UNIVERSITY OF CALIFORNIA, SAN DIEGO

Fundamental Challenges for Hybrid Electrical/Optical Datacenter Networks

A Thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Hamid Hajabdolali Bazzaz

Committee in charge:

Professor Amin Vahdat, Chair
Professor Stefan Savage
Professor Alex C. Snoeren

2011

The Thesis of Hamid Hajabdolali Bazzaz is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2011

EPIGRAPH

دوست من،

می‌دانم که چه می کشی خوب می‌دانم. اما تو که در دامنه آتشفشان منزل گرفته‌ای باید بدانی که چگونه می‌توان زیر فوران آتش زیست. ما را خداوند برای زیستنی چنین به زمین آورده است. چرا که مرغ عشق ققنوس است که در آتش می‌زید نه آنکه رنگین کمان می‌پوشد و در بوستان‌های عافیت شکر می‌خورد و شکر شکنی می‌کند. مگر سوخته دلی و سوخته جانی را جز از بازار آتش می‌توان خرید؟

برگزیده از نامه‌ای از مرتضی آوینی به ابراهیم حاتمی‌کیا

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This thesis has been literally written over the period my advisor Prof. Amin Vahdat has been on sabbatical. But that didn't slow it down from happening as I knew Amin would still be always accessible and help me through it. I would like to express my gratitude to him for everything over the period of my graduate studies at UC San Diego. I also would like to appreciate the rest of my thesis committee members, Prof. Stefan Savage and Prof. Alex C. Snoeren for their valuable feedback. In addition, I am honored to had a chance to work closely with a few other professors/researchers from Carnegie Mellon University, Rice University, and Intel Labs Pittsburgh over the course of our joint research effort in form of twice-weekly teleconference meetings. Specifically, Prof. Dave Anderson, Prof. T. S. Eugene Ngr., Dr. Michael Kaminsky, Dr. Michael A. Kozuch, and Dr. Konstantina (Dina) Papagiannaki. I also would like to thank Guohui Wang, my colleague PhD student at Rice University. I learned a lot from the thought and discussions each of them put on the table.

I also would like to thank each and everyone of my colleagues at UC San Diego working with whom has been such a great experience for me: Nathan Farrington, Dr. George Porter, Sivasankar Radhakrishnan, Vikram Subramanya, and Malveeka Tewari. I am sure if it is not impossible, it certainly is very though to find such a team in place. Specifically, everytime I sat with George to work on something, I ended up learning and admiring him even more than before. I can't express how much I enjoyed every minute of collaborating with him and how much I owe him. I also would like to express my greatest appreciation to Sivasankar who always sacrifices himself for the success of project and coordinating closely with him has been an amazing experience for my personal and academic life.

This thesis is partially based on the Helios paper [FPR+10] published in SIGCOMM'10 of which I was a coauthor. Specifically, among my contributions to that effort was the Topology Manager software which is described in this thesis. There are also material from the paper titled "Switching the Optical Divide: Fundamental Challenges for Hybrid Electrical/Optical Datacenter Networks" submitted to SOCC'11 which I was an author of.

VITA

2003-2007	B.Sc. in Computer Engineering, University of Tehran
2007-2009	M.Sc. Student in Computer Engineering, University of Tehran
2009-2011	M.Sc. in Computer Science, University of California, San Diego

SELECTED PUBLICATIONS

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." Submitted to *ACM SOCC'11*, May 2011.

Hamid Hajabdolali Bazzaz, Sajjad Zarifzadeh, and Caro Lucas. "Bandwidth Sensitive Routing: From Individualism to Collectivism." Under revision with *Elsevier Computer Networks Journal*, 2011.

Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. "Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers." In *Proceedings of ACM SIGCOMM '10*, New Delhi, India, August 2010.

Hamid Hajabdolali Bazzaz, and Ahmad Khonsari. "How Would Ants Implement an Intelligent Route Control System?" in *Proceedings of ASMTA'09*, Madrid, Spain, 2009.

Hamid Hajabdolali Bazzaz, Sajjad Zarifzadeh, Ahmad Khonsari, and Amir Nayyeri. "On optimizing survivable multihoming." in *Proceedings of IEEE LCN'09*, Zurich, Switzerland, 2009.

Hamid Hajabdolali Bazzaz, Marjan Sirjani, Ramtin Khosravi, and Shamim Taheri. "Modeling networking issues of network-on-chip: a coloured petri nets approach." in *Proceedings of SimuTools'09*, Rome, Italy, 2009.

Sajjad Zarifzadeh, Amir Nayyeri, Nasser Yazdani, Ahmad Khonsari, and Hamid Hajabdolali Bazzaz. "Joint range assignment and routing to conserve energy in wireless ad hoc networks." in *Elsevier Computer Networks Journal*, 2009.

ABSTRACT OF THE THESIS

Fundamental Challenges for Hybrid Electrical/Optical Datacenter Networks

by

Hamid Hajabdolali Bazzaz

Master of Science in Computer Science

University of California, San Diego, 2011

Professor Amin Vahdat, Chair

Recent research proposals on building hybrid electrical/optical networks as a way of interconnecting the core layer of a modular datacenter promise reduction in the cost, deployment complexity as well as energy requirement of large scale datacenter networks. We summarize our two years of experience in dealing with such hybrid interconnects. We first show the feasibility of building hybrid networks under homogeneous environments where one application dominates the workload through building a completely functional prototype of Helios/c-Through systems. Then we uncover a number of fundamental challenges and problems, some quite unexpected, which must be addressed for wide adoption of these proposals in industry settings under heterogeneous traffic pattern with multi-tenancy. We then

propose a significantly more flexible, fine-grained and responsive way of controlling such hybrid networks under Observe-Analyze-Act framework which is based on OpenFlow. Although the complete picture of how to build these networks in particular the best circuit scheduling algorithm to use is still unclear, but we show that this framework enables a variety of future solutions to the remaining challenges.

Chapter 1

Introduction

The speed of emergence and adoption of diverse Internet services such as email, search, MapReduce computations, and social networks has been tremendous over the past few years. Behind-the-scene of a cloud computing service like this is a datacenter spanned across tens to hundreds of thousands of servers coordinating together to handle clients requests in a cost-efficient manner. Organizations such as Google, Facebook and Amazon have already built such huge dateacenters. The main motivations for building such large-scale cloud computing infrastructures are for the most efficiency and leveraging economics of scale, i.e., amortizing the infrastructure cost for running different services over a single cloud computing infrastructure.

Traditionally, these large datacenter interconnects are built in a three-level hierarchy of switches (e.g., access, aggregation and core layers). Due to the multiples of 10Gbps upper bound on the speed of traditional Ethernet switch ports as well as the limited number of ports they have at those high rate, (for instance, at the time of this writing, Broadcom BCM56840 series has only 64 ports of 10Gbps [bro10]) the current practice is to introduce an oversubscription ratio at each layer of this hierarchy [cis07]. In fact, oversubscription ratios as high as 240:1 are not unheard of in today's commercial datacenters [GHJ+].

On the other hand, many of the applications running on the cloud need tremendous concurrent amount of bandwidth to if not all the nodes in the cluster for sure to considerable portion of them at the same time. For instance, MapRe-

duce has all-to-all communication pattern in its shuffle phase to move around the data generated locally on each node to other nodes to prepare for the map phase. However, given the high oversubscription ratio in current datacenter architecture, provision of required bandwidth at such a scale is a significant challenge. There are quite a few recent research work in networking community to come up with solutions for this problem. As a result of these efforts, quite a few SIGCOMM papers proposed new architectures for datacenter using commodity switches which promise full bisection bandwidth across the entire datacenter [WLL+, AFLV, GHJ+, GWT+, GLL+]. Section 6 overviews these proposals.

However, the huge cost and/or complexity in deployment and maintenance of these proposals is still a serious barrier for their wide adoption in very large scale datacenters. For instance, even a relatively small instance of 3456 ports fat-three needs 6921 long cumbersome cables [FRV09]. Moreover, many consider provision of full-bisection bandwidth at these scale an overkill because some research studies show that except for a few outliers, application demands can be generally met by a slightly oversubscribed network [KPB].

Hence, a reasonable alternative is to provision high bandwidth pipes only "when" and "where" needed through an additional dynamically reconfigurable switching fabric. In fact, the modular design of datacenters (also known as *Pod-based* design) provides the opportunity to augment the core of the network with an extra switching infrastructure to connect arbitrary set of Pods to each other based on the aggregated Pod-level traffic pattern at each time.

Specifically, optical circuit switching is a quite promising technology for allocating such high speed bandwidth at the scale of a datacenter. The advantage is that a single optical link can carry multiples of 10Gbps at low monetary cost. On the other hand, the drawback is that optical switches are circuit-based and they can not exploit statistical multiplexing similar to electrical packet switches. Hence, all traffic coming to the optical switch from a certain port are forwarded over a single output port and there is a "switching delay" for changing the input-to-output ports mapping of the switch. This circuit switching based forwarding introduces a new set of challenges specifically on when to change the topology of

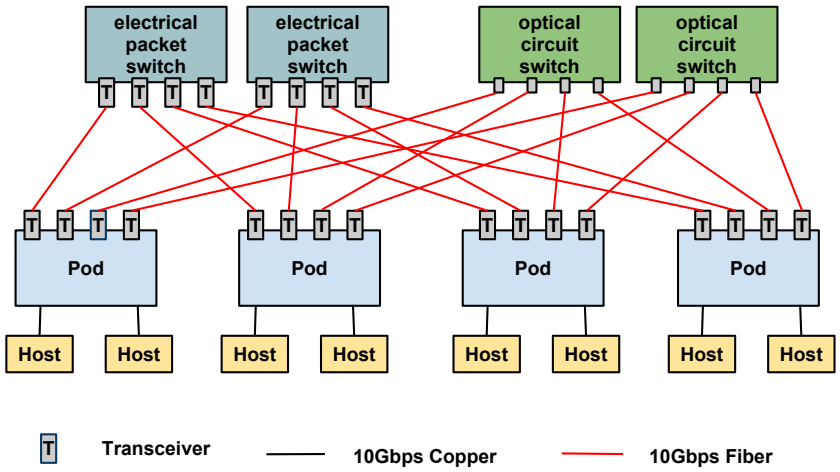


Figure 1.1: The hybrid electrical/optical switch architecture

the circuit switches and how to determine the new topology. In addition to this, as the rest of the network is built out of packet switches, unifying these two different technologies to build a *hybrid* electrical/optical network is also a challenge.

This thesis is a summary of our experience in dealing with these hybrid networks over a period of almost two years. We describe the problems we faced in building a fully-functional prototype of these hybrid interconnects as well as the solutions we came up with and possible future works in this space.

1.1 Hybrid architecture

Figure 1.1 shows a small instance of a hybrid electrical/optical network which is a two-level multi-rooted tree of Pod and core switches. Core switches can be either electrical packet switches or optical circuit switches; the idea is that the strengths of one type of switch compensate for the weaknesses of the other type. The circuit-switched portion handles baseline, slowly changing inter-Pod communication whereas the packet-switched portion is best suited for the bursty portion of inter-Pod communication. The optimal mix is a trade off of cost, power consumption, complexity, and performance for a given set of workloads. In Figure 1.1, each Pod has a number of hosts connected to it by short copper links. The Pod switch contains a number of optical transceivers (labeled T) to connect

to the core switching array. In this example, half of the uplinks from each Pod are connected to packet switches, each of which also requires an optical transceiver whereas no transceiver is required on the circuit switch side which is itself another advantage of using optical circuit switches. It is important to notice that the topology of the packet-switched portion of the interconnect is the same and the flows share the links in a statistical multiplexing fashion. However, the topology of the circuit-switched part of the network is dynamic in the sense that there needs to be a controller software to determine the input-to-output ports mapping of each individual core circuit switch over time based on the workload.

This flexible design comes with a set of challenges which needs to be addressed to make it a feasible solution. In the next section, we overview some of these high-level challenges.

1.2 Hybrid networks challenges

Although hybrid networks are promising in reducing cost and energy requirements, there are a set of issues in building these networks. Specifically, the traditional electrical switches are packet-switched whereas the optical switches are circuit-switched. This main paradigm difference perhaps is the root of most of the challenges in unifying these two technologies. In the following we enumerate these challenges:

- Performance: There is an overhead associated with changing the topology of a circuit switch. Although theoretically free-space MEMS circuit switches should be able to achieve switching times as low as a few milliseconds [Edd01], in practice the current switching time in available circuit switches is in range of (10,25)ms. For instance, see [gli]. In a datacenter setting where the traffic workload might change in the aggressive order of a few milliseconds (e.g., based on [KSG+09] half of the flows in a datacenter has a duration of as low as only 10ms), this long circuit switching time results in significantly poor performance. So, the main challenge here is how to deal with this switching delay. Also whether this design is well-suited only for stable workloads or

is there a way to deploy these interconnects under *any* traffic pattern? To this end, to use these hybrid networks "in the wild", the system should not get stalled in a rapidly changing workload where there is no considerable stability in the traffic pattern.

- Hardware: Not only the circuit-switched portion of the hybrid network needs to be configured in the scale of every few milliseconds, depending to the traffic pattern, but also other underlying hardware components needs to be reconfigured at such a short time-scale. This in particular includes the Pod switches which should be notified of the change in the destination of their circuit uplinks. This change in turn needs to be reflected in Pods forwarding table quickly.

- Scheduling circuits: A software component needs to be in charge of determining the circuit switches topology. The underlying scheduling objectives and algorithms are interesting challenging research problems. Specifically, when and how to change the topology are the main questions to explore. It is important for the scheduling algorithm to avoid needless reconfiguration of the topology due to the performance overhead each reconfiguration event incurs.

- Applications and the system interaction: Among the challenges in hybrid networks is the interaction between the applications and the controlling logic of the hybrid networks. Specifically, it is non-obvious and unstudied how different applications would perform under such an environment and how could they effect the scheduling of the system. For instance, what happens if part of the traffic of an application goes over the high speed circuit links and the rest over the packet switch network? How would that impact the progress of different applications? On the other hand, how could different applications impact the scheduling decisions made by the system? Are there any applications which could mislead the system to get into "bad scheduling decisions" and decrease the overall performance of the system?

- Multi-tenancy: How should the system perform when there are multiple

applications with different quality of service requirements? For instance, one application might need high speed and better be suited for the circuits path whereas another might need short latency and better to be forwarded over the packet network. Specially in a large cloud computing system applications might have huge differences in performance and QoS requirements. It is then a significant challenge to differentiate between these applications and also make scheduling decisions accordingly. We might also want to have the ability of making scheduling decisions at the granularity of each individual flow or per applications flows.

1.3 Thesis overview

This thesis is a summary of the experiences we earned over a period of almost two years in dealing with hybrid networks. In this chapter we gave an overview of the trends in datacenter networks architectures and explained why we believe using a hybrid electrical/optical solution is promising in this space. We then discussed the main challenges in building such an efficient hybrid network. In the rest of this thesis we give an overview of our first proposal in this space, Helios [FPR+10]. We also describe a very similar research effort, c-Through [WAK+10] which was proposed independently though in parallel with Helios. We then perform a comprehensive evaluation of these two systems under different scenarios. These evaluations base the rest of our research effort. Specifically, we get to realize (1) how the design differences in Helios/c-Through make these systems behave differently (2) what are possible scenarios in which these two systems may fail? (3) going beyond the last item, we realize what are underlying assumptions that these systems which might not be always correct (4) we then discuss at a higher layer and propose a new general framework which we denote as Observe-Analyze-Act which is better suited for managing hybrid networks (5) we also argue that a finer-grained control over the forwarding decision at the granularity of flows is needed to avoid some of the issues we observe in our experimental results (6) we provide an initial evaluation of our new proposed framework and show that it

outperforms Helios/c-Through and is more flexible, finer-grained and responsive for dealing with hybrid networks.

1.4 Organization of the rest of thesis

The rest of this thesis is structured as follows:

- Chapter 2 provides a short overview of the Helios/c-Through research efforts and a concrete set of evaluation results of different aspects of these systems such as bisection bandwidth, number of circuit flapping events, etc. We also provide certain experiment scenarios to pronounce the weakness of these systems.

- Chapter 3 discusses our evaluation results in detail and argues what are the main assumptions these systems are taking which lead into these inefficient behaviors under particular workloads. We then articulate the fundamental requirements for hybrid interconnects to get around such scenarios. We end this chapter by proposing an Observe-Analyze-Act framework which satisfies these requirements.

- Chapter 4 describes in detail the architecture and design of our prototype, this includes the software components of our system as well as the design decisions made and the reasons.

- Chapter 5 provides an initial evaluation of OpenFlow based framework to show the functionalists achievable through this design which are not feasible in Helios/c-Through systems. We also show the effectiveness of Traffic Analyzer component which is proposed for handling scheduling problems in certain workloads.

- Chapter 6 explores related works in particular domains this thesis contributes to.

- Chapter 7 concludes this thesis by summarizing the lesson earned and potential future research works in this area.

- Chapter A is an appendix Chapter which shows the results of some related experiments we performed and some tools we developed throughout this thesis. As these are not directly connected to the main story of the thesis, we decide to present them as an appendix chapter.

Chapter 2

Helios/c-Through overview and evaluation

Helios and c-Through [FPR+10, WAK+10] are the first research efforts which propose a hybrid electrical/optical network for modular datacenters which were published in SIGCOMM'10. In this section, we describe how these systems work. Our main focus here is the underlying mechanism and control plane. We then provide a comprehensive evaluation of these systems under different artificial and real traffic patterns to study their strengths and weaknesses.

2.1 Overview

Both Helios/c-Through have a logically centralized potentially physically distributed controller software component, called Topology Manager, which is in charge of managing the whole system. We save a thorough software architecture description for next chapters of this thesis. So, instead at this point we provide a high level sketch of how these systems work. These systems starts off with an initialization phase in which the Topology Manager establishes a control connection to different components in the system for the most Pods and circuit switches. After this, there is a control loop which is executed forever.

The objective of the control loop is very simple. Given the hybrid network in Figure 1.1, the goal is to maximize the amount of traffic which gets forwarded

over the circuit network. The intuition behind this is that as reconfiguring the topology of each of the circuit switches is expensive (e.g., has switching delay), it is reasonable to use these paths for long-lived aggregated traffic. As such, the control loop performs a series of operations in each control cycle to handle this objective. Specifically, the control loop starts by collecting traffic statistics from Pods/end-hosts (more detail coming). These statistics are then used to build a Pod-level traffic matrix. Then, both systems use this instantaneous snapshot of traffic to maximize circuits throughput. That is, given this traffic matrix, what topology should be used to maximize the traffic for the circuit switches. To this end, Edmonds matching algorithm [Edm65] is used for finding such an optimal topology over the next cycle. None of these systems by default incorporate history or state for sake of this scheduling. Once the next topology is determined, the Topology Manager communicates with Pods as well as circuit switches and notifies the new topology to them. And then it iterates over the same loop on and on possibly after a short period of sleep at the end of each cycle to decrease the circuit switching rate and hence the overall overhead of circuit switching.

