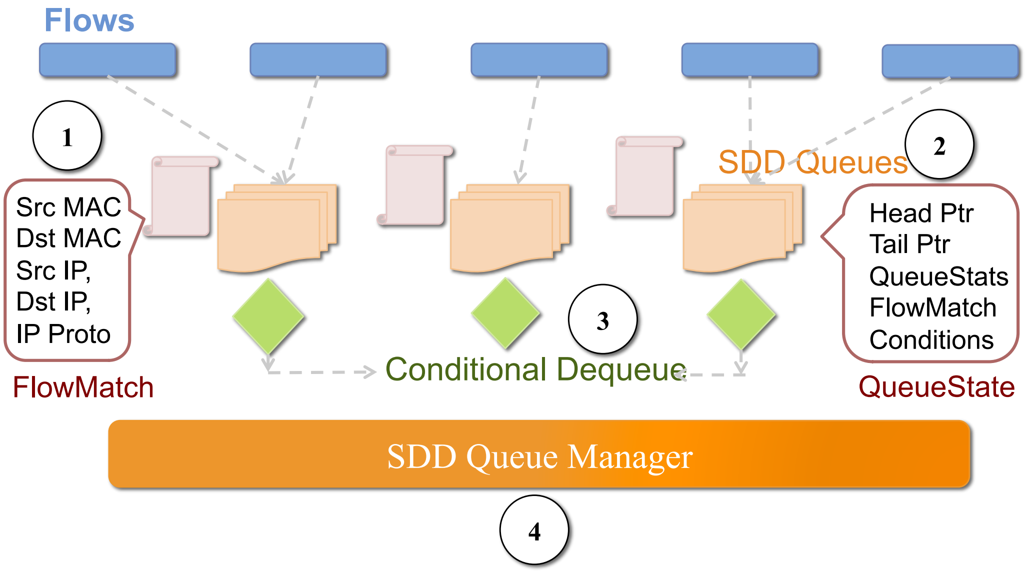


Figure 5.2. State and data structures for SDD.

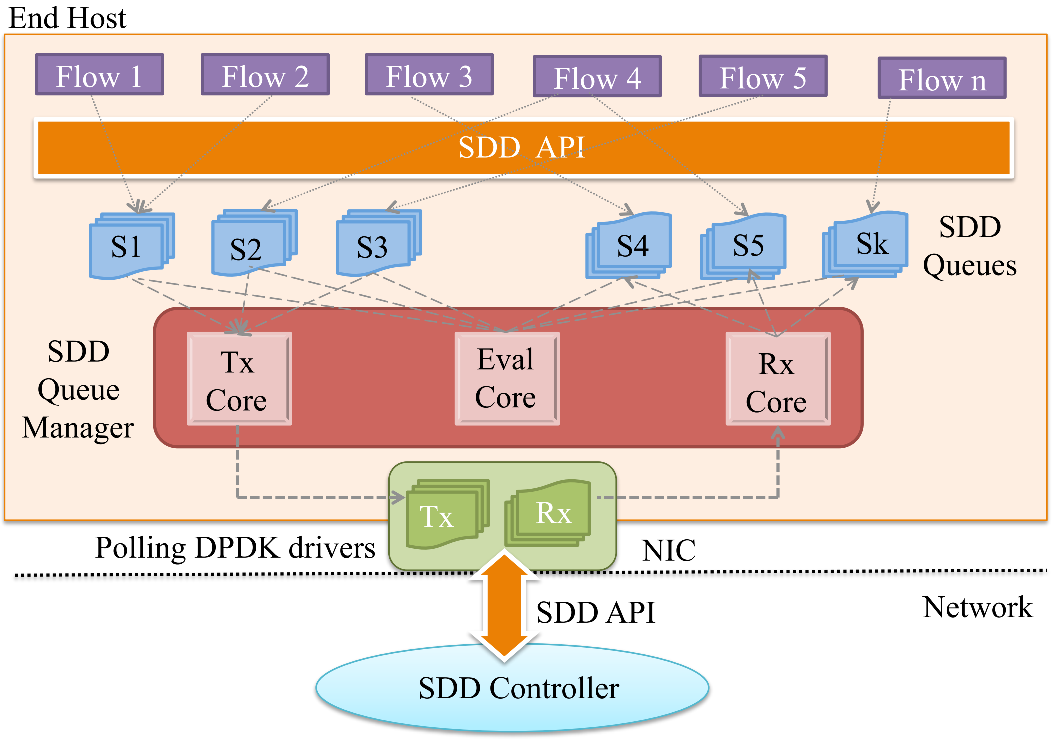


Figure 5.3. System overview of the SDD architecture.

