Reducing the Cost of Operating a Datacenter Network

by

Andrew R. Curtis

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Datacenters are a significant capital expense for many enterprises. Yet, they are difficult to manage and are hard to design and maintain. The initial design of a datacenter network tends to follow vendor guidelines, but subsequent upgrades and expansions to it are mostly ad hoc, with equipment being upgraded piecemeal after its amortization period runs out and equipment acquisition is tied to budget cycles rather than changes in workload. These networks are also brittle and inflexible. They tend to be manually managed, and cannot perform dynamic traffic engineering.

The high-level goal of this dissertation is to reduce the total cost of owning a datacenter by improving its network. To achieve this, we make the following contributions. First, we develop an automated, theoretically well-founded approach to planning cost-effective datacenter upgrades and expansions. Second, we propose a scalable traffic management framework for datacenter networks. Together, we show that these contributions can significantly reduce the cost of operating a datacenter network.

To design cost-effective network topologies, especially as the network expands over time, updated equipment must coexist with legacy equipment, which makes the network heterogeneous. However, heterogeneous high-performance network designs are not well understood. Our first step, therefore, is to develop the theory of heterogeneous Clos topologies. Using our theory, we propose an optimization framework, called LEGUP, which designs a heterogeneous Clos network to implement in a new or legacy datacenter. Although effective, LEGUP imposes a certain amount of structure on the network. To deal with situations when this is infeasible, our second contribution is a framework, called REWIRE, which using optimization to design unstructured DCN topologies. Our results indicate that these unstructured topologies have up to 100-500% more bisection bandwidth than a fat-tree for the same dollar cost.

Our third contribution is two frameworks for datacenter network traffic engineering. Because of the multiplicity of end-to-end paths in DCN fabrics, such as Clos networks and the topologies designed by REWIRE, careful traffic engineering is needed to maximize throughput. This requires timely detection of elephant flows—flows that carry large amount of data—and management of those flows. Previously proposed approaches incur high monitoring overheads, consume significant switch resources, or have long detection times. We make two proposals for elephant flow detection. First, in the Mahout framework, we suggest that such flows be detected by observing the end hosts' socket buffers, which provide efficient visibility of flow behavior. Second, in the DevoFlow framework, we add efficient stats-collection mechanisms to network switches. Using simulations and experiments, we show that these frameworks reduce traffic engineering overheads by at least an order of magnitude while still providing near-optimal performance.

Acknowledgements

First and foremost, I thank my advisers S. Keshav and Alex López-Ortiz. Their advice and guidance over the years has shaped me into the researcher I am today. Their lessons will forever change the way I think and act. This dissertation could not have been written without their mentorship. I especially appreciate Keshav's detailed comments on all my work. This feedback has greatly improved my research. I thank Alex for his guidance bridging the gap between theory and systems, for pushing me to understand networks from both theoretical and systems perspectives. Chapters 3 and 4 describe work done in collaboration with Alex and Keshav.

Praveen Yalagandula at HP Labs, Palo Alto, also mentored me during much of this research. Many others at HP Labs also played a role in this thesis, including: Jeff Mogul, Jean Tourrilhes, Puneet Sharma, and Sujata Banerjee. This group of people taught me how to find research problems in practical, operational issues. Crucially to this dissertation, they helped me understand how real datacenters operate. Chapter 6 presents work I did in collaboration with this group. Chapter 5 was also written during my time at HP Labs. It presents the results of a collaboration between Praveen Yalagandula, Wonho Kim, and myself.

Others have played a role in this research as well. I am indebted to Tommy Carpenter and Mustafa Elsheikh, who also collaborated on the results described in Chapter 4. I appreciate the many discussions on this work I've had with others, especially other graduate students at Waterloo. I am especially grateful to my fellow students in the ISS4E group for beneficial discussions and comments.

I thank my committee members Ant Rowstron, Jochen Koenemann, Ken Salem, and Bernard Wong. These people pushed me to improve this dissertation by clarifying details and developing a deeper understanding of the theory behind network design and routing.

I thank Bryan Shader for introducing me to research during my undergraduate studies. I had not considered attending graduate school before my experience conducting research under his guidance. This opportunity put me on the path to this dissertation.

Finally, I thank my partner Bryanne Myers for her continued support over the years. Thanks for helping me through this journey!

The results that make up this dissertation have been previously published. Chapter 3 was originally published as reference [36]. Chapter 4 was published as [35]. Chapter 5 was published as [34]. And, chapter 6 was published as [38].

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

The scale of today's datacenters is unprecedented. Large datacenters contain 50,000–250,000 servers and consume up to 60 MW of power. Applications running on these datacenters operate at massive scale by distributing their workloads across the many available servers. For example, distributed file systems such as Google File System (GFS) [51] and Hadoop Distributed File System (HDFS) [61] provide efficient, scalable, and reliable access to data and applications like MapReduce [39] and Dryad [74] allow one to distribute computation across thousands of servers. Because these systems are distributed, they all rely on the network. However, network architectures have previously not been designed to interconnect hundreds of thousands of high-performance hosts.