2.1.1 Hardware and topology

We use some of the SEED cluster servers for building our prototype. The SEED project is an NSF-funded research platform at UC San Diego hosted in the CallIT2 building (see [SEE] for more information). Our prototype (Figure 2.1) consists of 24 servers, one Glimmerglass 64-port optical circuit switch, three Fulcrum Monaco 24-port 10 GigE packet switches, and one Dell 48- port GigE packet switch for control traffic and out-of-band provisioning (not shown). The 24 hosts are HP ProLiant DL380 rackmount servers, each with two quad-core Intel Xeon E5520 Nehalem 2.26 GHz processors, 24 GB of DDR3-1066 RAM, and a Myricom dual-port 10 GigE NIC. They run Debian Linux with kernel version 2.6.32.8. The 64-port Glimmerglass crossbar optical circuit switch is partitioned into 5 virtual 4-port circuit switches, which are reconfigured at run-time by Topology Manager. The Monaco packet switch is a programmable Linux PowerPC-based platform uses an FM4224 ASIC for the fast path. We partitioned two Monaco switches into two

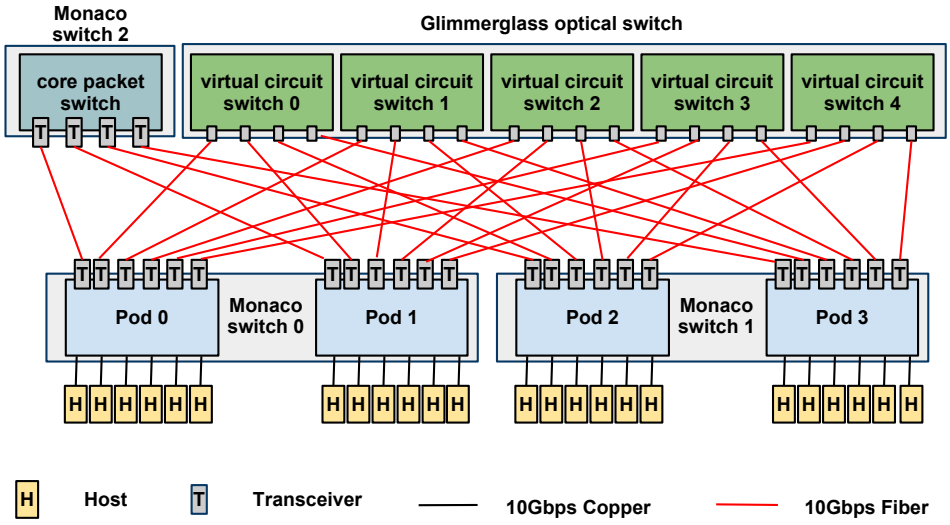


Figure 2.1: The hardware architecture of our prototype with 24 servers, 5 core circuit switches and 1 core packet switch

12-port virtual pod switches each, with 6 downlinks to hosts, 1 uplink to a core packet switch, and 5 uplinks to core circuit switches. We used four ports of a third Monaco switch as the core packet switch. These virtualizations allowed us to construct a more balanced prototype with less hardware. Note that our testbed also has some equipment for supporting wavelength division multiplexing (WDM) [MG02] technology which allows a single fiber to carry several parallel channels of 10Gbps. However, the WDM discussion is out of scope of this thesis. See [FPR+10] for related material.

In addition, as our switches do not support traffic-shaping, for slowing down the core packet switch in some experiments, we dedicate two of the servers in each Pod to generate a CBR UDP flow of rate 2.5Gbps. This slows the core packet switch network down to 5Gbps. This is done to introduce bandwidth difference between an upstream circuit to a certain Pod vs. the core packet switch path to the same Pod.

2.1.2 Helios vs. c-Through logic

Although Helios and c-Through use the same underlying design. They use different sources for estimating the network demand. Helios uses Hedera [AFRR+]

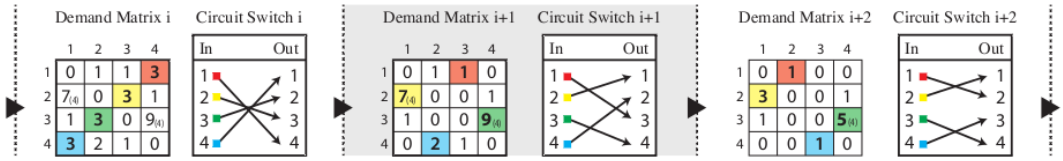


Figure 2.2: A sample illustration of circuit scheduling approach with 4 Pods

demand estimation technique which uses flow counters as observed by the Pods to estimate the demand whereas c-Through uses the socket buffer occupancy as observed by individual end-hosts to estimate the demand (more detail in Section 4.6). However, both of these systems after estimating the demand, use identical algorithm to decide the next topology to configure the core circuit switches as well as the Pod switches with. Although, it might seem like these two systems are equivalent or should behave totally the same, it is not the case. We reveal this through a comprehensive comparison-based evaluation of these systems in Section 2.2.

2.1.3 Scheduling circuits

Helios and c-Through both use a greedy approach for scheduling the circuit switches in which the optimization is done based on the instantaneous snapshot of the traffic matrix. Specifically, after the traffic matrix is built based on the source of demand in each of these systems, the circuit switches are scheduled in such a way that the total amount of traffic which traverse the circuits are maximized. We formulate scheduling each circuit switch as an instance of a matching problem in a bipartite graph as follows.

We build a bipartite graph with two sets of vertexes; an ingress set and an egress set. For each Pod, we place a node in each of these sets. We then connect the nodes in these two sets together with a weighted edge. The weight of each edge is the minimum of the bandwidth of demand from the ingress Pod (source) to the egress Pod (destination) and the capacity of the circuit uplinks (which is 10Gbps). The configuration which maximizes the demand through this circuit

switch corresponds to a maximum matching in this bipartite graph. The reason for this is that the sum of weights of the edges which are selected as the solution to the matching problem corresponds to the maximum bandwidth which can be forwarded over this circuit switch.

Once the optimal mapping for a circuit switch is found. We update the demand matrix by reducing the capacity which is already provisioned in the previous circuit switch. We then move forward with scheduling the next circuit switch with the updated traffic matrix. Figure 2.2 shows a step by step run of our scheduling algorithm over 3 circuit switches for a certain demand matrix.

It is important to realize that we schedule the circuits *unidirectionally* which means, if there is a circuit from Pod i to j, there might or might not exist a circuit from Pod j to i. The reason to perform this unidirectional scheduling is that the optimal solution is not necessarily bidirectional.

There are different algorithms for solving the maximum weighted matching problem. We chose Edmonds's matching algorithm [Edm65] which run in $O(V^3)$ for general graphs but there might exists faster algorithm for the bipartite graphs. We decide to not explore this space more as firstly the Edmonds's algorithm is fast enough for our prototype which just has four Pods and secondly as we find an available Edmonds library [edm] which we could use for implementing our circuit scheduler.

2.2 Helios vs. c-Through evaluation

Although Helios and c-Though use a similar control loop and optimization objective for scheduling the circuits, they use a completely different approach for estimating the demand matrix. Helios uses the Hedera demand estimator which takes as input the observed Pod-level traffic matrix whereas c-Through uses the buffer occupancy as observed by the end-hosts. We perform a series of experiments to understand the differences between these two systems and study the pros and cons of each of these approaches. This section provides a summary of the results of these experiments.

2.2.1 Experimental setup

We use four Pods each with three servers resulting in a total of 12 servers. There is 5 core circuit switches with one pair of links (uplink and downlink) to each Pod. The performance metric we focus on is the bisection bandwidth which has an absolute maximum of 120 Gbps if all the circuits get totally utilized. Although as we see below the maximum ideal bisection bandwidth in practice is less than that (more on this later). We vary the traffic pattern to be either Pod-level stride or all-to-all. In the case of Pod-level stride there are 8 different stages where in stage $0 \leq i \leq 7$ Pod number p sends to Pod number $(p + i + 1)\%4$. In the case of all-to-all pattern, each host sends to any other hosts throughout the entire experiment period.

We perform three different experiment settings: 1) Helios 2) c-Through 3) ideal case which is done over a non-blocking Cisco Nexus 520 network that all the servers are connected to through a separate interface.

In addition to this we vary the α parameter for determining the bandwidth of each flow for the case of Helios and study its effect. This parameter determines the damping factor of the Exponentially Weighted Moving Average (EWMA) [ewm] schema which Helios uses for calculating the rate of flows based on past history. Specifically, if r_f denotes the rate of flow f, then $r_f^{new} = \alpha r_f^{measured} + (1 - \alpha) r_f^{old}$. Thus, the smaller the damping factor α is the more value is given to the past rate of the flow in estimating its future rate. The default is to set $\alpha = 1$ which takes into account no history.

For the case of c-Through one parameter of interest is the socket buffer size which we vary on all of the servers from the default value (4MB) to 100MB. In a Linux operating system the following sysctl (Figure 2.3) can be used for this purpose:

```
/sbin/sysctl -w net.core.wmem_max=100000000
/sbin/sysctl -w net.core.wmem_default=100000000
/sbin/sysctl -w net.ipv4.tcp_wmem="4096 100000000 100000000"
```

Figure 2.3: Sysctl to change the socket buffer size for c-Through

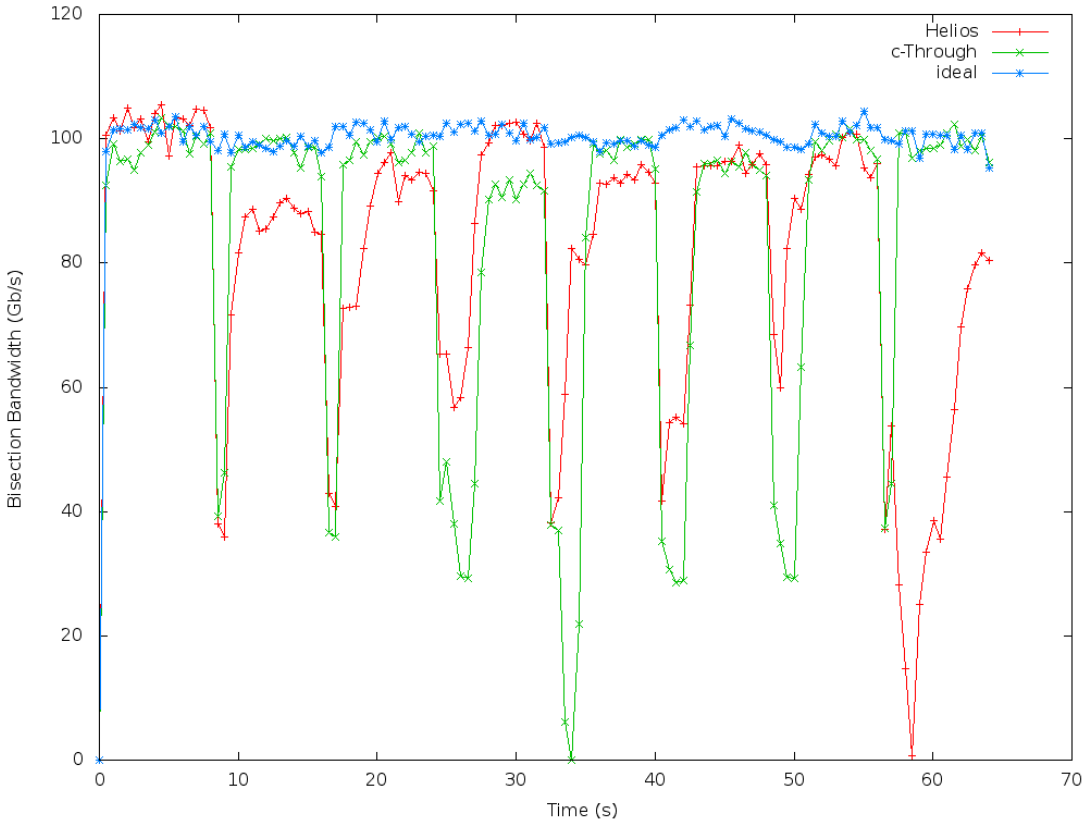


Figure 2.4: Bisection bandwidth of Helios, c-Through and ideal case with Pod-level stride traffic pattern. In this case, the average bisection bandwidth for Helios, c-Through and ideal case is 87Gbps, 89Gbps, 107Gbps, respectively.

2.2.2 Results for the Pod-level stride traffic

Figures 2.4 shows the bisection bandwidth for Helios, c-Through and the ideal case for the Pod-level stride overlaid on a single graph. The ideal bisection bandwidth over time corresponds to the performance of a non-blocking Cisco switch to which all of our servers are connected through a separate interface. We make a few observations. Although the maximum ideal bisection bandwidth is 120Gbps (12 servers each sending at 10Gps), however the achieved performance over the Cisco switch is always less 110Gbps. This is due to TCP congestion control algorithm which changes the rate of each flow individually and dynamically resulting in not completely utilizing the available bandwidth of the links. As seen in the figure, Helios and c-Through perform quite similar in terms of bisection bandwidth in this case and both compared to the ideal case are off by around 15Gbps

on average. The deep gaps in the graphs associate with the circuit reconfiguration events. This also explains why the maximum bisection bandwidth during these events is 40Gbps. The reason for this is that once the circuits are being reconfigured the only available paths across Pods are those of the core circuit switch which has one 10Gbps link to each of the Pods; resulting in a total of 40Gbps bisection bandwidth. Another point to notice is that the bisection bandwidth achieved at each different stage of the Pod-level traffic pattern is different. This is due to the hashing effect. At each stage different set of flows are hashed over different upstream circuit uplinks resulting in different bisection bandwidth. So, this in fact is an instance of problems with hashing which we discuss in Section 2.5.

2.2.3 Results for the all-to-all traffic pattern

Figure 2.5 depicts the bisection bandwidth over time for Helios, c-Though and the ideal case for the all-to-all traffic pattern. There are a few interesting points to notice. First that the ideal case performs very similar to the case of Pod-level traffic which is reasonable given the non-blocking nature of the Cisco switch. On the other hand, c-Through achieved bisection bandwidth is fairly stable (around 82Gbps on average). The reason for this is that as there is traffic from each Pod to every other Pods, the aggregated Pod level socket buffer occupancy across different Pods is very similar, as a result the number of circuit reconfiguration events is very low (a total of 14 in this case). On the other hand, Helios which uses the momentary flow rates as the source of demand estimation ends up changing the circuits too often (128 times) resulting in a significantly low bisection bandwidth. The reason is that the instantaneous flows rate is changing rapidly at the short period scale of the control loop (around 200ms). However, if flows rate is estimated over a longer time period they would seem more stable. This is exactly what happens when we use different α values for estimating the flows rate. Figure 2.6 shows the bisection bandwidth for α being 1, 0.8 and 0.6, respectively. It is observed that as α decreases (the past rate of the flow is more valued in estimating its rate) the oscillation in the bisection bandwidth reduces as well. To this end, $\alpha = 0.6$ completely smooths out the unnecessary flapping events and achieves a bisection

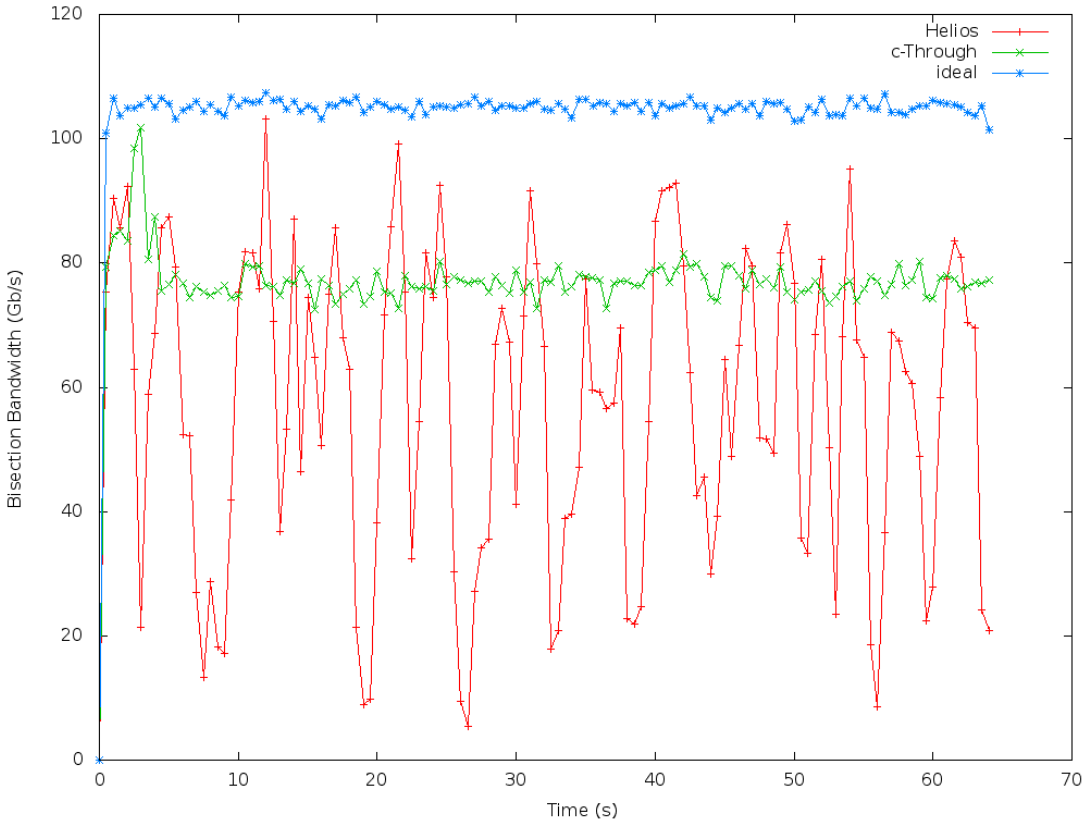


Figure 2.5: Helios, c-Through and ideal bisection bandwidth over time with all-to-all traffic pattern. c-Through and ideal case achieve a fairly stable bisection bandwidth of around 82Gbps and 112Gbps, respectively whereas Helios reconfigures the topology too often getting only an average of 61Gbps.

bandwidth of 110Gbps which is completely close to the ideal case.

We also study the impact of the socket buffer size on the performance of c-Through. Figures 2.7 shows the bisection bandwidth for c-Through when the buffer size is set to 4MB and 100MB (overlaid). In these cases, as seen in the figure the achieved bisection bandwidth match up closely and we don't see much differences in the bisection bandwidth (82Gbps vs. 81Gbps). However, we realize that the number of circuit flapping events reduces from 14 to 5 as we increase the buffer size. The reason for this is that as the socket buffer size increases there is more stability in the Pod-level traffic matrix which in turn reduces the number of circuit flapping events.