5.3.1 Overview

An SDD-enabled network (shown in Figure 5.3) consists of several major components.

- SDD Controller: A logically centralized, though potentially physically distributed controller responsible for managing the data planes of a number of end hosts in the data center. This controller would likely be co-located with SDN or other network controllers (e.g., FastPass controllers).

- Network and NIC: Between the central controller and the SDD-enabled data plane is the network and NIC. We place no special requirements on these two components.

- Software programmable data plane: Each SDD-enabled end host requires a high-performance, programmable network stack such as DPDK [26], netmap [75], or SoftNIC [42] We use DPDK for our implementation.

To ensure that end hosts can adhere to strict time requirements requested by the central controller, it is necessary to synchronize the end-host clocks. We leverage the IEEE 1588 Precision Time Protocol (PTP) [71] to achieve sub-microsecond clock synchronization. Most modern NICs (including the Intel 82599 NICs used in this paper) provide PTP support in hardware.

5.3.2 SDD State and Data Structures

Our proposed SDD framework introduces a “logical queue” abstraction along with an associated API that allows an external controller to inspect queue-specific state and control transmissions from the queue. One or more flows can be associated with a single *SDDQueue*. A simplified version of the state kept in an *SDDQueue* is:

```
struct SDDQueue {
  int priority;
  float drainRate;
  QueueStats stats;

  List <Ptr> headPtrs;
  List<Expression> flowMatchExpressions;
  List<Condition> dequeueConditions;
  List<Event> triggerEvents;

  Packet* (*getNext)(SDDQueue*);
};
```

The *priority* and *drainRate* fields control flow prioritization and per-SDDQueue rate limiting. *stats* are read-only packet and byte counters. Next are any flow match expressions, dequeue conditions, and trigger events registered with this *SDDQueue* along with pointers for accessing the packets for different flows. Finally, *getNext*() is a virtual function used by, e.g., DPDK to extract the next packet for transmission. The default implementation of the next packet is to pull the first available packet from the flows mapped to the *SDDQueues*. This default implementation could be replaced by a custom implementation to support priority dequeueing or hierarchical bandwidth sharing among flows mapped to the same *SDDQueues* as shown in section 5.4.4

5.3.3 SDD Interface

The SDD API allows the network controller to control flow-to-*SDDQueue* bindings, choose which packet is next to send out of the NIC, and evaluate and signal triggers. Due to space constraints, we omit descriptions of the trigger APIs.

Flow match expression: In order to map flows to an *SDDQueue*, we provide the *MapFlow*() API call that takes a *flowMatchExpression* as an argument and associates that with an *SDDQueue*. We support matching flows on the IP 5-tuple (source IP address, destination IP address, source port, destination port and IP protocol). The API supports wildcarding certain fields in order to map a group of flows to a single shared memory queue and dynamic remapping of flows to queues. Wildcard flow-matching expressions allow the SDD framework to support co-flows [18], where the overall performance of a collection of flows is dependent upon the collective performance and interaction of individual flows, and also hierarchical sharing of bandwidth among a collection of flows.

1. *MoveFlows(SDDQueue* old, SDDQueue* new, FlowMatchExpression f)*: moves a flow mapping f from one queue to another, additionally moving any enqueued packets to the new *SDDQueue*.

2. *GetQueuesForFlows(FlowMatchExpression f)*: returns a set of *SDDQueues* that the given FlowMatchExpression f maps to (some fields in f can be wildcarded, which can result in returning more than one *SDDQueue*). For brevity we omit the description of API calls that, given a handle to an *SDDQueue*, return things like the number of packets in the queue, the number of bytes enqueued, etc.

Conditional dequeue: A key functionality of SDD is controlling individual packet transmissions at fine time scales with low overhead, which we refer to as "conditional dequeue." SDD supports the ability to express queue draining operations as conditional clauses which the SDD framework will evaluate to determine which packet should be transmitted at a given time. This functionality allows the SDD manager to execute "microprograms" sent by a centralized controller within the context of the packet-processing framework (e.g., DPDK). Conditional dequeue is implemented with the following API calls:

1. *AddDequeueCondition(SDDQueue* q, Expr cond)*: associates a dequeue condition with an *SDDQueue*. *Expr* is an expression that consists of logical comparisons of sub-expressions, each of which can refer to the current time, the state of a particular *SDDQueue* (such as queue length), and even the contents of packets in an *SDDQueue*. Dequeue conditions can be added to an *SDDQueue* by the end host itself in addition to those supplied by the central controller.