First-generation datacenter network (DCN) architectures were based primarily on enterprise network architectures. These proved to be inadequate for large-scale datacenters, because of scaling and performance issues [55]. A DCN must connect hundreds of thousands of end-hosts and provide up to petabits of bisection bandwidth[1] between them. Guidelines from equipment vendors, such as Cisco [29], arrange the DCN topology as a 1+1 redundant tree[2]. Doing so results in underprovisioned networks—Microsoft researchers [55] have found links with a 1:240 oversubscription ratio in their datacenters! Such high levels of oversubscription are appropriate for enterprise networks, but not for high-performance networks like DCNs. Oversubscribed links limit server utilization because they restrict *service agility*—the ability to assign any server to any service [57]. Underprovisioned networks also are a bottleneck in modern distributed applications such as MapReduce [39], Dryad [74], partition-aggregate applications such as search [11], and scientific computing.

Because of these high-performance demands, researchers have proposed DCN architectures, that provide up to full bisection bandwidth and can scale to hundreds of thousands of servers, e.g., [10, 33, 55, 59, 60, 123]. They achieve this by using high-performance topology constructions and custom addressing, routing, and load-balancing schemes. For example, VL2 [55] requires a Clos topology, CamCube [33] requires a 3D torus topology, and BCube [59] requires a BCube topology. (Details are in Chapter 2.) The topologies used by these DCN architectures are prescriptive

[1] A *bisection* of a network is a partition of its nodes into disjoint sets of equal size, say (S, S'). This is also called a *cut*. The *bandwidth* across this cut is the sum of link bandwidths for links with one endpoint in S and the other in S'. A network where the bandwidth of all bisections is equal to half the number of servers is said to have *full bisection bandwidth*.

[2] A *1+1 redundant tree* is a topology that consists of two identical, disjoint trees. The second tree provides connectivity in the event of a single failure

constructions, that is, the topology is defined by a small set of inputs, typically the switch radix (i.e., the number of ports per switches) and a number of recursive levels. Additionally, the load balancing schemes used by some of these architectures (such as VL2's randomized load balancing, which randomly selects a path for a flow), are not effective for some workloads. For example, Al-Fares et al. [8] found that dynamic load balancing can improve aggregate throughput up to 100% versus VL2's randomized load balancing for some workloads on a fat-tree topology.

Because these architectures rely on custom topologies, they are best suited for "greenfield", or new, datacenters. Even if a DCN was built for a specific architecture, it can be challenging to expand or upgrade these networks since they are regular, prescribed constructions. Therefore, we investigate new DCN topologies—ones that support heterogeneous switch radices and link rates. To support heterogeneous switch types, we developed the theory of heterogeneous Clos networks, which generalizes the Clos network. To automate the design of these topologies, we developed an optimization framework called LEGUP (Chapter 3). Because the heterogeneous Clos network still imposes structure on the topology, we then initiated the study of unstructured DCN topologies. We present our optimization framework for unstructured DCN design, called REWIRE, in Chapter 4. Overall, we find that unstructured networks can significantly outperform networks that are based on regular topology constructions.

A DCN needs a large bisection bandwidth to increase service agility, but it also needs load balancing to enable the use of the full bisection bandwidth. Because DCN topologies contain numerous end-to-end paths for each pair of endpoints, traffic engineering can often improve the aggregate throughput by dynamically pinning flows to paths. In prior work, Al-Fares et al. proposed a system for dynamic DCN traffic engineering called Hedera [8]. They showed that Hedera can improve network performance significantly; however, their approach relies on OpenFlow [85], which has scaling issues as summarized in Section 6.2. Therefore, we study low-overhead DCN traffic engineering. We study the problem from two angles. First, we introduce a traffic engineering framework, named Mahout, that uses a low-overhead shim in the end-host's network stack to classify elephant flows, which are long-lived, high-throughput flows. These elephant flows are then dynamically monitored and routed by a centralized controller. Mahout is described in Chapter 5. Finally, we evaluate the effectiveness of using DevoFlow [86, 38], a modification of the OpenFlow framework that reduces OpenFlow's overheads. These results are described in Chapter 6.

LEGUP and REWIRE significantly reduce datacenter capital expenses. Recent estimates peg the capital cost of networking equipment—switches, routers and load balancers—at 5–15% of the total

monthly budget of a typical datacenter [57, 62]. Therefore, reducing network capital expenditure can significantly reduce a datacenter's total cost of ownership. For example, REWIRE can save up to $3 million per year in a typical datacenter with 50K servers.