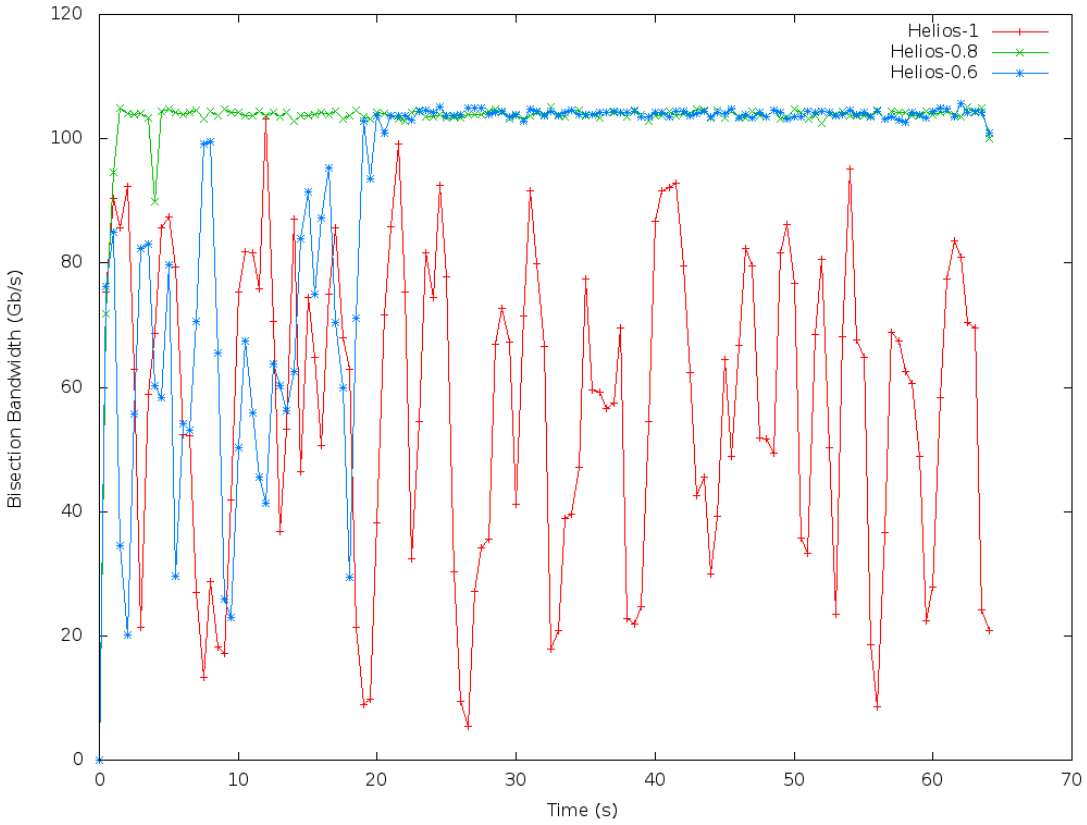


Figure 2.6: Helios bisection bandwidth over time with all-to-all traffic pattern with three different EWMA α parameter (1, 0.8, and 0.6) for flows rate estimation.

2.2.4 Summary of the comparison experiments results

Tables 2.1 and 2.2 summarize the results of the Pod-level and all-to-all traffic patterns for Helios, c-Through as well as the ideal case. Largely speaking Helios and c-Through perform very similar in terms of achieved bisection bandwidth. However, c-Through is more stable by nature and does less unnecessary circuit flapping. We however observe that using EWMA technique for estimating the rate of individual flows is quite promising for Helios/Hedera system. In particular, it stablizes the demand estimation and reduces the number of circuit reconfiguration events.

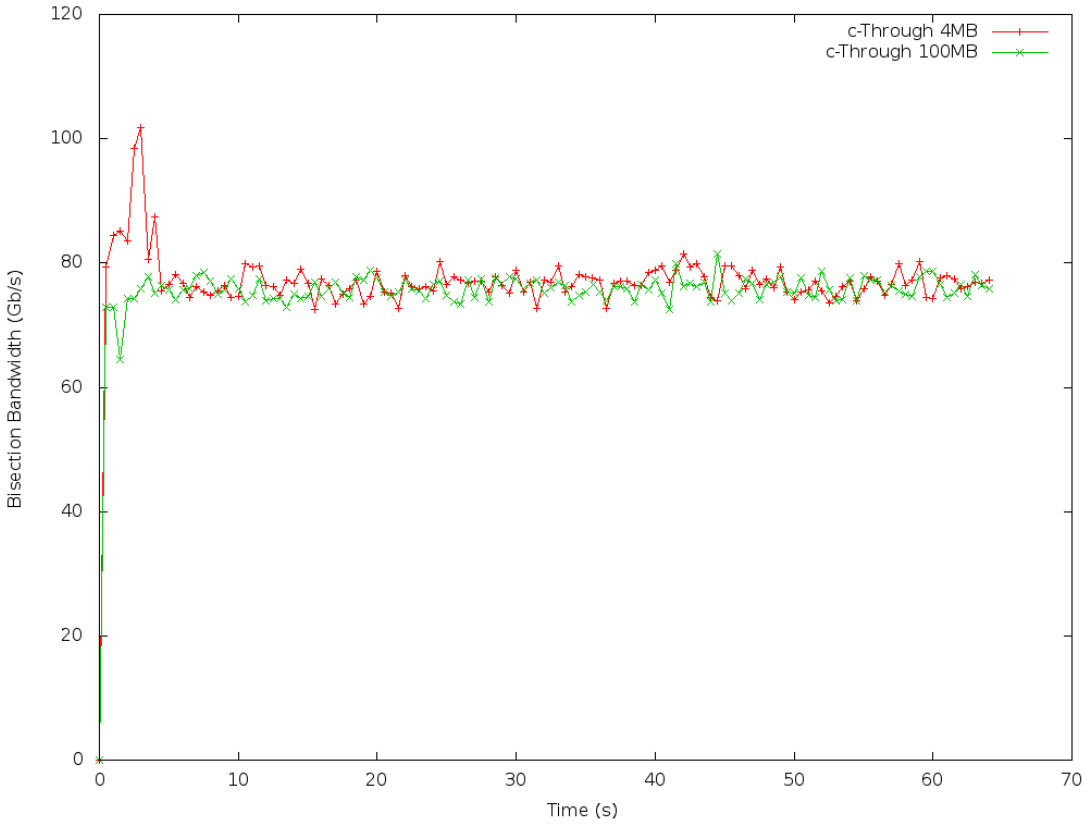


Figure 2.7: c-Through bisection bandwidth over time with all-to-all traffic pattern for the TCP socket buffer occupancy size of 4MB and 100MB

2.3 Mixed workload (TritonSort plus background traffic)

We next consider using a mixed workload which consists of TritonSort [RPC+11] plus some background traffic over Helios network. There are multiple reasons to pick TritonSort as the main application in this case. First it can drive network bandwidth much faster than what other applications such as Hadoop Sort job does over our testbed (See Appendix A.1). Second, it is a balanced system and it makes progress only at the speed of its lowest flow. Due to this allocating circuits with higher bandwidth to only some of its flows, will not improve the overall performance as TritonSort would be bottlenecked by the slower flows which are being forwarded over the packet network. This results in a situation where circuits extra bandwidth are wasted. Specifically, we would like to study such a

Table 2.1: Average bisection bandwidth and number of circuit reconfiguration events for the Pod-level traffic pattern experiments

Experiment	Avg. bisection bandwidth	# Circuit change events
Helios $\alpha = 1$	87Gbps	23
c-Through 4MB	89Gbps	15
ideal	107Gbps	–

Table 2.2: Average bisection bandwidth and number of circuit reconfiguration events for the all-to-all traffic pattern experiments

Experiment	Avg. bisection bandwidth	# Circuit change events
Helios $\alpha = 1$	61Gbps	128
Helios $\alpha = 0.8$	98Gbps	36
Helios $\alpha = 0.6$	110Gbps	5
c-Through 4MB	82Gbps	14
c-Through 100MB	81Gbps	5
ideal	112Gbps	–

scenario and see how is the performance of the application over the Helios network compared to that of a non-blocking switch.

The experiment setting is that there is one core packet switch with 5Gbps links and a core circuit switch with 10Gbps links. TritonSort is run over three Pods, specifically Pod0, Pod2 and Pod3. We also run a long lived background TCP flow from a host in Pod1 to another in Pod2 (different from hosts running TritonSort). This TCP flow competes with TritonSort for the only circuit link available from Pod1 to Pod2 and the output of the scheduler is such that it gives the circuit to this flow in case of ties. Figure 2.8 shows the duration of completion time of this experiment for three different scenarios: (1) over a non-blocking 10Gbps switch (2) over Helios network without any background traffic (3) over Helios network with this background TCP flow. It is observed that the first and second cases are mostly identical and TritonSort performs well over Helios network without any background traffic. However, in the presence of a background competing flow the

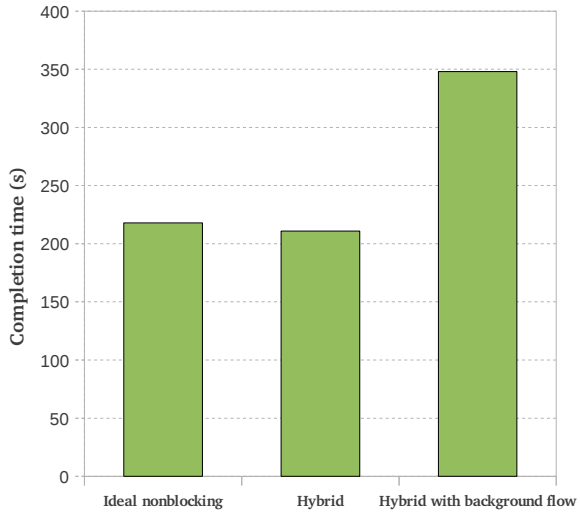


Figure 2.8: The duration of a 900GB sort over ideal nonblocking switch, Helios and Helios with a background flow

duration of the sort is much longer. In fact, in this case the duration can increase by as high as 69 percent of the ideal case. This is due to the balanced nature of the TritonSort system. As such, although circuits are allocated to part of its traffic, because the rest is forwarded over the slower packet network, the entire system is slowed down.

This experiment makes the case for the necessity of a more complex scheduling mechanism in such hybrid networks. In particular, the only way to not get stuck into such a sub-optimal circuit scheduling solution in mixed workloads like this is that the scheduling procedure would figure out the dependencies across flows in the system and assign circuits such that either all dependent flows are forwarded over circuit paths or all over the packet network. We discuss this in more detail in Section 3.

2.4 Demand estimation challenges

Estimating demand at a scale of a datacenter is a really challenging problem. Alfares et al. propose Hedera system [AFRR$^+$] which generally improves the bisection bandwidth through estimating the *actual* demand (as opposed to the

observed traffic matrix) of each individual server in the cluster. Helios uses this approach for estimating Pod-level demand but as we see in this section, applying this technique in such a hybrid network might lead into suboptimal circuit scheduling decisions.

The main idea behind Hedera is to figure out the maximum bandwidth each flow could saturate if it was routed over a path with infinite capacity and then to schedule flows over the available paths with objective of increasing the bisection bandwidth. As Hedera assumes that the bottleneck in the bandwidth is the network not the application (i.e., an application is willing to send as fast as the NIC rate), it might overestimate the rate of a flow. The drawback of this overestimation in hybrid interconnects like Helios/c-Through is that if that flow is a bursty traffic with an on-off pattern, it could potentially lead to frequent reconfiguration events as the estimated demand is the major input for the circuit scheduling component.

Consider the following experiment setup. There is one circuit switch and one packet switch. Suppose the traffic load for Pod1 consists of a long lasting flow to Pod0 and a bursty traffic with certain on and off period to Pod2. Each of these two flows in Pod1 are sourcing from a separate node. In this case, Hedera estimates the rate of each follow to be 10Gbps which is the rate of the end-hosts NIC. However in practice a bursty follow with an on/off pattern doesn't necessarily take up to 10Gbps. By overestimating the rate of the bursty flow to 10Gbps, the long lasting flow and the bursty one compete in getting the only single circuit of Pod1. And if the on/off period of the bursty flow is short, the circuit could get reconfigured forever. In the following we discuss the results for this experiment with and without Hedera demand estimator.

Figure 2.9 shows the throughput of Pod1 circuit and packet switch when the observed traffic matrix is used for the input into the scheduling algorithm. This graph is generated with the Monaco ports logger script (see Section 2.6.1 for the description of this script). In this experiment, the bursty flow has an on period of 10ms and an off period of 2sec. In addition, the long lasting flow is an iperf TCP flow and the bursty flow is generated using the bursty load generator (Section 2.6.2

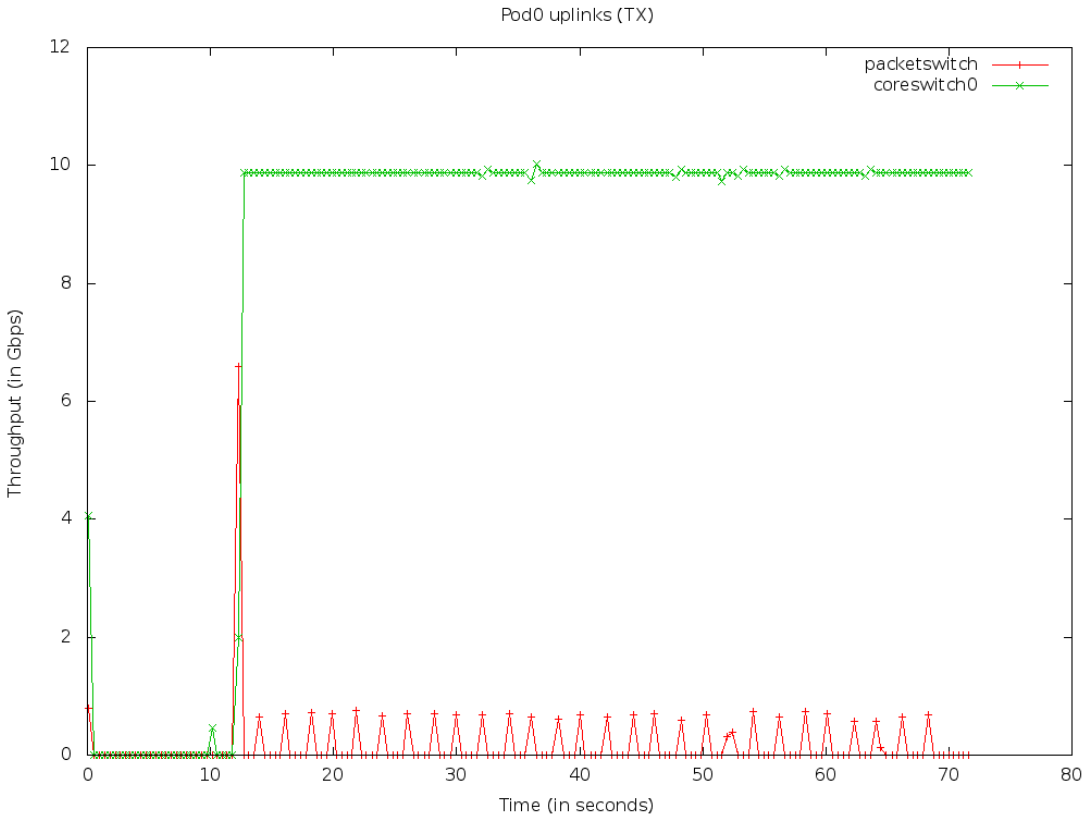


Figure 2.9: The throughput of Pod0 packet switch and circuit switch links when natural demand is used for scheduling the circuits.

describes this tool). As seen in the graph, the circuit link is completely utilized and there is no flapping event. The bandwidth ticks over the packet switch link corresponds to the bursty traffic and the steady traffic over the circuit link is for the long lasting flow. On the other hand, Figure 2.10 shows the throughput Hedera achieves for the same experiment. Here, the bursty traffic demand is estimated to 10Gbps which then competes for getting the circuit. In this case, there is 67 circuit flapping events whereas using the observed traffic as the source of scheduling does not change the circuits at all. We realize through increasing the on period of the bursty traffic that the same trend continues to happen till on periods of around 60ms. Once the on period is bigger than 60ms even using observed traffic matrix results in some circuit flapping. The reason for this is that in this case sometimes the observed rate of the bursty traffic is close to the rate of the long lasting flow which would result in circuit flapping (Note that the experiment scenario is built

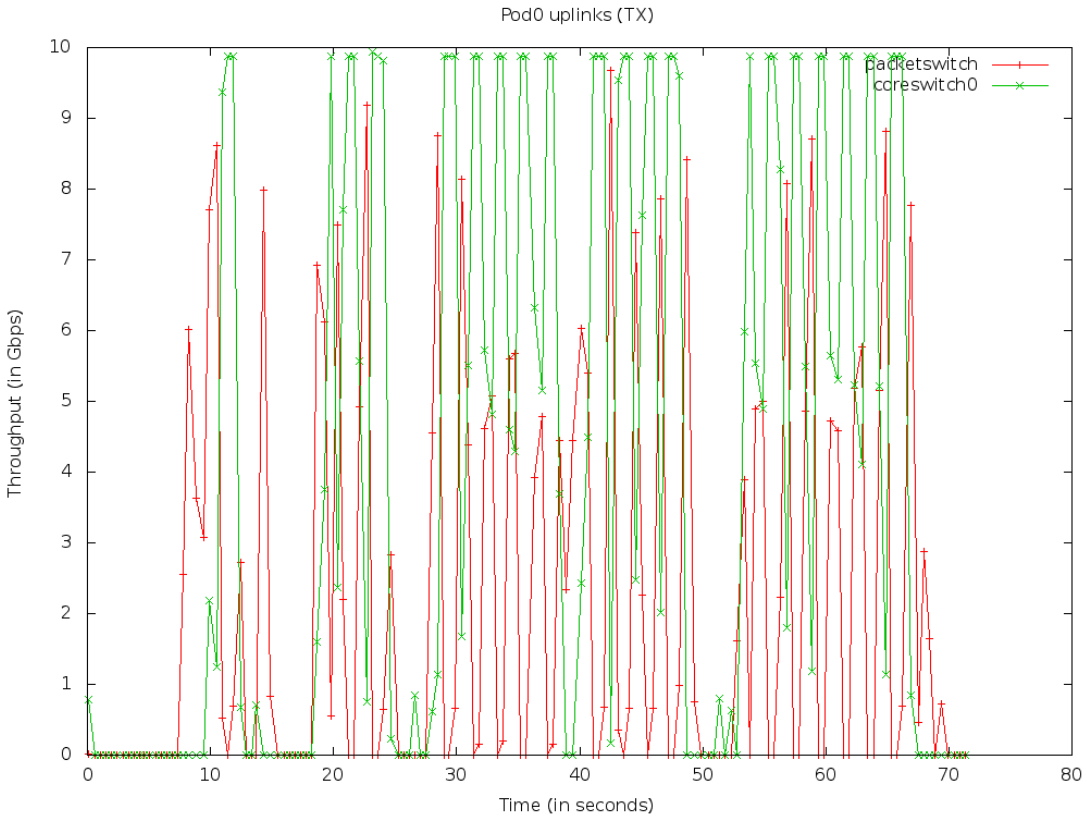


Figure 2.10: The throughput of Pod0 packet switch and circuit switch links when Hedera is used for scheduling the circuits.

in such a way that the scheduler assigns the circuit to the bursty flow in case of ties). As the on period of the bursty traffic increases the number of circuit flapping events increases dramatically, resulting in much worse performance. For instance, Figure 2.11 shows the throughput for cases where on period equals to 80ms and 1sec. There is 18 circuit flapping events in the first case and 54 in the later. The reason that there is so many circuit flapping in the later case is that the bursty traffic is getting totally competitive rates to that of the long lasting flow. As the flows rate oscillate up/down each other too many circuit flapping events occurs.