2. *RemoveDequeueCondition(SDDQueue* q, Expr cond)*: removes a dequeue condition from an *SDDQueue*.

5.3.4 SDD Grammar

In this section, we formally describe the grammar for the dequeue condition statements. The primitive value of a dequeue expression is either “True” or “False”. The true value means that the queue is eligible for sending packets and the ’false’ value indicates that the queue is not eligible to send packets. Based on evaluation of the different conditions, each dequeue expression will either evaluate to true or false.

```
expr := TRUE | FALSE
expr := var OP constant
expr := expr (AND|OR) expr


var := linear combination of:
       [queueStats, currTime, lastXmitTime, lastRecvTime]
OP := <, >, <=, >=, =, !=
constant := numerical value
```

5.3.5 SDD Execution Framework

Each end host runs a ***SDDQueue manager*** process that is responsible for interfacing the SDD data plane with the controller by executing the SDD API calls. These calls control flow-to-*SDDQueue* bindings, choose the next packet to send out of the NIC, and evaluate and signal triggers. The manager is the context in which conditional dequeue and trigger expressions are evaluated and it can leverage the multicore architecture for separating packet transmission and dequeue condition evaluation on different cores as shown in Figure 5.3.

The time required to evaluate any dequeue conditions directly impacts the performance of the SDD data plane framework. It is a function of the size of the "microprograms" sent by the controller, the number of *SDDQueues* at the end host, and the *type* of the conditions. If the network controller chooses not to impose conditions and instead directly manages the different queues by explicitly indicating from which queue packets should be drained at each moment in time, the evaluation task is trivial. In this case the main task for data plane manager is to simply dequeue the packets from the multiple queues efficiently.

In the common case, however, the controller specifies conditions that need to be evaluated explicitly at the end host. Depending on the type of condition, it might be possible to evaluate the condition beforehand and have the conditional result cached so that the data plane manager can simply inspect the result when dequeueing packets. For example, if the conditions are specified based on time slots (see 5.4.2), the data plane manager can precompute the schedule so that it does not have to evaluate the dequeue condition individually for every queue and can perform a quick lookup to determine which queue to drain at the time of packet transmission. In these cases, the evaluation and transmission functions can be decoupled and executed on different cores to provide improved scalability for a large number of queues.

For certain dequeue conditions, however, the evaluation and the transmission cannot be easily decoupled. E.g., if the controller program seeks to dequeue packets from the instantaneously smallest queue, the condition cannot be pre-evaluated. In such a case, we can have different cores manage different *SDDQueues* to improve performance.

5.4 SDD "Recipes"

An advantage of the extensible SDD interface is that it allows us to express existing proposals using the proposed API. We now describe three networking functionalities implemented with the SDD API.

5.4.1 Tightly-coupled TDMA

REACToR: REACToR is an examples of tightly controled SDD framework. Here the controller sends explicit control packets to start and stop the transmissions of packets from the queues.

```
// j = source Host ID; k = destination host ID
// For Host[j]: slot[j][k] = TDMA slot duration
controlPacket p_start, p_end;
  // Do for each destination k
p_start.add(SDD[j].DequeueCondition(Q[k], TRUE))
// wait for duration slot[j][k]
p_end.add(SDD[j].DequeueCondition(Q[k]), FALSE)
```

5.4.2 Loosely-coupled TDMA

FastPass: In order to implement FastPass using the SDD framework, we begin by creating per-destination *SDDQueues* since the FastPass arbiter assigns time slots based on source-destination pairs. The controller sends TDMA slot start and end times to the host as part of its scheduling algorithm. The SDD manager then compares the current time with the TDMA slot boundaries to determine which *SDDQueue(s)* are active. In this particular case, only one queue will be active. We evaluate this recipe in Section 5.5:

```
// j = source Host ID; k = destination host ID
// For Host[j]: start[j][k] = TDMA slot start time
//              end[j][k] = TDMA slot end time
controlPacket p;
  // Do for each destination k
  cond1 = currTime() > start[j][k]
  cond2 = currTime() < end[j][k]
  p.add(SDD[j].DequeueCondition(Q[k], cond1 && cond2))
```