Further, increasing datacenter service agility allows for a higher peak server utilization. If a 50K-server datacenter has a peak server utilization of 40% and this could be improved to 80% by providing better load balancing in the network or upgrading it, then the datacenter operator could cut the number of servers they deploy in half. As servers represent 45–60% of the datacenter's overall cost, reducing their expense is crucial to overall cost reduction.

Before discussing related work in the next chapter, we describe (1) typical datacenter applications, (2) the characteristics of an ideal DCN, and (3) the contributions of this dissertation.

1.2 Datacenter Applications

To understand the requirements of a datacenter network, we first need to understand the load applications place on the network. Applications in the datacenter broadly fall into two categories: interactive or batch. Both are typically built on top of a distributed file system, which we describe below.

1.2.1 Interactive applications

Interactive application need fast response times to keep users engaged. Examples of interactive applications include: search, web, games, and messaging. Adding 100s of milliseconds of latency to response times has been shown to decrease website usage [108]; therefore, interactive datacenter applications need to respond to queries as quickly as possible.

To minimize response times, it is common for computation-heavy applications (such as search) to use a partition-aggregate computation model. Under this model, computation is partitioned across many end-hosts (up to thousands of servers), and then their responses are aggregated by a few machines. To satisfy end-user SLAs, a typical query resolved by a partition-aggregate computation must be answered within a deadline of 200 ms or less [11]. To achieve this, an aggregator partitions the query and assigns a task to each worker. Each worker is given a deadline of 10–100 ms to complete their task. If a deadline is missed, the aggregator ignores that response, lowering the quality of the result[3].

[3] Partition-aggregate workloads cause another problem: *incast*, a form of congestion collapse that occurs when

Other common interactive applications include web and data-transfer services. Both are well-suited to distributed implementations. Because modern web services are typically implemented with a tiered architecture, they create intra-datacenter network traffic as well as egress traffic that leaves the datacenter. These egress flows tend to consist of mostly "mice" flows, that is, very short-lived flows, and have few "elephant" or long-lived flows [14]. Data-transfer services, such as video and music streaming services, tend to create elephant flows; however, it is often acceptable for these flows to have have a relatively low throughput. This happens if, for example, the bit rate of an audio file is 128 Kb/s. In this case, the file only needs to be sent at 128 Kb/s since the user listens in real-time to the audio.

1.2.2 Batch jobs

Many datacenter applications are batch jobs. That is, their results are not needed within a strict time limit. For example, an application that analyzes log files to learn user behavior characteristics is a batch job, because its results are not immediately needed by a user. A few other types of batch jobs include:

- Analytics
- Analyzing user behavior
- Machine learning
- Data mining
- Natural language processing
- Log analysis
- Image analysis

The use of distributed, data-flow computation frameworks (such as [39, 74, 126]) make it easy to implement these types of jobs at huge scale.

For example, frameworks that implement MapReduce [39] require that developers implement a "map" function and a "reduce" function. When the job is executed, the MapReduce system divides the input across workers and executes the map function in parallel. Once complete, the results of the map phase are sent to all the reduce-phase workers, where the reduce function is executed in

many workers reply simultaneously to an aggregator. This behavior has prompted the study of DCN-specific TCP variants [11, 118, 122]

parallel. Doing so puts a high load on the network, and if the network does not have enough bisection bandwidth, it can be a bottleneck in a job's run-time [13]. In general, batch jobs can create huge amounts of network traffic, because they may process terabytes, or more, of data.

1.2.3 Distributed file systems

To provide high availability and scalability, storage in the datacenter is often provided by a distributed file system (DFS), which allows any server to access the data stored at any other server in a transparent, scalable way. Google has described their proprietary DFS, called the Google file system [51], and the Hadoop distributed file system (HDFS) [6] is a popular open source implementation.

Because a DFS allows servers to access remote data, this type of file system can heavily utilize the network if applications do not take data locality into account when making scheduling decisions. Additionally, a DFS can place load on the network when replicating data. Typical DFS implementations maintain at least three copies of each data block, so if one of these copies fails, the DFS creates another replicate.

1.3 Datacenter Network Goals

An ideal DCN should be agile, scalable, flexible, resilient, manageable and cost-effective. We now describe each of these traits in detail.

Agile

The switching fabric is never the limiting factor in the transmission rate between end-hosts in an ideal DCN. The end-hosts' network interface cards (NICs) are the only limitation on transmission rates in such a network. Therefore, we would like to define network agility so that it measures a network's ability to handle any possible workload. A network with *full agility* can handle *any* traffic matrix feasible under the server NIC rates. A more precise definition is given shortly.