Figure 2.12, 2.13 and 2.14 show a summary of the impact of the bursty traffic for Helios and c-Through for different on period of the bursty flow from 1ms up to 2000ms. Specifically, Figure 2.12 depicts the normalized number of circuit flapping events as the on period of the bursty flow increases. It is interesting that c-Through is less sensitive to bursts of on period of less than 40ms. The reason for this is that

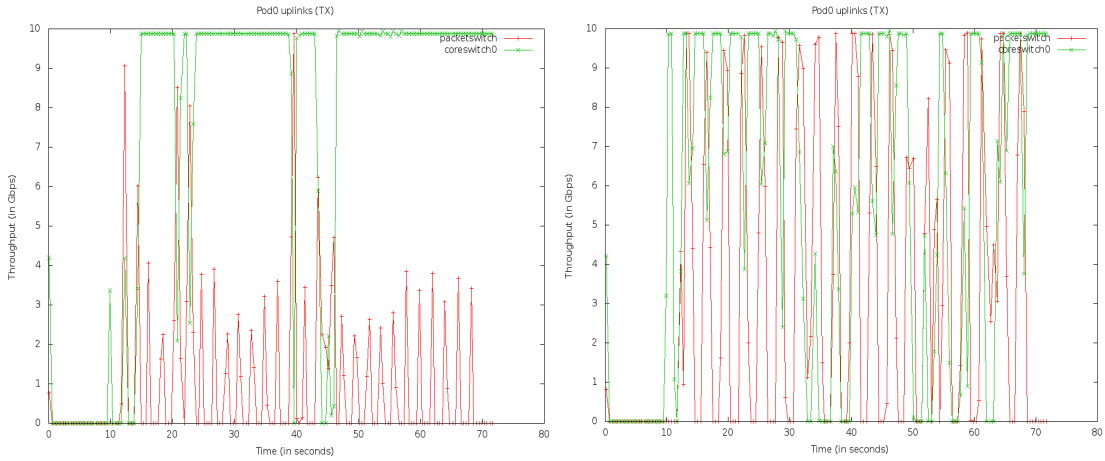


Figure 2.11: The throughput of the Pod1 circuit and packet switch links when natural demand is used for scheduling with a background burst flow with on period of (a) 80ms (b) 1sec.

c-Through uses socket buffer occupancy information to make scheduling decisions and for short on periods the socket buffer occupancy value is quite negligible if not zero. Overall, the impact is more pronounced in the Helios due to using the Hedera demand estimator. It is also obvious that increasing the on period, augments the number of circuit flapping events as both the rate of the flow and socket buffer occupancy values would get increased.

Figure 2.13 shows the average throughput the long lasting flow can get under different values of the on period. Apparently, increasing the on period decreases the throughput the long lasting flow gets due to circuit being stolen by the bursty flow. Moreover, all the variation in the rate the flow faces over each circuit flapping event is another reason for this decreasing trend. Ideally the throughput that this flow can get at each time is either close to 10Gbps if being routed over the circuit link or 5Gbps if being forwarded over the packet switch link. This explains why the throughput of the flow changes from 10Gbps to 5Gbps as the burst duration increases.

Moreover, Figure 2.14 shows the throughput of the circuit link in both systems. As the duration of the bursty traffic increases the utilization of the circuit initially decreases. However, the utilization starts to increase at burst duration of

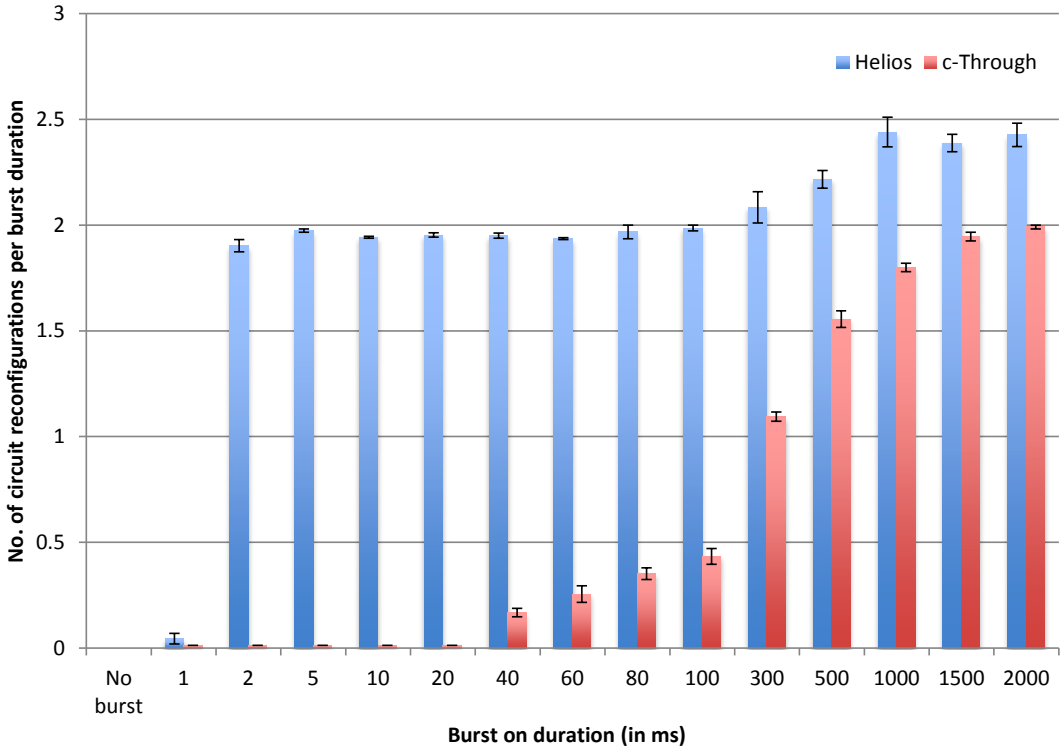


Figure 2.12: Number of circuit reconfiguration events for different burst duration

1000ms. The reason for the initial decreasing trend is that the circuit is assigned to the burst traffic which can not utilize it completely. Furthermore, circuits get reconfigured only after every completion of the control loop cycle. As such, bursts smaller than the control loop will cause the circuit to be under utilized for that part of the control loop cycle. As the duration of the bursty traffic increases, it sends more traffic over the circuit allocated for it and the overall utilization of the circuit increases as well.

2.5 Hashing effects

Multi-path routing is being increasingly used in datacenter networks for reliability and performance reasons. When there are several equal-cost paths available, the upstream port for each flow is determined by hashing different fields of the flow header. This hashing which is implemented in either hardware or software is done to ensure that the packets of each flow take the same end-to-end path to

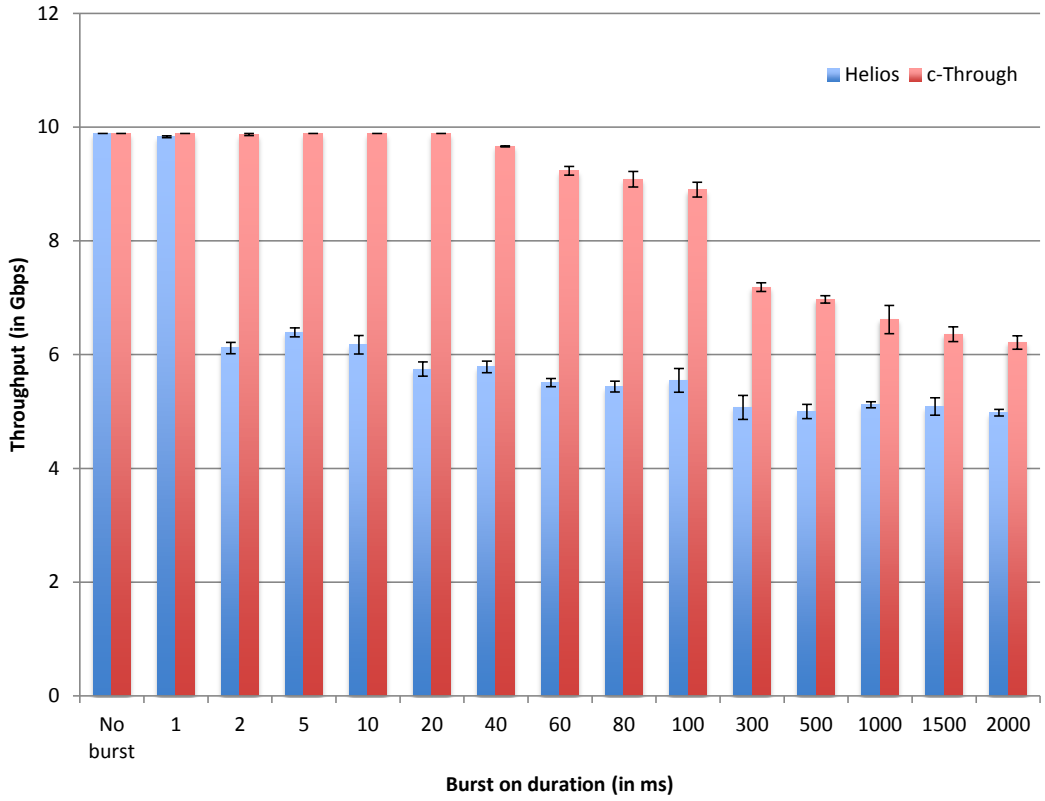


Figure 2.13: The average throughput the long lasting flow gets as the duration of the background bursty flow increases.

avoid TCP re-ordering issues. Although, this hash-based loadbalancing approach has shown to work well when the number of flows is large, but there is no guarantee of fairness or optimal performance under all workloads. Here we present a few experiments to present the hashing issues we have observed over the course of building Helios prototype. Specifically, in a workload where most of the flows are mice (small) with only a few elephant flows, hashing uniformly across the entire set of flows can result in elephants flow receiving too little bandwidth. We study this setting through the following experiments scenario.

There are 4 circuits from Pod0 to Pod1 and 4 hosts in Pod0 are sending traffic to 4 hosts in Pod1. Specifically, each host in Pod0 has a number of flows to one of the hosts in Pod1. We increase the number of flows and measure the throughput of the circuits. Figure 2.15 shows the throughput gained on the hosts in an experiment where there is 1 flow between every pair of hosts. As seen in

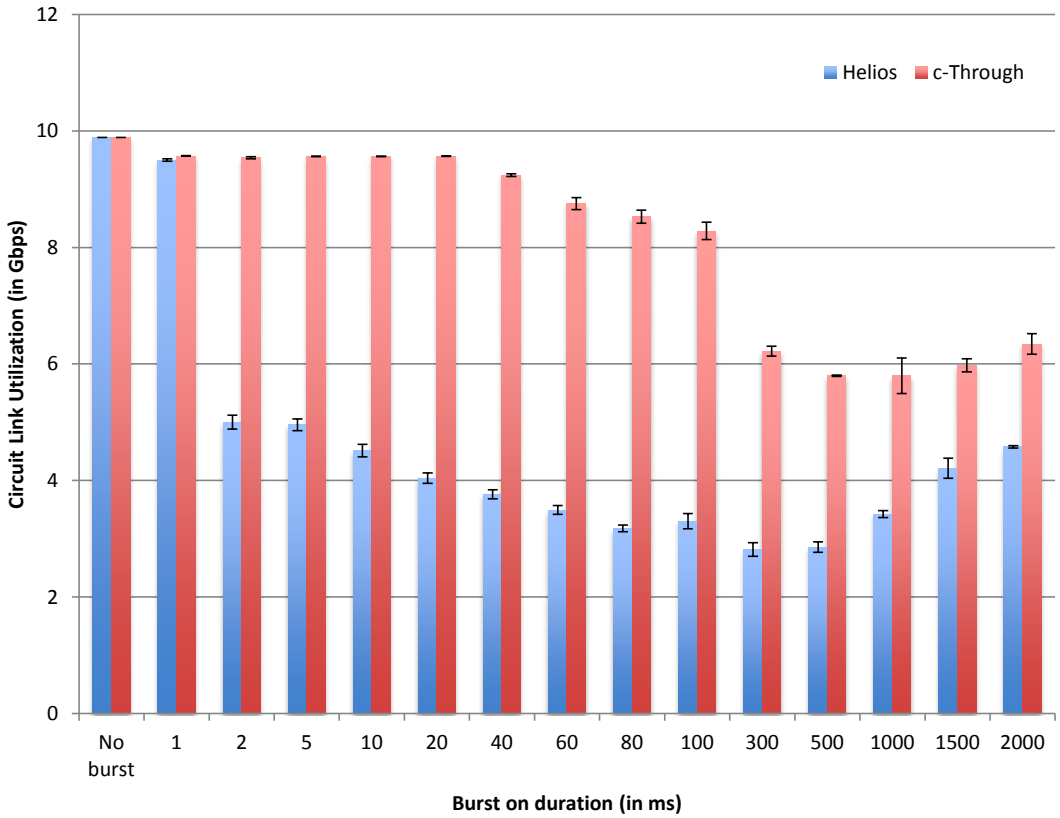


Figure 2.14: The throughput of the circuit link

the figure, two of the hosts get unlucky and their flows are hashed into the same output port and as a result each can on-average send only at 5Gbps, i.e., at half of the two other flows rate. This results in both low performance and lack of fairness.

We repeat the same experiment setting increasing the number of flows to 16 in total (each host sourcing 4 flows). Figure 2.16 shows the throughput over each of the 4 circuits across the two Pods. From these 16 flows, 4 are hashed into circuit0 and 4 into circuit1. However, 5 flows get hashed into circuit2 and only 3 to circuit3. And because each flow is rate-limited by the end-host link to 2.5Gbps, this scenario leads to a very unfair situation. Although, both circuit0 and circuit1 are fully utilized to 10Gbps but circuit3 is only utilized by around 7.5Gbps and circuit2 oscillates too much above and below 9Gbps. This undesirable differences in utilization of circuit links comes down to the unpredictability and randomness of the hashing function.

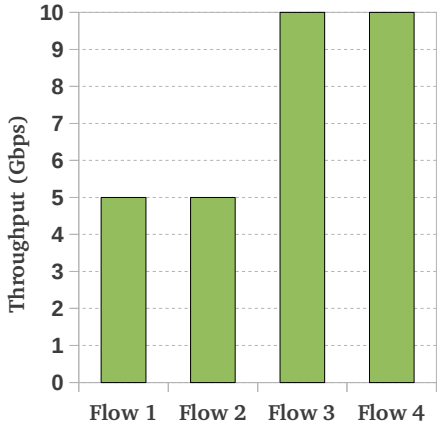


Figure 2.15: Throughput of 4 flows which are hashed over 4 circuit uplinks. Two of the flows are hashed over the same circuit, each only getting 5Gbps.

We then increase the number of flows to 64 in total (16 flow from each host). Figure 2.17 shows the circuits throughput in this case. The hashing effect is less pronounced and all the 4 circuits are being used almost at their maximum rate. This is expected because as we add more flows, the flows are more uniformly distributed from a statistical perspective and this should remove the drawbacks of imbalanced hashing.

2.6 Auxiliary components

This section provides a short document on some of the major tools which we developed throughout the duration of this thesis for the purpose of running and debugging experiments.

2.6.1 Monaco ports logger

The Monaco ports logger script is a tool which logs the TX/RX rate of each of the ports of the Monaco switches in a file throughout the duration of an experiment. This is used for capturing the throughput of Pods links and generating graphs such as Figure 2.11. For doing this the script pulls the TX/RX counters of each of the ports of the Monaco switches every certain milliseconds by a remote

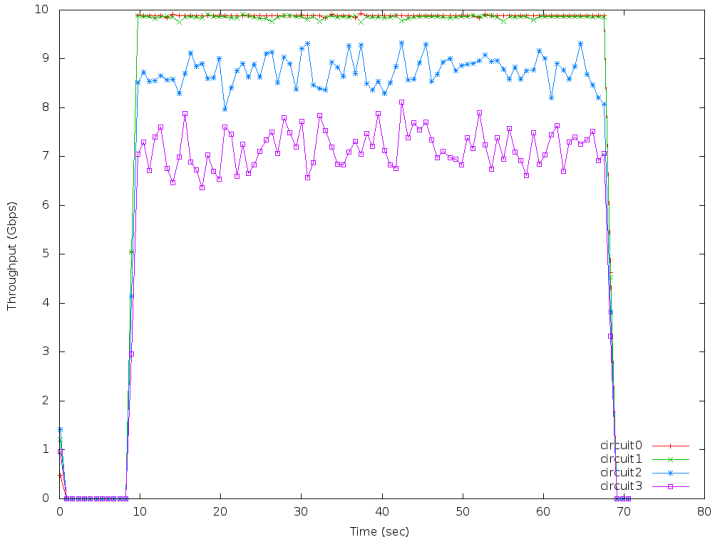


Figure 2.16: Hashing effects for 4 circuit uplinks that 16 flows are hashed over. Due to randomness of the hashing function the circuits are not utilized fairly.

procedure call and calculates the traffic rate (the difference between the new counters and last ones divided by the time interval associated with each counter) over each of the Moncao ports.