5.4.3 Rate Limiting

Rate-based congestion control: Rate-based congestion control relies on tight coordination between end hosts and network switches to dictate sending rates for sources. To implement the host-portion of RCP, we must enforce rate limits to maintain the desired inter-packet gap between transmitted packets. When SDD receives an updated rate limit for a particular flow, it removes the old dequeue condition and adds a new dequeue condition which signifies an *SDDQueue* as eligible to transmit packets once enough time has elapsed to enforce the rate limit:

```
// j = source Host ID; k = destination host ID
// newIPG[j][k] = desired rate between j, k
controlPacket p
  // Do for each destination k
  newcond = currTime() - Q[k].stats.lastXmit > newIPG[j][k]
  p.add(SDD[j].DequeueCondition(Q[k], newcond))
```

5.4.4 Hierarchical Bandwidth Sharing

Rate limiting multiple flows within a single SDD queue: To support hierarchical bandwidth sharing, more than one flow is assigned to an *SDDQueue*, and a modification is made to how packets are dequeued from the *SDDQueue*. Specifically, we choose a custom implementation of the *getNext*() function. To share the bandwidth assigned to an *SDDQueue* uniformly across the multiple flows mapped to it, *getNext*() is defined to pull packets in a round robin manner:

```
// k = destination Host ID
function getNextHierarchical(SDDQueue *q):
  static counter
  counter = counter + 1
  if counter == q->stats.numFlows:
    counter = 0 // avoid overflow
  flow_id = counter % q->numFlows
  Packet* p = (Packet*)q->headPtr[flow_id])
  q->headPtr[flow_id] = q->headPtr[flow_id] + p->size
  return p
Q[k].getNext = getNextHierarchical
```

5.5 Evaluation

In this section, we show that a software-defined data plane based on the SDD API is capable of implementing novel data-plane functionality, and that it is responsive enough to be programmable under centralized control. We focus on implementing the TDMA link scheduling necessary to support recent proposals such as FastPass [68] and REACToR [58]. TDMA requires very tight synchronization between the end host and the network controller, in the limit modulating the release of individual packets (in the case of FastPass)—a significant challenge for today's end hosts.

A centralized controller programming any SDD-enabled end host has a choice in what timescale those programs can remain active, as mentioned in Section 5.3.5. Thus we evaluate TDMA link scheduling under two different timing regimes: (1) a very tightly-coupled timescale in which the controller "microprograms" each scheduling and descheduling action, and (2) a more loosely-coupled timescale in which the controller programs a TDMA regime that remains in effect until the controller programs a new TDMA schedule. Each of these timescales has a natural evaluation metric:

Loosely-coupled TDMA: Systems like FastPass implement TDMA on individual packets, yet batch and amortize link scheduling algorithms across a larger batch of packets. Thus for such loosely-coupled regimes, the programming time is less critical, and the key evaluation metric we measure is the overall throughput of flows, which is a function of the *efficiency* of getting packets onto the wire after a TDMA slot begins, the *precision* of stopping packets when the slot is about to end, and the *accuracy* of ensuring that packets are not sent during the wrong slot. Of particular importance in this second regime is the scalability of accurately evaluating dequeue conditions across a potentially large number of *SDDQueues*. We evaluate the ability to implement loosely-coupled

TDMA as a function of the number of *SDDQueues* and time constraints on dequeue conditions.

Setup: We deployed SDD on a testbed consisting of eight HP DL360p servers, each with a pair of Intel E5-2630 six-core CPUs (2.3 GHz) and an Intel 82599-based 10-Gb/s NIC, running Debian Linux with kernel version 3.4.44. We implement the centralized SDD controller using a 1-Gb/s NetFPGA FPGA board since it allows us to control the jitter of control message transmissions.

Each of our hosts is directly connected to a 24-port, 10-Gb/s Xilinx Virtex 6 FPGA, which implements a simple packet switch. Furthermore, this FPGA measures the traffic transiting it and generates a measurement record for each packet it sees, which includes the packet's source and destination address, its size, and a timestamp of when the packet left or arrived to the port. The timing precision of the timestamp is 6.4 ns. The FPGA forwards these measurement packets to a measurement host, which we use to generate the results in this section. For all our experiments, we use 1500-byte packets unless otherwise indicated.

5.5.1 Tightly-coupled TDMA: REACToR

Table 5.1. Overhead of processing REACToR control packets

Tx Rate	External control overhead			
	Fixed		Variable	
(Gb/s)	OFF (μs)	ON (μs)	ON (μs)	OFF (μs)
1	5.3	12.0	0.81	1.8
1.5	5.7	12.6	0.75	6.0
3	9.6	12.5	0.70	3.6
6	10.9	11.8	0.89	1.2
9.6	11.3	11.8	0.69	1.3

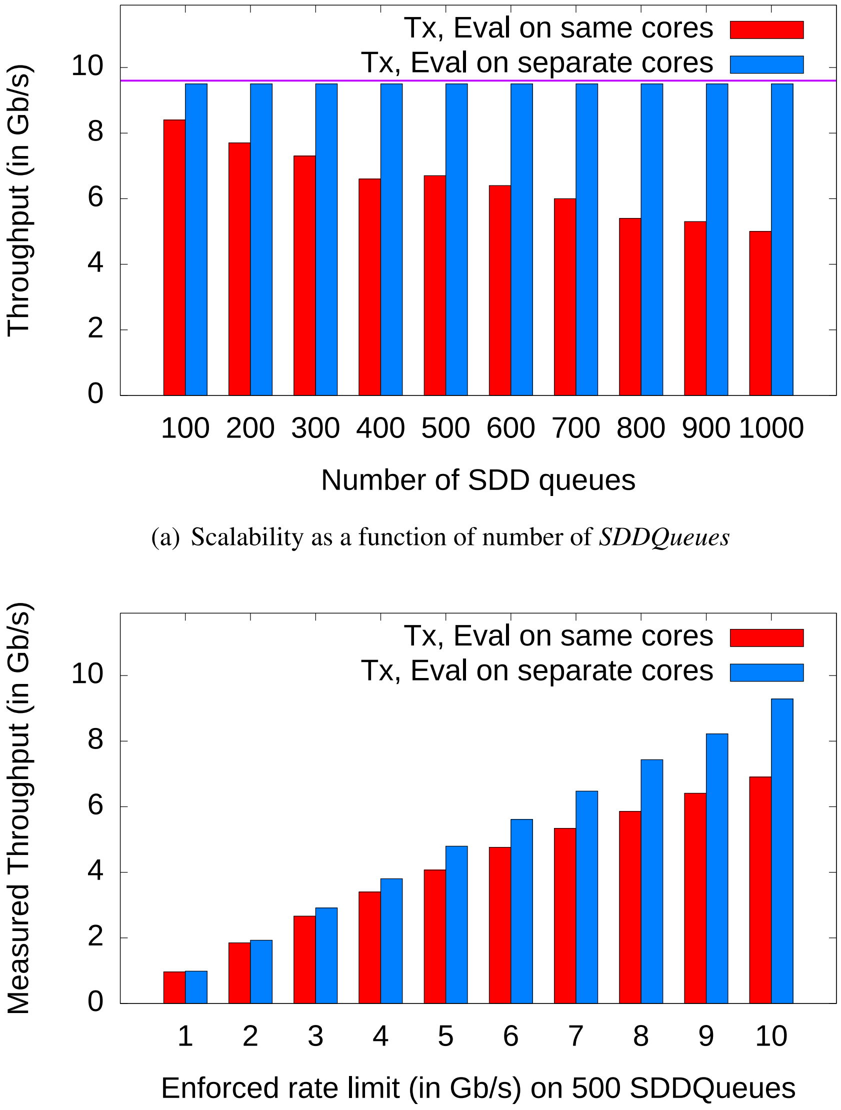
We implemented the recipe for supporting REACToR and measure the overhead in responding to the control packets received by the remote controller. Table 5.1 shows the baseline and the variable components of responding to the control packets received for achieving REACToR style TDMA scheduling.

5.5.2 Loosely-coupled TDMA: FastPass

Proposals such as FastPass do not send explicit start and stop events to end hosts, but rather send a TDMA schedule amortized over a batch of packets. This loosely-coupled TDMA model is more complex to implement than tightly-coupled TDMA, because the *SDDQueue* manager execution context can become a bottleneck. To quantify this, we evaluate the scalability of our SDD framework as a function of (1) the number of *SDDQueues*, and (2) the constraints on evaluating the dequeue conditions. As shown in Figure 5.4(a), we see that sharing a core between packet handling and condition evaluation results in poor scalability. Instead, by separating those two functions, precise TDMA support is attainable at scales of up to 1000 *SDDQueues*. We ran further experiments to evaluate when does the scalability starts suffering even with a separate core and our results indicate that beyond 2600 *SDDQueues*, we would require an additional core for evaluating dequeue conditions in order to maintain line rate.