This tool which is written in Python was developed to observe the traffic rate over the optical fiber links, however as each of the end hosts are also connected to the Monaco switches we could also capture the end hosts traffic rate using this tool. Another tools was written to parse the log file of this script and generate different graphs of end hosts and circuit switches TX/RX. Figure 2.18 shows the usage of this script.

```
Usage: log_monaco_ports_rate.py <superman config> <log dir> [options]
Options:
  -h, --help
        Show this help message and exit.
  -p PERIOD, --period=PERIOD
        The period of logging in msec. The default is every 200 msec.
  -d DURATION, --duration=DURATION
        The duration of running the script in sec. The default is to run forever.
  -w, --wait
        Waits for a signal on port 5000 to start logging.
  -v, --verbose
        Show verbose output.
```

Figure 2.18: The Monaco port logger script

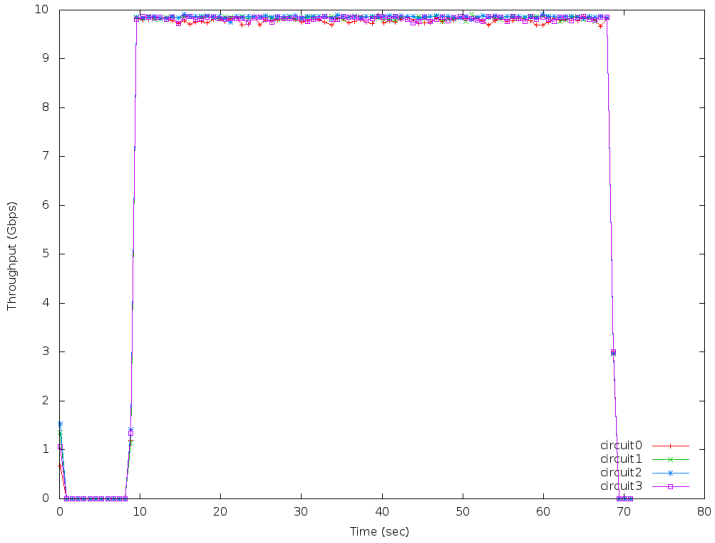


Figure 2.17: Hashing effect is less pronounced when 64 flows are hashed over 4 circuit uplinks. The figure shows the throughput of each of the circuits which is almost 10Gbps.

2.6.2 Bursty load generator

The bursty load generator tool is a typical TCP client-server socket program written in C which is used in different experiments such as the one in Section 2.4. This tool generates background bursty traffic of desired on and off period. The server should be running before the client and then the client should get started by passing the IP address of the server. The client once started makes a TCP connection to the server. Then the server starts a timer to switch between on and off periods. Over the on cycles the server side sends data over the established socket channel which the client side acknowledges. Over the off periods no data is being transmitted. Figure 2.19 shows the usage of the bursty load generator tool.

```
Usage: bursty-loadgen
Options include:
  -c <server ip> for the client mode
  -s to run in the server mode (default)
  -d to set the duration in seconds (the default is to run forever)
  -n to set the on duration in mseconds (the default is 500)
  -f to set the off duration in mseconds (the default is 500)
  -v to set verbosity level [0,3]
```

Figure 2.19: The bursty load generator tool

Chapter 3

Discussion; the Observe-Analyze-Act framework

In this chapter we first discuss about the origin of the problems we observed in Helios/c-Through evaluation in Chapter 2. In particular, we articulate three separate lists which summarize our experiences: (1) deep assumptions that Helios/c-Through systems in particular hold which cause them to end-up in a sub-optimal solution in certain workloads. (2) fundamental challenges in building hybrid networks in general (3) software-engineering challenges in our specific prototype. We then argue for some fundamental design decisions to avoid these drawbacks. To this end, we also propose an Observe-Act-Analyze framework which accommodates such design decisions and promises a more fine-grained and flexible control loop for hybrid networks.

3.1 Helios/c-Through underlying assumptions

By looking into the evaluations results presented in Chapter 2, we realize that the reason Helios/c-Through get into some sub-optimal scheduling decisions in certain cases comes down to some deep underlying assumptions these systems make. Specifically, these proposals make five overly restrictive assumptions about the traffic on the hybrid network as follows:

- Flows are independent and uncorrelated: Assigning circuits between two Pods will not affect the bandwidth desired by other flows in the system.
  Reality: Many flows depend on the progress of other flows, such as TritonSort flows.

- All flows have same priority: Helios and c-Through schedule optical circuits without taking into account any application or flow level traffic prioritization.
  Reality: Datacenter traffic has a rich mix of priorities and fair sharing desires within and between applications.

- Flows will not underutilize the circuits: Once flows are assigned to circuits, they will continue to use the same (or more) bandwidth as their predicted demand indicated until the control process reassigns that circuit elsewhere (which might not happen for hundreds of milliseconds or more).
  Reality: Perfectly predicting future traffic demand without taking explicit feedback from the application, if not impossible is certainly not as easy as this assumption.

- Randomly hashing flows to optical circuits is effective: Given a set of N 10Gbps circuits between two Pods, we can create a single *superlink* of $10N$ Gbps capacity using random hashing, in which flows are randomly hashed across the set of available circuits.
  Reality: Random hashing is not always effective in leveraging $10N$ Gbps superlink capacity due to hash collisions. Moreover, random hashing cannot support flexible distribution of circuit bandwidth that maximizes application performance.

- All flows that can use a circuit should use that circuit: There is no cost to using an optical circuit when it is available, or to switching flows between the electrical and optical circuits. The Helios design made this assumption because of switch software limitations—traffic could not be scheduled on a per-flow basis to the electrical or optical network. The c-Through design made the same assumption to keep its optical/electrical selection tables small using only per-destination entries.

Reality: There are cases where keeping some lower-bandwidth, latency-sensitive flows on the packet switch reduces latency and variance for some applications.

3.2 Hybrid networks challenges

Besides the specific assumptions Helios/c-Through has made which lead to undesired behavior in certain situations. There are some fundamental challenges which make integration of electrical and optical networks hard. In the following, we provide a set of high-level challenges which needs to be addressed to make wide adoption of these networks feasible.

- **Scheduling**: Both Helios and c-Through use a simple greedy approach based on the instantaneous Pod-level traffic matrix to decide the next topology. That means the only objective in these systems is to maximize the momentary traffic which traverses the circuit switches. This although at the first glance seems like a reasonable objective but has some side effects. For the most, the instantaneous best topology might be a local-maxima not a global maximum in terms of maximizing the total bisection bandwidth. This would hurt the system overall performance considering a longer period of time. In addition, using the raw measured traffic matrix without any processing might result into wrong scheduling decisions. For instance, as shown in Section 2.4 a bursty background traffic can easily disturb this circuit scheduling policy, resulting in continuous circuit flapping which in turn leads to significantly poor performance. Moreover, it is critical to consider the effect of correlated flows into the scheduling mechanism. Assigning circuits to only part of some tightly correlated flows may decrease the overall performance of the system as opposed to increasing it as studied in detail in Section 2.3. These are only a few example of type of circumstances which might happen due to a naive circuit scheduling policy. Thus, we argue that the circuit scheduling is a significant challenge in hybrid networks.

- **Hashing**: Many of the existing commercial switches today including the Fulcrum Monaco Pod switches we deployed in our prototype, use layer two

Link Aggregation (LAG) or IEEE 802.1AX standard [LAG08] for combining multiple physical links between two switches into one logical link for loadbalancing across these links. Specifically, a LAG is created for each destination Pod and the ports which are connected to that Pod are added into that LAG (this set changes dynamically over time based on the circuit configuration). Then, forwarding decision for each individual flow is done by matching it against a LAG and for determining the actual port to forward the flow over in the set of ports which belong to the matched LAG, hashing over layer 2, 3 and 4 headers of the flow is used.

This although being simple has shown to be unfair, inefficient and problematic in some aspects: 1) The hashing function might not lead to a fair distribution of flows across the available links resulting in unfairness as well as poor utilization of circuits available bandwidth. We studied this in Section 2.5 in detail. 2) There is absolutely no fine-grained control over the path each individual flow is forwarded over when there are several upstream paths as this is completely handled by the switch hardware/software hash function. This is a significant limitation because this prevents any fine-grained control at the granularity of flows. In addition, this is a barrier to provide QoS or similar services.

- **Hardware**: There are a few hardware hurdles to achieving fast and efficient network reconfiguration in hybrid networks. The major problem is that current optical components are not designed to support rapid reconfiguration due to their original requirements in the telecom industry, and thus there has been a lack of demand to develop faster optical control planes.

  The switching time of current optical switches are still far from their ideal limits. The best switching time is still around 10 to 25 ms for commercially available optical switches [gli]. However, free-space MEMS switches should be able to achieve switching times as low as a few milliseconds [Edd01]. Another hardware issue comes from the design of optical transceivers. Currently available transceivers do not re-sync fast enough after a network reconfiguration. The issue is that the time between light hitting the photoreceptor

in the transceiver, until the time that an actual data path is established is needlessly too long–as long as several seconds in practice [FRV09]. The underlying transceiver hardware can theoretically support latencies as low as a few nanoseconds. Beyond the physical limitations, much of the control plane performance limitations are due to the electrical control aspects of the design, including the implementation of the switch software controller.

The improvement in both optical transceivers and switching components is crucial in accelerating the adoption of hybrid networks in the cloud. Non-MEMS optical components show promise in delivering switching times in the microsecond range. This is small enough to hide the physical network reconfiguration from a large class of upper layer protocols and applications which will significantly improve the performance of hybrid networks.

3.3 Helios prototype specific challenges

In addition to the previous lists which were general to hybrid networks, over the course of implementing a completely functional prototype of Helios, we faced a series of software-engineering issues which we summarize below.

- One of the problem is again with hashing which is particular to our Monaco switches hardware. Specifically, the hashing doesn't take place over both circuit switch and packet switch paths. In other words, once there is a circuit established across a pair of Pods, it is impossible to forward any flows over the packet switch network. Although the packet switch network is better suited for bursty traffic as well as the delay-sensitive one, we wouldn't be able to forward over both packet and circuit paths concurrently due to this limit of our switching hardware.

- In our Helios proposal, using the Fulcrum SDK we developed a customized software to control the Pod switches. For the most, the software exposes an API for the Topology Manager to control the forwarding tables of the Pods. This not only was not a pleasant experience but also was a very complicated

procedure with diverse software engineering challenges to address. Moreover, for wide adoption of Helios in an industry setting, this literally means that for every switch vendor such a customized software needs to be developed using the specific SDK for that particular hardware, which is completely infeasible. This alone completely argues for a more general approach such as building our prototype on top of OpenFlow [MAB+08] which is supported by many vendors already [Ope].

- The Topology Manager software which is the heart of Helios over time turned into a relatively large and complex software with many different components. As a result, it has became really hard to add and try out new scheduling algorithm. To get around this software engineering issue, we re-factor our Topology Manager to have a pluggable module which is in charge of decision making parts (more detail in Section 4.6).

3.4 Design requirements for hybrid networks

Having learned from the previous subsections the main challenges in designing/building hybrid networks, we now provide a set of reasonable requirements to build effective hybrid interconnects. Specifically, an optical circuit controller should be able to meet four key requirements:

- **Tolerate inaccurate demand information:** 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.

- **Tolerate ill-behaved applications:** As we showed in previous sections, bursty datacenter applications can cause unnecessary network reconfiguration. Non-malicious but selfish applications could claim to need more ca-

pacity than they really do. All such ill-behaved applications can reduce the performance of a hybrid network. The circuit controller must therefore be robust to their behavior, ensuring that the network's performance is not affected by them.

- **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 traffic that is on the critical path of an application vs. less important flows differently. It should also provide a way for the controller to gather sufficient information to understand the application's dependence upon a flow's performance.

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

3.5 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. As such, we propose a three phase approach for managing hybrid networks which we call the *Observe-Analyze-Act* framework.

To get a better understanding of the network dynamics and application heterogeneity in the cloud ecosystem, the hybrid scheduler should be able to interact

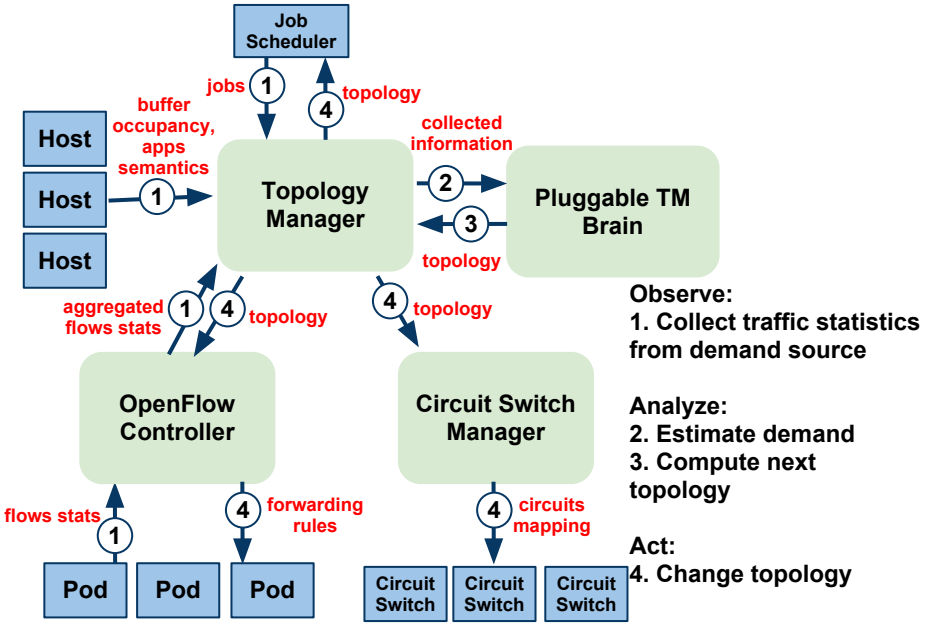


Figure 3.1: The control loop of the Observe-Analyze-Act framework

with different components and collect information from them. This information collected in this *Observe* phase would include the flow counters and 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 hybrid 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.

Figure 3.1 shows the control loop and the main software components involved. Our framework includes three main components: Topology Manager, OpenFlow Controller application and Circuit Switch Manager. The Topology Manager is the heart of the system, coordinating with other components to collect information for the Observe phase. To collect input from the network, we lever-

age the existing OpenFlow API using an OpenFlow Controller system (e.g., NOX, Ethane, etc). We argue that the Analyze phase logic should be implemented as a pluggable piece of software in the Topology Manager. We refer to this component as *Brain* because it has the algorithms and logic to make circuit scheduling decisions (e.g., "Pluggable TM Brain" in the figure). This allows us to separate policy from mechanism and evaluate different optimization goals and algorithms. Based on the output of analysis, during the Act phase, the Topology Manager provides the new topology to the Circuit Switch Manager to configure the optical links, to the OpenFlow Controller application, and to the application job scheduler. The OpenFlow Controller configures the switches to forward traffic over appropriate links. The job scheduler can potentially use this information to schedule jobs which can take advantage of the current network topology.

Our experience in using an OpenFlow Controller for this framework has been positive: it provides sufficiently fine-grained control over flow placement on the optical links, and it has allowed us to implement different hashing policies and flow management decisions that achieve better sharing among applications in the cloud. Our experience meshes well with prior work that used OpenFlow to create a unified control plane for IP/Ethernet and optical circuit-switched networks for long-haul backbone networks [DPS+10].

3.6 Contributions of this framework

The main contributions of the Observe-Analyze-Act framework is addressing the aforementioned challenges in the previous subsections. Specifically:

- This framework doesn't focus on certain scheduling algorithm. Instead it provides a general approach through which any scheduling algorithm can be plugged in to the Topology Manager and used for the purpose of circuit scheduling. So, effectively for trying out a new algorithm, just a new Brain needs to be implemented and plugged in and no other major modification to the rest of software is needed.

- Given that we use OpenFlow switch controller, this gives us the flexibility of

making decisions at the granularity of flows. So, the Topology Mananger can decide to put any flow over any specific circuit or packet switch path. This resolves the some of undesired issues such as treating all the flow the same or randomly hashing flows across the set of uplink circuits. The scheduler could decide to use packet switch path if the flow is bursty or delay-sensitive. If the flow is better suited for the circuit path, it could then decide which particular circuit to route the flow over.

- Given the proposed fine-grained controller, it is possible to provide certain guarantees. For instance, application level fairness or certain QoS policies can be supported based on any user-defied strategy or policy.

- There is no need to develop a customized software for the Pod switches by using the OpenFlow API for all sorts of communication with the Pod switches through an OpenFlow Controller system. As such, the Pod switch only needs to support OpenFlow API which is the case for many of the today's switches from different vendors.

Chapter 4

System architecture, design and implementation

This section describes different aspects of the software modules which build our prototype based on the Observe-Analyze-Act framework (Section 3.5). Our prototype has four independent yet cooperating softwares: Pod Switch Manager, OpenFlow Controller application (in particular, we use NOX), Glimmerglass Manager and the Topology Manager. Pod Switch Manager and OpenFlow Controller are in charge of managing Pod switch hardware and providing an abstraction for the Topology Manager to configure their forwarding behavior. In addition, Glimmerglass Manager software acts as a driver between the Glimmerglass circuit switch and Topology Manager to provide an interface for changing the topology of the Glimmerglass circuit switch. The Topology Manager software is the heart of our prototype which runs a certain control loop in which it coordinates with other components to change circuit switches topology dynamically. In the following we describe each of these softwares and their interactions in more details.

4.1 OpenFlow based Pod Switch Manager

For the Helios research effort we wrote a software to manage Pod switches. This was a C++ user-level process using Fulcrum's SDK to communicate with Moncaco's FM4224 switch ASIC and Apache Thrift library to communicate with

the Topology Manager. However, in our new prototype we decided to use a pre-release version of an OpenFlow based module which was provided to us by Fulcrum. This OpenFlow implementation runs as an application in the Embedded Linux based environment that runs by default on Monaco switches. To this end, we have re-architectured our Topology Manager from what is described in [FPR+10] to interact with a NOX application which in turn interacts with Pod switches through the OpenFlow API instead of a customized API. As a result of this now the functionalities that the Pod switch software provides is a subset of those provided by the OpenFlow abstraction. Specifically, the following functionalities are expected from the Pod switch software:

- Providing an OpenFlow implementation through which the Topology Manager can effectively configure the forwarding table of the switch at the granularity of each individual flow
- Maintaining and exposing flows statistics (e.g., byte counters) to the Topology Manager through corresponding OpenFlow function call

Switching to OpenFlow API makes our Pod switch hardware/software to be totally commodity as long as this API is implemented by the switch vendor. In fact, today many commercial switch vendors such as Arista Networks, Juniper Networks, Hewlett-Packard, and NEC [Ope] implement and expose OpenFlow API as a built-in component of their switch softwares. As a significant benefit of this, any OpenFlow based switch can be used in a plug-and-play fashion as the Pod switch in our prototype.

4.2 OpenFlow Controller application

We use NOX [GKP+08] system for managing our switches through a logically centralized potentially physically replicated controller. The controller application has a complete view of the topology. As such, when a packet reaches a controlled switch which holds no flow-entry for the description of that packet, it will be forwarded to NOX application to make decision for. The idea is to construct

the OpenFlow packet manually, filling out the flow description and the actions and then send it to the controlled switch to update its forwarding table.

We implement a NOX application which is run over one of our servers. The IP address of this server over the control network is passed to each of the Pod OpenFlow software as a command line argument to which they establish a control connection. In addition, the controller application has a connection to the Topology Manager through which it sends the flow statistics upon the request of the Topology Manager and receives the new topology of the network. Then, the controller updates the forwarding table of each of the Pod switches correspondingly through OpenFlow API. Moreover, the Topology Manager might configure a particular Pod to change the forwarding path for a certain flow.

Specifically, as in this controller application we need to have a connection to the Topology Manager, we decide to extend the available NOX messenger component as a way of sending commands from Topology Manager to the remotely running NOX for executing. The advantage of extending the messenger component is that the messenger component already has the functionality of listening on a certain port to receive commands for execution. Hence, we define a new message type , MSG_NOX_STR_CMD, for sending commands from the Topology Manager to the NOX messenger component. The command string that we currently support is one of the following:

- initParams: For initializing the controller with different parameters. Specifically, the number of Pod switches and number of circuit switches. These are needed for sanity check of number of Pods as well as figuring out the mapping for each of them.

- getOctetCounters: This is used for requesting the flow counters from the Pod switches which is used as a raw input for building the traffic matrix in the Helios system.

- getAllPortsOctetCounters: This command requests for the ports counters of each individual Pods. This statistic is exposed for the purpose of figuring out the TX/RX rate of each circuit. This is not only a valuable information to

collect but also potentially can be used in more complex circuit scheduling algorithms which would make decisions based on the circuits utilization over time.

- setUplinkMap: This is used for announcing the new configuration to each of the Pods. Upon receiving this command, the controller figures out the new topology for each of the Pods and configures each Pod with corresponding flow-level forwarding table.

For the purpose of serializing the data sent over the socket between the NOX controller C++ application and Topology Manager Python code base, we use Python struct module. This module interprets strings as packed binary data for performing conversions between Python values and C structs represented as Python strings.

The NOX component is event based. That is, upon execution of each event the corresponding registered handler for that event is triggered. Specifically, the main functionality of the controller application is consists of capturing the following set of events:

- Msg_event: This event is triggered whenever the messenger component receives a packet through its listen port. In this event handler besides the command types the messenger component already parses, we look for MSG_NOX_STR_CMD command sent from the Topology Manager and call another method which based on the command string dispatches the request to different functions.

- Datapath_join_event: This event handler is called upon join of a new switch to the network. We use this to initialize the switch forwarding table such that the switch uses the core packet switch path by default. We use a soft-state approach; thus, switches do not keep much state information. The only state which is maintained is the last destination of uplink circuit ports of the switch which is used to update the forwarding table only when the topology is different (which is an optimization for removing the unnecessary

reconfiguration of the same topology). So, this minimum state design makes it easy to replicate the NOX controller for fault-tolerance reasons.

For setting the initial forwarding rules in each Pod, we insert a set of static rules to each of the switches upon their connection to the NOX Controller as follows. Because we divide a single Monaco switch to two virtual Pods (Figure 2.1), the rules should be set such that packets across Pods on the same physical switch are forwarded over an upstream port to a core switch not the direct port to the destination host. As in our prototype the number of hosts is moderate (24), we decide to take a simple approach of inserting a separate rule for all combination of source and destination hosts in each of the Monaco switches. Then, on each Pod switch we insert: 1) a set of intra-Pod rules to forward the packets of the hosts on the same Pod among each other through forwarding over the output port of the destination host (6 * 6 = 36 rules on each Pod) and 2) a set of inter-Pod rules to forward packets to host on other Pods through taking the core packet switch path (6 * 18 = 108 rules on each Pod). An alternative approach is to define different VLAN ids for intra-Pod vs. inter-Pod traffic and label all the packets with appropriate VLAN id. Then the forwarding for each packet can be done based on its VLAN id over the link of the destination host (intra-Pod case) or the core switch uplink (inter-Pod case).

- Flow_stats_in: This is triggered upon receiving a flows statistics response from any of the switches. In particular, once the controller application receives a request from the Topology Manager to collect the flows statistics, it sends a request to each of the Pod switches for their statistics. Once each Pod switch replies to such a request, this event is triggered. The controller collects the statistics from all of the Pod switches and sends the aggregated statistics to the Topology Manager.

- Port_stats_in: This event is called upon receiving ports statistics from any of the Pod switches and is similar to the above event. The ports counters can be used to capture the circuit uplinks throughput and also in Brains for

deciding the next topology based on circuit utilization.

4.3 Core packet switches

The core packet switches in our prototype are traditional plug-and-play layer two/three switches which use switch learning process (based on layer two mac addresses) to build their forwarding table. As such, they don't require any customized software or specialized configuration to run with. However, in a large deployment it might be more efficient, if not necessary, to configure the switches forwarding table with the set of mac address of the end-hosts a priori.

4.4 Circuit Switch Manager

As in our prototype we just deal with a single physical circuit switch (i.e., the Glimmerglass switch), we choose to implement the Circuit Switch Manager as a library instead of a standalone software component. So, in our prototype the Circuit Switch Manager is a library implemented in Python which is exposed to the Topology Manager to configure the circuit switches to set up new circuits between specific ports or break up certain existing circuits. To this end, we wrote a library on top of the TL1 interface that our Glimmerglass circuit switch provides to interface with the switch. Specifically, as we only have a single circuit switch hardware, this library provides the notion of virtual circuit switches and makes this translation transparent to the Topology Manager. The library provides two major API function calls, one to make a connection from a Pod to another through a certain circuit switch and another to break an existing connection of a pod through a specific circuit switch. There are also function calls to do the same functionality for a group of circuits. This reduces the overhead of Remote Procedure Call (RPC) as the Glimmerglass has the ability of setting/breaking the circuits in groups. Moreover, the Glimmerglass provides two different RPCs: *synchronous* and *asynchronous*. In the synchronous mode the RPC doesn't return before the requested action is completed whereas in the case of asynchronous call the RPC returns as

soon as the request is received and afterwards the Glimmerglass acknowledges the completion of the requested action through a so called *notification events*. In our implementation we use the synchronous mode as its performance is good enough for our prototype meanwhile it is easier to deal with and also it makes the exposed API to the Topolgy Manager simpler to use.

4.5 Topology Manager

The Topology Manger is the heart of the Observe-Analyze-Act framework-which organizes the coordination among other components. The core of the Topology Manager is implemented in Python for ease of development, however it uses two libraries (i.e., Hedera demand estimator and the scheduler library) which are implemented in C for the sake of efficiency. For interfacing between C and Python we use SWIG [swi]. Although in our simple prototype the Topology Manager is implemented as a central component resulting in a potential single point of failure, in a large scale deployment, it can be replicated to overcome this problem. In fact, given the lack of strict consistency and state in the system, it is straightforward to deploy any replication techniques.

This software is completely configurable through a configuration file which is passed as the command line argument. We refer to this file by *superman config*. The configuration file declares (1) different physical/logical parameters of the system such as number of Pods and core circuit/packet switches, the IP/MAC address of different component of the system including hosts and OpenFlow Controller application, the physical port number of the uplinks as well as the hosts on the Monaco switches, etc (2) different configurable parameters of the logic/algorithms which are used such as Brain (more later), demand estimator type, scheduler type and its parameters, etc. In sum, Topology Manager software does the following tasks:

- It starts off by making connection to each source of demand matrix being c-Through end-hosts or Helios Pod switches. The implementation is done in such a way that only minor modification to the existing code is needed for

introducing a new source of demand or adding a new scheduling algorithm. We describe this further in Section 4.6.

- Establishes a control TCP connection to the OpenFlow Controller application to send command to it. This is different from our Helios prototype where the Topology Manager would connect to each of the Pod switches directly.
- Establishes a control connection to the Circuit switch to change its topology dynamically using the library described in Section 4.4.
- After doing the steps above the Topology Manager executes a control loop continuously which we call the *control loop* and is described in Section 4.8.

4.6 Pluggable Brain

In Observe-Analyze-Act framework we have refactored the Topology Manager from what implemented in [FPR+10] to make it much modular and easier to plug new algorithms in. In fact, the Topology Manager control loop doesn't have the algorithms to decide how to store the collected demand and how to decide a new topology; instead the so called Brain component is in charge of these and the Topology Manager interacts closely with this component through provided API. Specifically, each Brain must provide an implementation for two methods: One to store the collected demand information at a certain time and another to decide the circuit mapping for a specific time. The idea in taking a timestamp is that generally the Brain decisions could be time-dependent and could take into account information from past to make future decisions. For instance, as explained in Section 2.2 there is an $0 \leq \alpha \leq 1$ parameter in the superman config file which determines the EWMA damping factor for calculating the rate of flows based on past history. This is useful in some traffic patterns such as all-to-all to build a more accurate estimate of the demand. Moreover, using some history in particular for determining the circuit switches topology is promising in terms of improving the overall performance of the system. We have currently implemented the following Brains:

- Helios Brain: This Brain is mostly identical to our previous work [FPR+10]. The demand sources are the Pod switches which are reached through the OpenFlow Controller. The circuit scheduling is done using the Edmonds [Edm65] algorithm. The weight of the edge of the graph from Pod i to Pod j for the first circuit switch is set to either 10000 (rate of uplink circuit) or rate of demand from Pod i to j, whichever is smaller as explained in detail in Section 2.1.3.

- c-Through Brain: This Brain is similar to the [WAK+10] paper. The traffic demand is pulled from each individual server's TCP socket buffer occupancy. In our small size prototype the Topology Manager serially pulls each of the servers. As this didn't result in poor performance we decided to use this simple approach but in practice in a large scale deployment this will most likely be implemented in parallel across different servers. For the c-Through Brain we also use the Edmonds algorithm. The selection of the weight of the edge in this case is tricky as the demand estimation is done through the socket buffer occupancy which is in number of bytes. So, there is no notation of time in the estimated demand. We believe the well-known *delay-bandwidth-product* metric to some extend is capable of capturing the amount of bytes which can be forwarded over each circuit for the duration of each control loop cycle. For this we use the average duration of the cycle as the delay and 10000 (bandwidth of circuits) as the bandwidth. We again consider the minimum of this value and the estimated demand as the weight of the edge. We decide to not explore the underlying scheduling problem for c-Through more as this approach performs well enough in our prototype but there is potentially interesting future theory work in this space.

4.7 Traffic Analyzer

The Traffic Analyzer provides a set of simple yet quite useful processing tasks on a given set of measured flows. It is implemented as a Python library to be used by different Brains. So, literally this component is an optional part of the

Algorithm 4.1 updateCounters(Flow f)

```
1: if f.byteDelta == 0 then
2:     f.activeCounter = 0
3:     f.idleCounter += 1
4: else
5:     f.idleCounter = 0
6:     f.activeCounter += 1
7: end if
```

Algorithm 4.2 isBursty(Flow f)

```
1: if f.activeCounter ≥ 5 then
2:     return False
3: else if f.idleCounter ≥ 3 then
4:     reset(f)
5: end if
6: return True
```

Algorithm 4.3 reset(Flow f)

```
1: f.activeCounter = 0
2: f.idleCouner = 0
```

Brain. The functionalities that this library provides are as follows:

- Given the list of observed flows, filtering out the bursty ones and returning the non-bursty flows. This is done through considering an idle and an active counter for each flow to represent the number of cycles the flow has been active and idle, respectively. Over the cycles the flow is sending data (the TX byte counter for the flow is positive) the active counter is incremented and the idle counter is reset to zero. Similarly, if the flow is not sending any data over a cycle the idle counter is incremented and the active counter is reset to zero (Algorithm 4.1). Then, based on our bursty flow detection algorithm, a flow is non-bursty if its active counter is greater than a certain constant number (Algorithm 4.2). If a flow has been idle for more than specific number of control cycles, we reset both of the counters for the flow to zero (Algorithm 4.3) in order to account for changing nature of flow; e.g., when a previously bursty flow becomes stable or vise verse.

- Given the list of observed flows, removing out any traffic which should not be considered in determining the next topology. This includes the UDP background traffic that we use to slow down the packet switch network (see Section 2.1.1).

4.8 Control loop

The control loop of our current prototype is a subset of the Observe-Analyze-Act framework described in Section 3.5. Specifically, the Topology Manager interacts closely with the Brain to decide the best topology and then configures the Pod and circuit switches with that topology. To this end, Topology Manager collects the required information and pass them to the Brain and then the Brain decides the next topology. After this the Topology Manager performs the required configuration actions for setting up the new topology in place. So, in a sense Topology Manager is the worker for the Brain which has the logic/algorithm to decide the right topology at each time. This is done periodically in the Topology

Manager control loop which is summarized in Fig 4.1. Note that in our current prototype we don't have algorithms which use job scheduler's data or end-hosts application's semantics for deciding the next topology. This explains the difference between this figure and Fig 3.1 which is the more general approach we propose in Observe-Analyze-Act framework. We also identify the Traffic Analyzer component as a part of the Brain. In sum, the control loop of our prototype consists of the following actions:

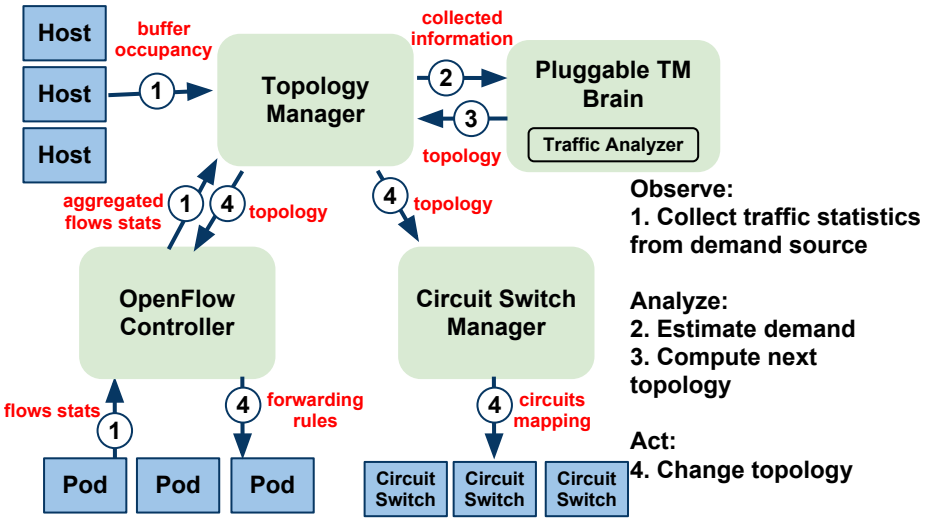


Figure 4.1: The control loop of our prototype

- Collecting the demand matrix from the Brain's specified demand source. The demand source is specified in the superman config. The currently implemented source of demand can be Pod switches for the Helios Brain and end-hosts for the case of c-Through Brain.

- Passing the collected demand information with corresponding timestamp to the Brain to estimate the traffic matrix. It is then up to the Brain how to process and store this information (if it wants to use it in future at all).

- Inquiring the Brain for the next topology passing it the current timestamp. As explained earlier, the idea behind using the timestamp is for the case Brain uses some sorts of history and has a time-dependent algorithm for choosing the topology.

- If the topology needs to be changed, the Topology Manager sends a command to the OpenFlow Controller informing it of the new topology. The OpenFlow Controller then configures the Pod switches forwarding table through OpenFlow API. The Topology Manager also configures the circuit switches with the new topology.

Chapter 5

Initial evaluation of OpenFlow based framework and Traffic Analyzer component

This chapter presents the initial evaluation of the OpenFlow based framework we proposed in Chapter 3. In particular, we show its feasibility and superiority compared to our previous work, i.e., Helios. Although we don't yet have a concrete answer to some of the challenges in building hybrid networks, for instance, the best circuit scheduling algorithms and how to detect dependent flows in mixed workloads, but we show that this framework is promising for solving the problems we described in Chapter 3. In fact, we believe that the modularity and fine-granularity control of this framework makes it easier to propose and try out future algorithms just by implementing new Brains.

5.1 OpenFlow implementation

The OpenFlow Controller approach provides the Topology Manager with a fine-grained control over the path of each individual flow in the network. This is specially useful to get around the problems we illustrated in Section 2.5 due to randomness of the hashing approach. To this end, when there are multiple uplinks for a particular Pod to another, using the proposed framework the Topology Manager

can determine which circuit each flow gets forwarded over as opposed to relying on a random hash function to make such an important decision. In addition, this fine-grained control makes it possible to provide certain performance guarantees at the application level. For instance, a scheduler could provide application-based fairness. We construct the following simple case study experiment to illustrates these benefits.

We run two applications: application A with 18 TCP flows and application B with only 2 TCP flows. These applications are being run across two Pods which are connected through two circuits to each other. In this scenario, we observe that over the Helios network (or c-Through), application A only gets $\frac{18}{20}$ of total 20Gbps bandwidth which is around 18Gbps and application B gets the rest of available bandwidth which is around $\frac{2}{20}$ = 2Gbps. This is because flows are uniformly hashed over the uplinks. Thus, this approach can provide only flow level fairness as in this case all the flows are treated the same each getting 1Gbps as its share of bandwidth. However, this flow level fairness is not always desirable and depending to the importance and type of the applications the network administrator might want to allocate the same amount of bandwidth to each of application A and B. We next provide this application-based guarantee by using the OpenFlow based approach. To this end, we configure the Pods such that application A's flows are routed over one of the two circuits and application B's flows on the other circuit (we insert an OpenFlow forwarding rule for each flow). By doing so, we observe that both of the applications can utilize corresponding circuit link and receive 10Gbps bandwidth.

To visualize this application-based fairness issue we use Jain's fairness index [JCH84]. Figure 5.1 shows this fairness metric for this scenario for Helios network vs. OpenFlow based approach. As it is seen in the graph, the fairness is significantly increased from 0.6 in Helios to 1 (which is the maximum) for OpenFlow based approach.

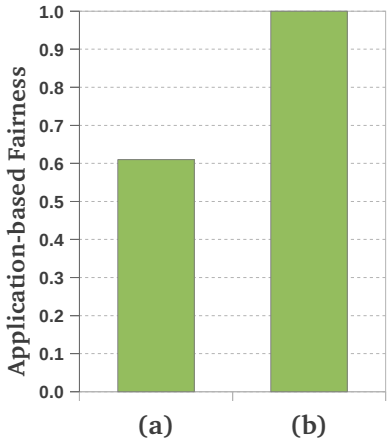


Figure 5.1: Application-based fairness in (a) Helios/c-Through vs. (b) the OpenFlow based approach

5.2 Traffic Analyzer component evaluation

Demand estimation is a major challenge in Helios and c-Through systems. In facts, as we discussed in the previous sections, even a single bursty flow may disturb these systems to the extend where they per from very poorly. We here run a case study to show the effectiveness of our Traffic Analyzer component which detects and removes bursty flows from the measured demand to avoid the unnecessary circuit flapping we observe in Section 2.4 due to the background bursty flow. We however note that we are not yet aware of any approach or traffic engineering work to completely detect the flows dependency at the granularity of few milliseconds in a datacenter workload. As such our focus here is on removing the bursty traffic and detecting the correlation between flows is a remaining future work.