5.5.3 Rate Limiting: SENIC

Evaluating TDMA schedules can be done independently without needing most accurate *SDDQueue* state. However, rate limiting requires maintaining the inter-packet gap and the condition evaluation is constrained by how soon in advance is the accurate *SDDQueue* state available. In Figure 5.4(b), we see that as the desired rate limit is increased, the constraints on inter-packet gap become tighter for a set of 500 *SDDQueues*. While it is possible to enforce smaller rate limits with a single core, enforcing higher rate limits requires separating the packet handling and condition evaluation functionality.



(a) Scalability as a function of number of *SDDQueues*

(b) Scalability as a function of constraints on Dequeue conditions

Figure 5.4. The SDD API scaling behavior as a function of number of *SDDQueues* and dequeue conditions.

Table 5.2. Accuracy of software rate limiters as we vary the number of flows and the per-flow rate limit.

Number of flows	Per-flow rate	Desired Inter-packet gap (IPG)	Root Mean Square Error in IPG
10	1 Mb/s	12 ms	3.1μs
100	1 Mb/s	12 ms	6.9μs
1000	1 Mb/s	12 ms	17.1μs
1	10 Mb/s	1.2 ms	1.7μs
10	10 Mb/s	1.2 ms	2.6μs
100	10 Mb/s	1.2 ms	4.1μs
1	100 Mb/s	120μs	1.6μs
10	100 Mb/s	120μs	2.6μs
1	1 Gb/s	12μs	1.4μs
2	1 Gb/s	12μs	3.1μs

5.6 Conclusion

Even as the end-host network stack becomes more programmable, it remains walled off from centralized control. In this work, we proposed a Software-defined Data plane (SDD) suitable for placing the stack under low-latency network control. Our evaluation shows that implementing SDD using today's software and hardware shows promise, and that as data centers become increasingly centralized, SDD serves as a mechanism for enabling the end-host network stack to participate in next-generation data center network designs.

We describe (1) state required at the end hosts in order to prove the SDD functionality, (2) interface and grammar for expressing how the packet transmissions are controller at the end host and (3) the execution framework for enforcing the packet transmission policies specified by a remote controller at the end host stack. In addition, we also presented examples of how to implement existing data center fabric management proposals using the SDD framework including REACToR, FastPass and SENIC. Our evaluation shows that SDD is in infact deployable in the data centers and can achieve the desired functionality with precision and low overhead.

Chapter 5, in part, have been submitted for publication of the material as Practical, Centralized Control of Programmable End-Host Data Planes. Tewari, Malveeka; Snoeren, Alex C.; Porter, George. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conclusion

6.1 Summary of Contributions

In this thesis we proposed network and end host stack based flow management primitives. We present WCMP for weighted flow hashing across the available next-hops to the destination and show how to deploy our techniques on existing commodity silicon and improve network maintenance/diagnosis. Our performance evaluation on an OpenFlow-controlled network testbed shows that WCMP can substantially reduce the performance difference among flows compared to ECMP, with the potential to improve application performance and network diagnosis; and complements dynamic solutions like MPTCP for better load balancing as it makes balanced capacity available through the fabric as a function of current topology.

Even as the end-host network stack becomes more programmable, it remains walled off from centralized control. In this work, we propose a Software-defined Data plane (SDD) suitable for placing the stack under low-latency network control. Although in its early stages, our evaluation shows that implementing SDD using today's software and hardware shows promise, and that as data centers become increasingly centralized, SDD serves as a mechanism for enabling the end-host network stack to participate in next-generation data center network designs.

6.2 Limitations and Future Work

In this section we discuss the limitations and potential future work for our proposed network based flow management solutions (WCMP) and the end host based flow management framework (SDD).

6.2.1 WCMP

A potential limitation of the WCMP framework is that it fails to account for asymmetric traffic patterns in the data center networks. The proposed algorithms for computing weights only account for topology asymmetry but not for traffic hot-spots that occur frequently in the data centers. To account for traffic imbalance within a data center, WCMP needs to periodically measure the traffic matrix in the network and the compute the weights based on the traffic matrix in order to optimize for fairness.

Another limitation of WCMP is that it only supports shortest path routing. In order to effectively leverage the multipath capacity of large scale data centers and route around congestion hot spots, WCMP should be able to support non-shortest path routing. This would involve avoiding routing loops while computing non-shortest paths, programming at switches at each hop to route a fraction of flows along the non-shortest paths using the right set of weights. The WCMP architecture can be easily extended to suport the above mentioned functionality once the centralized scheduler can support computing non-shortest path routes. We leave that functionality as future work.

6.2.2 SDD

For the SDD work, a potential future direction is to further explore the interfacing with data center applications. It is possible to attach event listeners to the different conditions so that the applications can be promptly notified of state changes to which they can respond to as desired.

While SDD addresses the missing interface between the end host network stack and the applications in order to exercise fine-grained control over packet transmissions, the interface between the applications and the network is still missing. An ideal framework would support more transparent interaction between applications and a centralized controller wherein applications would have visibility into network state and they can orchestrate the application functionality and transfers to fully leverage the potential of the software defined framework.

Bibliography

[1] Dennis Abts, Michael R. Marty, Philip M. Wells, Peter Klausler, and Hong Liu. Energy Proportional Datacenter Networks. *SIGARCH Comput. Archit. News*, 38, June 2010.

[2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proc. of ACM EuroSys*, April 2013.

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. In *Proc. of SC*, November 2009.

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

[5] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. of Usenix NSDI*, April 2010.

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

[7] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vehdat, and Masato Yasuda. HULL: A High Bandwidth, Ultra-Low Latency Data Center Fabric Architecture. In *Proc. of Usenix NSDI*, April 2012.

[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proc. of ACM SIGCOMM*, August 2013.

[9] David Applegate, Lee Breslau, and Edith Cohen. Coping with Network Failures: Routing Strategies for Optimal Demand Oblivious Restoration. In *Proc. of ACM SIGMETRICS*, June 2004.

[10] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, Volume 23, March 2003.

[11] Broadcom's OpenFlow Data Plane Abstraction. http://www.broadcom.com/products/Switching/Software-Defined-Networking-Solutions/OF-DPA-Software.

[12] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. of Usenix OSDI* , October 2014.

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

[14] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proc. of ACM CoNEXT*, December 2011.

[15] L. N. Bhuyan and D. P. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Trans. Comput.*, 33, April 1984.

[16] Bing Search Engine. www.bing.com.

[17] Y. Chen, R. Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proc. of ACM WREN*, 2009.

[18] Mosharaf Chowdhury and Ion Stoica. Coflow: A Networking Abstraction for Cluster Applications. In *Proc. of ACM HotNets*, October 2012.

[19] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *Proc. of ACM SIGCOMM*, August 2011.

[20] C. Clos. A Study of Non-blocking switching networks. *Bell Syst. Tech. J.*, 32:406–424, 1953.

[21] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[22] A. R. Curtis, Wonho Kim, and P. Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *Proc. of IEEE INFOCOM*, April 2011.

[23] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling Flow Management For High-Performance Networks. In *Proc. of ACM SIGCOMM*, Aug 2011.

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

[25] Impressive Packet Processing Performance Enables Greater Workload Consolidation. http://www.intel.com/content/www/us/en/communications/communications-packet-processing-brief.html.

[26] Data Plane Development Kit. https://www.dpdk.org.

[27] Anwar Elwalid, Cheng Jin, Steven H. Low, and Indra Widjaja. MATE: MPLS Adaptive Traffic Engineering. In *INFOCOM*, April 2001.

[28] Nathan Farrington, George Porter, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Hunting Mice with Microsecond Circuit Switches. In *Proc. of ACM HotNets*, October 2012.

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

[30] Facebook Social Network. www.facebook.com.

[31] Facebook Yosemite Modular Chassis. https://code.facebook.com/posts/1616052405274961/introducing-yosemite-the-first-open-source-modular-chassis-for-high-powered-microservers-/.

[32] Patrick Geoffray and Torsten Hoefler. Adaptive Routing Strategies for Modern High Performance Networks. In *Proc. of IEEE HOTI*, August 2008.

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

[34] Gnu Linear Programming Kit. http://www.gnu.org/software/glpk.

[35] Google Search Engine. www.google.com.

[36] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, August 2009.

[37] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proc. of ACM SIGCOMM*, August 2009.

[38] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *Proc. of the ACM SIGCOMM*, August 2008.

[39] Hadoop. http://www.hadoop.org.

[40] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting Data Center Networks with Multi-gigabit Wireless Links. In *Proc of ACM SIGCOMM*, August 2011.

[41] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network Support for Resource Disaggregation in Next-generation Datacenters. In *Proc. of ACM HotNets*, November 2013.

[42] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report EECS-2015-155, UC Berkeley, May 2015.

[43] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving Energy In Data Center Networks. In *Proc. of Usenix NSDI*, April 2010.