We perform a similar experiment as that of Section 2.4 albeit with the Traffic Analyzer component being activated. In particular, the workload consists of a long lasting flow and a bursty flow. These two flows are sourced from different Pods but send data to the same destination Pod. There is only a single circuit links for which these two flows are competing against each other. We discussed in Figure 2.11(b) that in this case each time the bursty flow is sending data it takes over the circuit and the system ends up oscillating between two different topology. Thus, the overall performance is significantly poor. On the other hand,

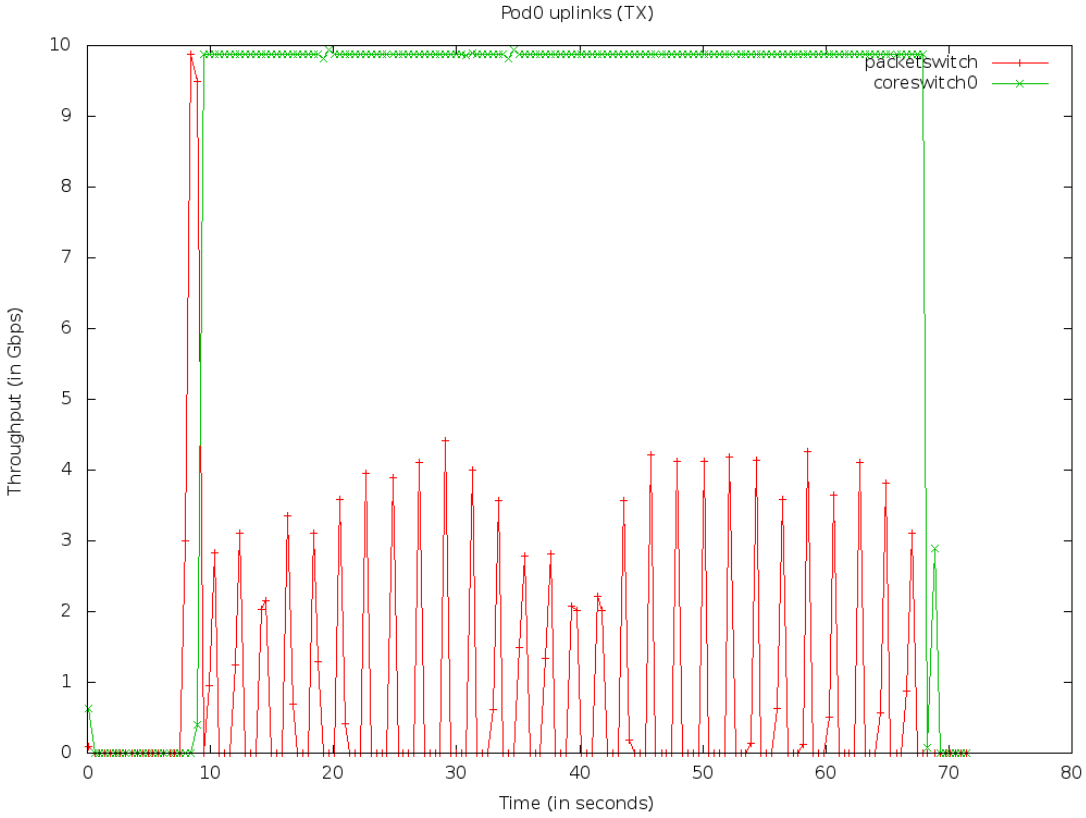


Figure 5.2: The throughput of the packet and circuit links for a bursty traffic of 100ms when the Traffic Analyzer component is used. The bursty flow is detected and not used in circuit scheduling decisions and hence no circuit flapping event is observed.

Figure 5.2 shows the throughput of the links for burst duration of 100ms when the Traffic Analyzer component is being used. There is no circuit flapping as the bursty flow is detected and removed by the Traffic Analyzer and hence is not being considered for circuit scheduling decisions. This shows that the simple Traffic Analyzer component we proposed is quite effective in handling unnecessary circuit reconfiguration events due to bursty flows.

As we reduce the duration of the bursty flow, we observe the same trend. However, once the bursty duration gets larger some circuit flapping is observed because the flow passes through the Traffic Analyzer filter and is not considered bursty any more. This introduces a trade-off between responsiveness of the system vs. tolerance of the system against the bursty flows. However, still overall the number of circuit flapping events is less when using the Traffic Analyzer component.

For instance, for the burst duration of 300ms, there is only 2 circuit flapping events whereas without using this component there is 54 of such event observed.

Chapter 6

Related work

We divide the related work into different categories: recent datacenter networks architectures, using optical networks in datacenters and centralized control mechanisms for enterprise and datacenter network.

There are quite a few SIGCOMM papers from the past couple years proposing new datacenter architectures with different characteristics such as full bisection bandwidth, low cost and power efficient [WLL$^+$, AFLV, GHJ$^+$, GWT$^+$, GLL$^+$]. Among these is [AFLV] which suggests building a scalable fat-tree based hierarchy of commodity Ethernet switches for provision of full bisection bandwidth though with lower cost than the traditional multi-level hierarchies which is being used in building today's datacenters. VL2 [GHJ$^+$] extends the fat-tree structure by using 10Gbps Ethernet to form its switching backbone.

DCell [GWT$^+$] is another proposal which uses the servers each with multiple network interface for forwarding the packets across the network. DCell has the the following properties: (1) there is no single-point-of-failure and hence the overall architecture is very resilient even in the presence of severe link or node failures. (2) it is exponentially scalable as the degree of the nodes increase. (3) it provides higher overall network capacity than existing multi-level architecture. BCube [GLL$^+$] is another recent work done by the same group which is built on top of DCell. The main idea is to also incorporate switches for faster processing and active probing for load-balancing. Another more closely related work to Helios and c-Through which also is a follow up on BCube and DCell papers is MDCube [WLL$^+$]. The objective

in this work is to interconnect modular shipping-container based datacenters. Their main idea is to use the BCube containers as the building blocks and also put the routing and fault-tolerance functionalities inside the end-hosts.

Recent works propose promising approaches for using optical networks for designing datacenter interconnects [FPR+10, WAK+10]. We covered these hybrid electrical/optical architectures in detail in Section 1. However, the idea of using circuit switching and packet switching together as the building block of the Internet network has been around at least for a decade. For instance, the idea in [NML98] is to use ATM hardware for forwarding long lived flows over circuit paths and traditional packet switches for forwarding the short flows over the IP network. They implement IP directly on top of ATM hardware to preserve the connectionless model of IP and not incurring overhead of circuit establishment. So, the gain is to use ATM forwarding technique (which was faster than IP packet-switching at the time) to improve the performance. Helios is different from this work in that first it is focused on the datacenter area and more importantly today the speed of forwarding in IP vs. circuit based techniques is comparable. As such the main incentive in using optical circuit switching is to reduce the cost and power.

[BBH+05] is another closely related work with similar properties of cheaper and more power-efficient design. The main differences between this work and Helios/c-Through proposals are that first it is in the space of high performance computing and that they look at the traffic stability at the level of individual end-hosts whereas later deal with aggregated Pod-level traffic stability. Another closely related work is [KPB]. The similarity comes from the fact that both proposals consider provision of full-bisection bandwidth a needless overkill given realistic datacenter traffic patterns. The difference between the works come from the approach to handle the almost stable yet with random hotspots traffic pattern observed in datacenters. Flyways main proposal is to use wireless links to augment bandwidth where and when needed whereas Helios and c-Through use optical circuit switches.

Another line of related works is recent proposals to control datacenter networks with a software controller with global view of the topology [GHM+05, CCS05, CGA+06, MFP+07, GKP+08, Ng10]. The idea is to provide a high level

logically centralized (potentially physically replicated) software component to observe and control a network. The main advantages of this is flexibility as the network programs/protocols would be written as if the entire network were present on a single decision element as opposed to requiring a new distributed algorithm across all network switching elements. Also that programs may be written in terms of high-level abstractions such as users, host names, not low level configuration parameters (e.g., IP and MAC addresses). We briefly overview some of these systems in the following.

Among the initial such systems was 4D [GHM+05]. The objective in that system is to control the forwarding decisions of the networking elements. Hence, the network view only includes network infrastructure. Later, the SANE [CGA+06] and Ethane [MFP+07] proposals moved beyond this by adding a namespace for users and nodes in the network view. In addition to this they provisioned a finer granularity forwarding control compared to 4D (e.g., per-flow vs. forwarding table based). Maestro [Ng10] is another OpenFlow Controller system which objective is to avoid making the controller the bottleneck through exploiting parallelism in every possible way.

We chose to build our system on top of current state of the art OpenFlow Controller called NOX [GKP+08] which enhanced the general programmatic control of the network. NOX is an operating system for network which leverages OpenFlow abstraction as the communication protocol with the networking devices. NOX software runs on one or some of the servers in the network which largely is several different controller processes and a single network view. NOX applications use this network view to make management decisions. Specifically, they may manipulate path of network traffic at granularity of flow level. This is done through registering different event handlers which are triggered upon execution of certain events such as switch join, switch leave, packet received, flow initiations, etc. Section 4.2 is dedicated to detail of our NOX application.

Chapter 7

Conclusion and future work

Hybrid electrical/optical networks promise significant reduction in cost, deployment complexity and energy consumption of large scale datacenter interconnects. In this thesis we summarize our experiences in constructing such hybrid networks over the past two years through building a functional moderate size prototype. To this end, we first propose Helios as a simple yet efficient starting point which maximizes the bisection bandwidth over the circuit switches. Then to move beyond this we evaluate this system and another similar proposal, c-Through, under heterogeneous workload and encounter a number of challenges, some quite unexpected. For the most, we often observe circuit scheduling decisions that lead to unnecessary circuit flapping and significantly low circuit utilization which is due to treating flows as interchangeable and undistinguished from each other as done in Helios/c-Through. We then articulate a set of assumptions and challenges which produce these undesired situations.

Based on these, we then enumerate the design requirements of hybrid interconnects and a meta solution to these challenges under an Observe-Analyze-Act framework. The framework collects traffic statistics and network status information from various source of information such as switches, hosts, job scheduler (such as Hadoop Job Tracker node). It then analyzes the collected information to estimate traffic demand and decide the next topology for the circuit switches. Then, the framework configures this topology into different elements in the system including Pod and circuit switches.

Although we don't yet have a final answer to some of the challenges for the most the "best" scheduling algorithm for hybrid networks but we believe this framework is promising with a fine-grained control to implement and evaluate any desired algorithm. In short, the contributions of this thesis are the following:

- A fully functional prototype of Helios/c-Through proposals
- A comprehensive evaluation of Helios/c-Through under different workloads and against each other
- A taxonomy of non-trivial challenges which arise under heterogeneous workloads and the underlying assumptions that Helios/c-Through make which lead to undesired circuit flapping events
- A high-level outline of the features of a feasible solution space for these issues
- A general Observe-Analyze-Act framework and its initial evaluation

7.1 Future work

In this thesis, the focus has been on summarizing our experience in dealing with hybrid networks, the challenges we faced and proposing a general framework which the hybrid datacenter network planner can use for managing her network. The idea is that she can plug any desired circuit scheduling algorithm in. The current scheduling algorithm we use is Edmonds weighted matching with objective of maximizing the bisection bandwidth. Exploring this space from a more theory-based prospective and in particular proposing history/state based scheduling algorithms which take into account priorities as well as sharing policies is necessary and very promising in making better scheduling decisions and provision of different QoS policies inside and across applications. The circuit scheduling algorithm should be able to make reasonable decisions for the topology of the network and the placement of each individual flow over the packet or circuit switches under an any arbitrary workload.

Moreover, there is certainly opportunities to study some of the challenges in this work from a traffic-engineering perspective. In particular, identifying flow correlations and dependencies across a dynamically changing set of flows is a very critical future work to avoid some of the undesired situations. The challenge here is to do so under a constant number of control loop cycles. Because exploring all possible combinations of scheduling the flows for such a huge dynamic set in linear time complexity does not seem promising given the dynamic nature of these systems at aggressive timescales of milliseconds. A completely different approach to pursue is devising a cluster-wide abstraction through which applications communicate with Topology Manager and provide such information. However, this has its own drawbacks such as the need to modify applications. Another aspect which was covered in this thesis is a simple heuristic for identifying bursty flows, extending this effort and detecting any ill-behaved flows such as those which disturb the overall network performance is also an interesting follow up.

7.2 Acknowledgment

This thesis is partially based on the Helios paper [FPR+10] published in SIGCOMM'10 of which I was a coauthor. Specifically, among my contributions to that effort was the Topology Manager software which is described in detail in this thesis. There are also material from the paper titled "Switching the Optical Divide: Fundamental Challenges for Hybrid Electrical/Optical Datacenter Networks" submitted to ACM SOCC'11 which I was an author of.

Appendix A

Supplementary material

This Chapter is dedicated to some additional effort which are supplementary work done along with the thesis.

A.1 Hadoop experiments results

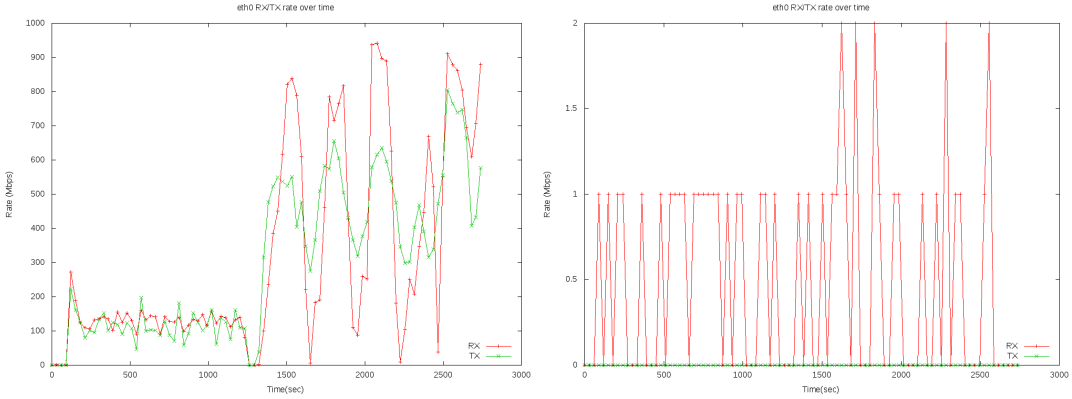
We use Hadoop as a typical real datacenter application to study its workload and traffic pattern. In this section, we present a short summary of some of the Hadoop results.

The experiment scenario that we focus on is for the case there is a single master node (Hadoop NameNode and JobTracker) and 38 slave nodes (Hadoop DataNode and TaskTracker). All the nodes are connected to each other through the same physical switch with 10Gbps network interfaces. The configuration files for each of the nodes are generated with the Hadoop tools we have developed which is explained in Section A.2.1. In particular, 7 disks on each node were used for data storage, one was used to collect logs from Hadoop itself, and 8 disks were used as intermediate disks by the map/reduce layer. No particular parameter tuning was performed on Hadoop itself. We first run a Hadoop "Randomwriter" job to generate the data for sorting in which each node generates 100GB of data into Hadoop Distributed File System (HDFS). Then we run a Hadoop "Sorter" job to sort all this data, 3800GB in total. We vary the number of simultaneous map/reduce tasks per node and provide the Sorter job results in the following.

```
public static String reverseDns(InetAddress hostIp, String ns)
  throws NamingException {
  //
  // Builds the reverse IP lookup form
  // This is formed by reversing the IP numbers and appending in-addr.arpa
  //
  String[] parts = hostIp.getHostAddress().split("\\.");

  // Modified for the SEED cluster
  if(parts[0].equals("192") && parts[1].equals("168"))
  {
      return new String("dcswitch" + parts[3] + "-eth" + parts[2]);
  }

  String reverseIP = parts[3] + "." + parts[2] + "." + parts[1] + "."
    + parts[0] + ".in-addr.arpa";

  DirContext ictx = new InitialDirContext();
  Attributes attribute =
    ictx.getAttributes("dns://"               // Use "dns:///" if the default
                       + ((ns == null) ? "" : ns) +
                       // nameserver is to be used
                       "/" + reverseIP, new String[] { "PTR" });
  ictx.close();

  return attribute.get("PTR").get().toString();
}
```

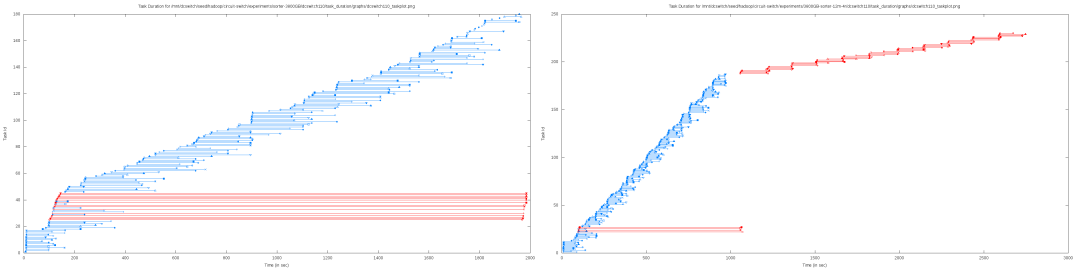
Figure A.1: The Hadoop 0.20.2 org.apache.hadoop.net.DNS class source code where the Reverse-DNS deamon reports eth0 address to the JobTracker as oppsed to the 10Gbps interface. The if block is the ad-hoc fix we devised for solving this bug for the SEED cluster.

Before going over the results of these experiments, it is interesting to point to one of the initial problems we faced when running Hadoop version 0.20.2, the most recent version at the time of running these experiment over the SEED cluster. The issue was that although we had configured Hadoop to run over eth5 (10Gbps) network, we realized it is still sending part of the map/reduce traffic over eth0 (1Gbps) network. This was due to a bug in the Hadoop source code in which the map/reduce daemons were hard-coded to listen on eth0. Furthermore, the reverse-DNS daemons were communicating the address of eth0 back to the JobTracker node. We did an ad-hoc fix to get around this issue and have the daemons report back the eth5 IPs instead. Figure A.1 shows our fix for this problem. Also, Figure A.2 shows the traffic over eth0 before fixing the bug and once the bug is



(a) (b)

Figure A.2: The traffic over the eth0 NIC of one of the Hadoop slave nodes (a) before fixing the bug (b) after fixing the bug

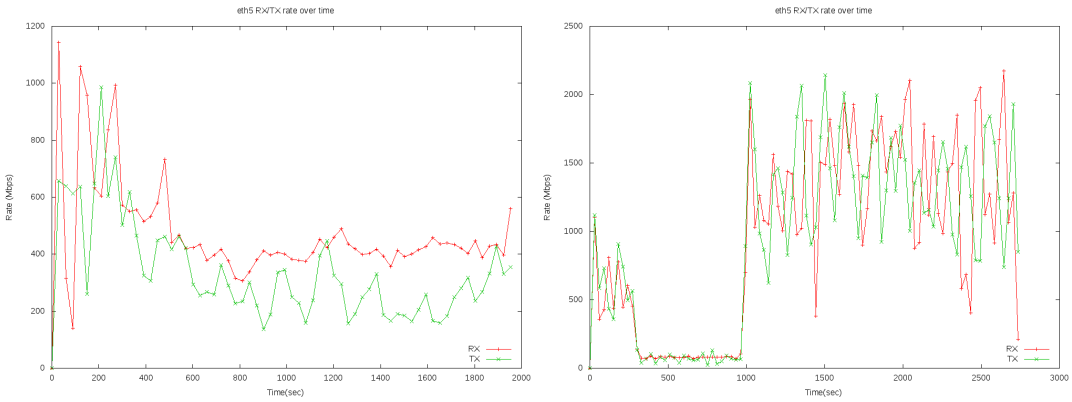


(a) (b)

Figure A.3: The duration of the Hadoop map/reduce jobs over time for one of the slave nodes (a) for 16 maps/16 reduces (b) for 12 maps/4 reduces

fixed which confirms the issue is indeed resolved after applying or patch to the Hadoop source code.