[44] Kaj Holmberg. Optimization Models for Routing in Switching Networks of Clos Type with Many Stages, 2007.

[45] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF, November 2000.

[46] HP SDN Ecosystem. http://h17007.www1.hp.com/us/en/networking/solutions/technology/sdn/.

[47] MPTCP Htsim Implementation. http://nrg.cs.ucl.ac.uk/mptcp/implementation.html.

[48] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Proc. of Usenix NSDI*, April 2013.

[49] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-value Services. In *Proc. of ACM SIGCOMM*, August 2014.

[50] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *Proc. of ACM SIGCOMM*, August 2005.

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

[52] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *Proc. of ACM IMC*, 2009.

[53] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proc. of ACM SoCC*, October 2012.

[54] John Kim and William J. Dally. Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks. In *ISCA*, June 2007.

[55] Murali Kodialam, T. V. Lakshman, and Sudipta Sengupta. Efficient and Robust Routing of Highly Variable Traffic. In *Proc. of ACM HotNets*, November 2004.

[56] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of Usenix OSDI*, October 2010.

[57] He Liu, Rishi Kapoor, Malveeka Tewari, Alex Forencich, Sen Zhang, Alex Snoeren, Geoff Voelker, George Porter, and George Papen. Scheduling Circuits in a Packet World. Under submission.

[58] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit Switching Under the Radar with REACToR. In *Proc. of Usenix NSDI*, April 2014.

[59] Vincent Liu, Daniel Halperin, Arvind Krishnamurhty, and Thomas Anderson. F10: A Fault-Tolerant Engineered Network. In *Proc. of Usenix NSDI*, April 2013.

[60] Memcached. http://www.memcached.org.

[61] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proc. of Usenix ATC*, June 2013.

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

[63] MPLS-TE. http://blog.ioshints.info/2007/02/unequal-cost-load-sharing.html.

[64] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, and Jeffrey C. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *Proc. of Usenix NSDI*, 2010.

[65] Eiji Oki, Zhigang Jing, Roberto Rojas-Cessa, and H. Jonathan Chao. Concurrent Round-Robin-Based Dispatching Schemes for Clos-Network Switches. *IEEE/ACM Transactions on Networking*, 10, 2002.

[66] Openflow. www.openflow.org.

[67] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable High-Performance Storage Entirely In DRAM. *SIGOPS System Review*, 2010.

[68] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized Zero-Queue Datacenter Network. In *Proc. of ACM SIGCOMM*, Aug 2014.

[69] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proc. of Usenix OSDI* , October 2014.

[70] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *Proc. of ACM SIGCOMM*, August 2012.

[71] IEEE 1588 Precision Time Protocol (PTP). www.ieee1588.com.

[72] Quantified Congestion Notification (IEEE 802.1Qau). http://www.ieee802.org/1/pages/802.1au.html.

[73] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. SENIC: Scalable NIC for End-Host Rate Limiting. In *Proc. of Usenix NSDI*, April 2014.

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

[75] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proc. of Usenix ATC*, June 2012.

[76] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proc. of Usenix HotCloud*, June 2011.

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

[78] Apache Spark. spark.apache.org.

[79] Twitter. www.twitter.com.

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

[81] Curtis Villamizar. OSPF Optimized Multipath (OSPF-OMP), draft-ietf-ospf-omp-02 , February 1999.

[82] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T. S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-Through: Part-time Optics in Data Centers. In *Proc. of ACM SIGCOMM*, August 2010.

[83] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proc. of ACM SIGCOMM*, August 2011.

[84] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM*, August 2011.

[85] Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, and Yongguang Zhang. MDCube: A High Performance Network Structure for Modular Data Center Interconnection. In *Proc. of ACM CoNEXT*, December 2009.

[86] Xipeng Xiao, Alan Hannan, and Brook Bailey. Traffic Engineering with MPLS in the Internet. *IEEE Network Magazine*, 2000.

[87] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. The Only Constant is Change: Incorporating Time-varying Network Reservations in Data Centers. In *Proceedings of the ACM SIGCOMM*, August 2012.

[88] YouTube. www.youtube.com.

[89] Yang Zhang and Zihui Ge. Finding Critical Traffic Matrices. In *Proc. of DSN*, June 2005.

[90] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proc. of ACM EuroSys*, 2014.

[91] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y. Zhao, and Haitao Zheng. Mirror Mirror on the Ceiling: Flexible Wireless Links for Data Centers. In *Proc. of ACM SIGCOMM*, August 2012.