We also process the Hadoop log files to capture different statistics to study how Hadoop performs under a full-bisection network. Figure A.3 shows the duration of Hadoop map/reduce tasks. The X-axes is the time and in the Y-axes we have the tasks. The blue horizontal bars correspond to map tasks whereas the red ones correspond to the reduce tasks. Each bar indicates when corresponding task is started and ended. We realize that overall it takes longer for the Hadoop Sort job to finish when the number of reduce tasks are fewer. This is because no reduce tasks can get completed before all maps are finished. Also as can be observed in Figure A.4, Hadoop can not drive the network more than 2Gbps per node at



(a) (b)

Figure A.4: The traffic over eth5 interface during the period of Sort job for one of the slave nodes (a) for 16 maps/16 reduces (b) for 12 maps/4 reduces

maximum.

The motivation for running these experiments has been to get a sense on the upper bound on the improvement we could potentially achieve in the Hadoop performance through optimizing its shuffle phase. Specifically, through having the Hadoop JobTracker coordinate with Topology Manager software for setting up the circuits across the Pods based on Hadoop jobs duration. Unfortunately, the Hadoop logs didn't contain enough information to construct a full traffic matrix, because it only logs the destination of the shuffle operations, not the source-destination pair but its all-to-all communication pattern over the entire experiment, doesn't leave much interest for circuit scheduling related studies. Moreover, the low rate of Hadoop over each node over our specific setting with limited number of nodes, made it even less interesting to be used for the sake of this project because overall traffic was not big enough to drive all the circuits. As such we decided to use TritonSort instead moving forward.

A.2 Auxiliary tools

This section documents some additional tools we developed throughout the duration of this thesis for the purpose of running Hadoop experiments and also a tool based on TCPProbe to capture different parameters of TCP connections.

A.2.1 Hadoop tools

For running Hadoop [SKRC07] over a cluster of around 100 servers (similar to the experiments settings in Section A.1), certain settings and configuration files needed on each of the servers. However, configuring each of the servers and generating these required files manually on a server-basis is not only a complete hassle but also an error-prone process. Specifically, each server might have different number of hard disks mounted under its own label. As a result various parameters in the Hadoop configuration files are dependent to the server hardware/software configuration and should be set differently across the servers.

To get around these challenges, we have developed a set of tools which automatically initialize the environment for running Hadoop over set of servers in a certain project in the SEED cluster. Specifically, the tools initialize the environment for running Hadoop by (1) configuring given network interface of servers with a specific network address which is then used in the Hadoop configuration files. Hadoop uses these addresses for the sake of different communication, (2) configuring disks on the servers as the storage space for Hadoop and (3) generating Hadoop configuration files in the right place for each individual server. Moreover, there are two other tools to easily start and stop a certain Hadoop job over the cluster.

For more detail you may check the document available at https://sites.google.com/site/ucsdseedproject/documentation/hadoop-support. Figure A.5 shows usage of the tools for running a Hadoop Sort job. Note that by using these tools running a Sort job is as easy as issuing just a few bash commands.

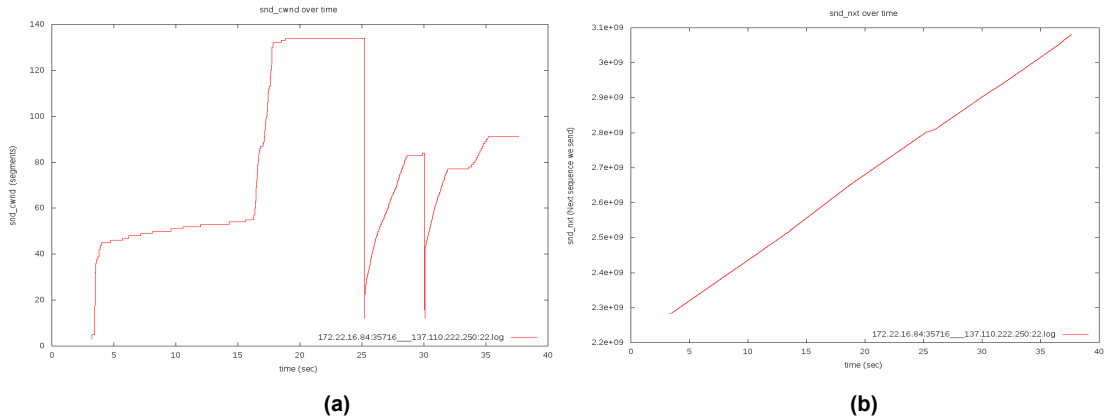


Figure A.6: Sample graphs generated with the TCPProbe tools. (a) The congestion window of the TCP connection over time in segments (b) The transmitted sequence number of the segments over time.

```
init_hadoop.sh <project> <interface> <hadoop conf dir>

start_hadoop_remote.sh <hadoop conf dir>

cd <hadoop dir>

bin/hadoop --config <hadoop conf dir> jar hadoop-*-examples.jar randomwriter
-Dtest.randomwrite.bytes_per_map=1073741824 -Dtest.randomwriter.maps_per_host=50
rand

bin/hadoop --config <hadoop conf dir> jar hadoop-*-examples.jar sort rand rand-sort

stop_hadoop_remote.sh <hadoop conf dir>
```

Figure A.5: Hadoop tools

A.2.2 TCPProbe tools

This section is an optional reading dedicated to an overview on some tools we built for the SEED cluster to enhance the debugging capabilities when running into issues during diverse experiments.

The flow-level trace of what's happening in a distributed networking system can be quite beneficial in troubleshooting performance and other problems. Specifically, being able to observe different parameters of each TCP connection such as throughput, congestion window size or sent sequence number diagram over time, number of timeout, etc could be a great source of information. TCPProbe

kernel module [TPr] is one of the tools which can be used to obtain such information. This module is automatically installed when the kprobe is enabled during the kernel compile process. After inserting the kernel module, it creates the file /proc/net/tcpprobe in which it logs a line for each incoming TCP packet. The log line includes a timestamp and a set of parameters related to that TCP connection. It is also possible to modify the kernel source file linux/net/ipv4/tcp_probe.c and recompile the kernel module to change the set of captured parameters. The way TCPProbe works is that it inserts a hook in the method tcp_rcv_established() which is called whenever a TCP packet is received.

We have developed a set of scripts to use TCPProbe over the SEED testbed. The SEED testbed works around the idea of supporting different *projects*. So, the way the tools are designed to work is to specify a project name (+tag) then (1) have the TCPProbe started on all the specified nodes of the project, (2) capture the logs and stop once the experiment is over (3) and finally generate the graphs for each TCP connection over these nodes. There are three scripts each of which does one of these actions. For more detail you may check the documentation page available at
https://sites.google.com/site/ucsdseedproject/documentation/tcp-probe-support.
Figure A.7 shows how to use these tools.

```
pssh_start_tcpprobe_logging.sh <project> <log dir> [TCPProbe args]
pssh_stop_tcpprobe_logging.sh <project> <log dir> <output dir>
generate_tcpprobe_graphs.sh <out dir> [param config file]
```

Figure A.7: TCPProbe tools

Figure A.6 shows two graphs which are generated with these tools for a Reno TCP connection which corresponds to uploading a file through scp command (sender side). The figure (a) is the TCP congestion window size (snd_cwnd) in TCP segments over time and the graph in (b) shows the sequence number of the segments sent over time for this connection. Below is a complete list of the TCP parameters that the tool generates by default:

- ssthred: This is the slow start threshold value as used in some TCP conges-

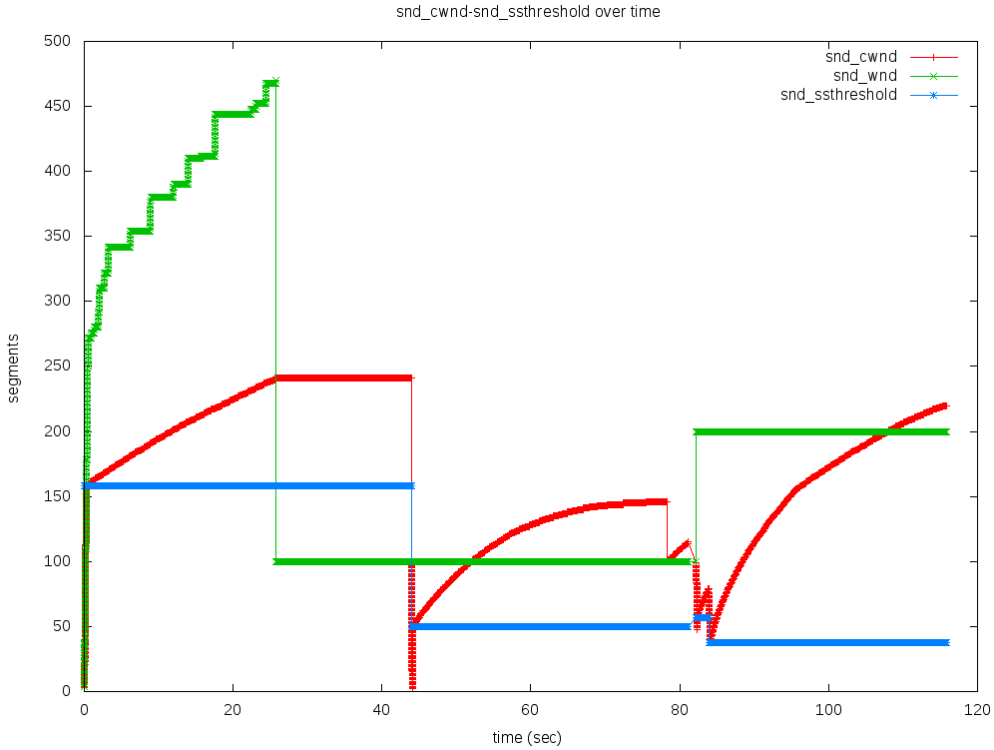


Figure A.8: The overlaid graph of snd_cwnd, snd_wnd, and ssthred all captured in segments for a TCP connection

tion control algorithms such as Reno to terminate the slow start phase and move to the congestion avoidance phase.

- snd_wnd: This is the receiver's advertised window and is reported in bytes instead of segments. However, one might modify tcpprobe_sprint() function in the TCPProbe source code to divide this variable by mss field of the logged packet to get this value in number of segments.

- snd_cwnd: This is the congestion window size in segments. It is important to realize that the rate at which the TCP sends is the minimum of this parameter and snd_wnd in segments. Figure A.8 overlays this parameter with snd_wnd (captured in segments with the hack explained above) as well as ssthred. Figure A.9(a) shows the number of in-flight packets corresponding to this TCP connection over time (this is packets_out parameter in tcp_log kernel structure used in the TCPProbe). Then, figure A.9(b) shows the

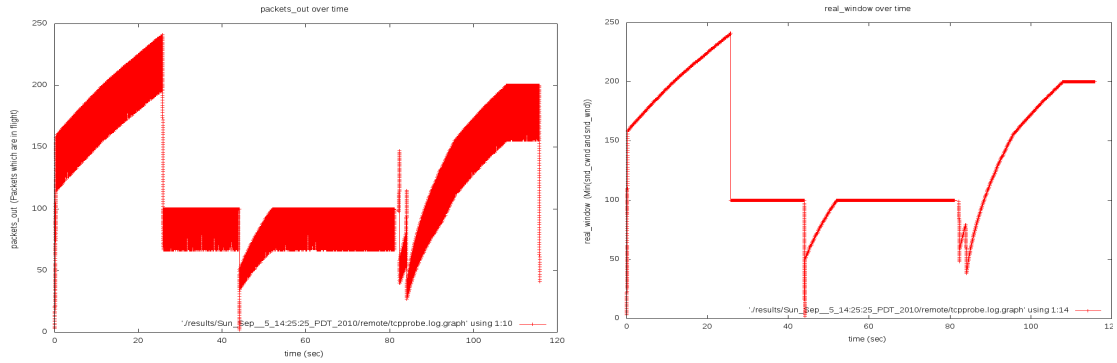


Figure A.9: The number of in-flight packets of a TCP connection as captured by (a) packets_out (b) by the real transmission window of the TCP which is min(snd_cwnd, snd_wnd) as shown in figure A.8. This Figure shows how these values match up.

real window size of this connection by plotting min(snd_wnd, snd_cwnd) over time. As seen in this figure the number of in-flight packets matches up closely with the real window size as calculated this way.

- length: This is the length of the received packet in bytes. Note that if you run TCPProbe on the receiver side of a server-client type TCP connection, most likely the length would be as small as 20 bytes which is the size of TCP header without any data.

- snd_nxt: The sequence number of the next segment to be transmitted.

- snd_una: The first unacknowledged sequence number.

- srtt: This is the smoothed RTT value corresponding to this connection as calculated by the RTT estimation algorithm of the TCP [Pax00].

A more detailed discussion on the Linux implementation of different TCP congestion control algorithm and other related issues is certainly out of the scope of this thesis (look [SK02] for a good reference).

Bibliography

[AFLV] Mohammad Al-Fares, Alex Loukissas, and Amin Vahdat. A Scalable, Commodity, Data Center Network Architecture. In *ACM SIGCOMM '08*.

[AFRR+] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI '10*.

[BBH+05] Kevin J. Barker, Alan Benner, Ray Hoare, Adolfy Hoisie, Alex K. Jones, Darren J. Kerbyson, Dan Li, Rami Melhem, Ram Rajamony, Eugen Schenfeld, Shuyi Shao, Craig Stunkel, and Peter A. Walker. On the Feasibility of Optical Circuit Switching for High Performance Computing Systems. In *Proceedings of SC*, November 2005.

[bro10] Broadcom bcm56480 10gbe series. http://www.broadcom.com/products/features/BCM56840.php, 2010.

[CCS05] Matthew Caesar, Donald Caldwell, and Aman Shaikh. Design and implementation of a routing control platform. In *ACM/USENIX NSDI*, 2005.

[CGA+06] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedmani, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proceedings of the 2006 Usenix Security Symposium*, 2006.

[cis07] Cisco data center infrastructure 2.5 design guide. http://www.cisco.com/univercd/cc/td/doc/solution/dcidg21.pdf, 2007.

[DPS+10] Saurav Das, Guru Parulkar, Preeti Singh, Daniel Getachew, Lyndon Ong, and Nick Mckeown. Packet and circuit network convergence with openflow. In *Optical Fiber Conference (OFC/NFOEC'10)*, March 2010.

[Edd01] Tze-Wei Yeow Eddie. Mems optical switches, 2001.

[edm] Edmonds's Algorithm Library. http://elib.zib.de/pub/Packages/mathprog/matching/weighted/.

[Edm65] J. Edmonds. Paths, trees and flowers. *Canadian Journal on Mathematics*, pages 449–467, 1965.

[ewm] Exponentially weighted moving average. http://en.wikipedia.org/wiki/Moving_average.

[FPR+10] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. HELIOS: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proceedings of ACM SIGCOMM '10*, New Delhi, India, August 2010. ACM.

[FRV09] Nathan Farrington, Erik Rubow, and Amin Vahdat. Data Center Switch Architecture in the Age of Merchant Silicon. In *Proceedings of IEEE Hot Interconnects '09*, August 2009.

[GHJ+] 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 *ACM SIGCOMM '09*.

[GHM+05] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin. Zhan, and Hui Zhang. Clean Slate 4D Approach to Network Control and Management. In *Proceeding of ACM SIGCOMM Computer Communication Review*. ACM, 2005.

[GKP+08] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martn Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. In *Proceedings of ACM SIGCOMM '08 Computer Communication Review*. ACM, July 2008.

[gli] Glimmerglass MEMS Optical Switches Switching time. http://www.glimmerglass.com/company/.

[GLL+] 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 *ACM SIGCOMM '09*.

[GWT+] 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 *ACM SIGCOMM '08*.

[JCH84] R. Jain, D.M. Chiu, and W.R. Hawe. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system.* Eastern Research Laboratory, Digital Equipment Corp., 1984.

[KPB] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways To De-Congest Data Center Networks. In *ACM Hotnets '09.*

[KSG+09] Srikanth K, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *Proceedings of Internet Measurement Conference'09*, 2009.

[LAG08] IEEE Std 802.1AX-2008 IEEE Standard for Local and Metropolitan Area Networks - Link Aggregation. IEEE Computer Society, 2008.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR*, 2008.

[MFP+07] Casado Martin, M J Freedman, Justin Pettit, J Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of ACM SIGCOMM '07 Computer Communication Review.* ACM, 2007.

[MG02] C. Siva Ram Murthy and Mohan Gurusamy. *WDM Optical Networks, Concepts, Design, and Algorithm.* Prentice Hall PTR, 2002.

[Ng10] Zheng Cai Alan L. Cox T. S. Eugene Ng. Maestro: A System for Scalable OpenFlow Control, 2010.

[NML98] Peter Newman, Greg Minshall, and Thomas Lyon. IP switching—ATM under IP. *IEEE/ACM Trans. on Networking*, 6(2):117–129, 1998.

[Ope] OpenFlow Switch Vendors. http://en.wikipedia.org/wiki/Openflow_Switching_Protocol.

[Pax00] V. Paxson. Computing TCP's Retransmission Timer, November 2000.

[RPC+11] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. Tritonsort: A balanced large-scale sorting system. In *USENIX NSDI'11*, 2011.

[SEE] UC San Diego SEED Cluster Project. https://sites.google.com/site/ucsdseedproject/documentation/quickstart-guide.

[SK02] Pasi Sarolahti and Alexey Kuznetsov. Congestion control in linux tcp. In *In Proceedings of USENIX*, pages 49–62. Springer, 2002.

[SKRC07] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system, 2007.

[swi] SWIG. http://www.swig.org.

[TPr] TCPProbe kernel module. http://www.linuxfoundation.org/collaborate/workgroups/networking/tcpprobe.

[WAK+10] 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 *ACM SIGCOMM '10*, 2010.

[WLL+] Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, and Yongguang Zhang. MDCube: A High Performance Network Structure for Modular Data Center Interconnection. In *ACM CoNEXT '09*.