At first blush, this goal sounds extreme. However, it is motivated by the available DCN measurement studies. From them, we know that DCN workloads exhibit a high degree of variance [76, 18, 55, 17]. In a 24 hour time period, there can be an order of magnitude difference between the peak

and minimum load in the DCN [66]. The network needs enough capacity to handle the peak load, and it needs to be flexible enough to cope with future workloads. DCN traffic is also unpredictable over short periods [76, 17]. Full details of DCN traffic are given in Section 2.1.

To precisely define network agility, we need to introduce a couple of terms. A network can *feasibly route* a traffic matrix (TM) if there exists a routing of the traffic demands such that no link's utilization is greater than 1. Note that we assume *multipath routing*, which means that the traffic from node s to node t can be split across multiple paths. Mathematically, if the NIC rate of server i is given by $r(i)$, then a hose TM T has:

$$\sum_{j \in V} t_{ij} \leq r(i) \text{ and } \sum_{j \in V} t_{ji} \leq r(i)$$

where V is the set of all servers and t_{ij} is the i-j entry in the TM T. The set of all hose TMs is known as the set of *hose traffic matrices* and this model is known as the *hose model* [43]. The hose traffic matrices form a polyhedron, and we denote them by $\mathcal{T}$. Now, we can define a network with full agility as one that can feasibly route all hose traffic matrices.

Some networks may only be able to route a scaled version of the worst-case TM. Therefore, the **agility** of a network G is the maximal value λ such that $\lambda \cdot T$ can be feasibly routed for all $T \in \mathcal{T}$, where $\mathcal{T}$ is the hose traffic matrices for G. An ideal network in our setting has $\lambda \geq 1$, meaning that it can feasibly route all hose TMs without over-utilizing any link. The hose model was introduced in the context of provisioning virtual private networks and the intuition is that each node has set ingress and egress rates, but we do not have any additional insight as to the amount of traffic one node will send another, so the network should be designed to feasibly route *any* possible traffic matrix possible given the nodes' ingress/egress rates [55].

Defining agility in terms of cuts. A result of this dissertation is to prove that agility can be equivalently described in terms of network cuts. This is a generalization of *bisection bandwidth* [100]. A *bisection* of a network is a segmentation of the network into two parts, say S and $\overline{S}$, such that each set contains the same number of nodes. The capacity of a bisection is the available link bandwidth across the bisection (i.e., the sum of link rates for links with one endpoint in S and the other in $\overline{S}$). Then, the bisection bandwidth of a network is the worst-case capacity of any bisection of the network's nodes. Note that this definition does not account for node rates, because it only depends on the link bandwidth, not the amount of traffic that may be sent across those links. Because of this limitation, bisection bandwidth is not a good metric for networks with heterogeneous node rates.

We generalize this notion to measure the worst-case bandwidth across any cut and to account for node rates. A *cut* is a partition of a network's nodes into two sets, denoted by S and $\overline{S}$. A network's *normalized cut bandwidth* is then the worst-case bandwidth across any network divided by the maximum amount of flow that may cross that cut, and is denoted by $\text{bw}(G)$. Let the bandwidth of a link e be denoted by $w(e)$. Then, the normalized cut bandwidth of a network $G = (V, E)$, where V is its nodes and E is its links and we assume G is connected, is:

$$\text{bw}(G) = \min_{S \subseteq V} \frac{\sum_{e \in \delta(S)} w(e)}{\min\{\sum_{i \in S} r(v), \sum_{i \in \overline{S}} r(v)\}}$$

where $\delta(S)$ is the set of edges with one endpoint in S and another in $\overline{S} = V - S$. We always deal with the normalized version of cut bandwidth in this dissertation, because it takes into account heterogeneous node rates.

A result of this dissertation is to show the equivalence of normalized cut bandwidth and agility. This is proved in Theorem 4, which shows that there is a form of a min-cut, max-flow theorem that exists for the hose traffic model.

As an example of agility, consider a network consisting of two switches, each attached to 48 servers at 1 Gbps and a single 10 Gbps port that connects the switches. The agility of this network is $10/48$. More generally, if we have n servers attached to the first switch and m attached to the second, then we have the agility of the network is $10/\min\{n, m\}$. Here, we divide by the minimum of the two values because the hose TMs do not allow any server to send or *receive* more than 1 Gbps of traffic, that is, even if there are 48 servers attached to one switch and 1 server attached to the other, the maximum receiving rate of lone server is 1 Gbps so no more than that will ever cross the connecting 10 Gbps link.

Scalable

The network should not be the limiting factor in determining the number of servers deployed in a datacenter. An ideal DCN architecture and topology should enable connection of up to hundreds of thousands of servers.

Flexible

Most datacenters grow and evolve over time[4]. The network should be flexible enough to accommodate this. At short time scales, servers may be turned on or off to match variable workloads. As a result, virtual machines (VMs) need to transparently migrate across servers [15]. Over longer time periods, an operator may need to add or remove servers and other IT equipment. The network should permit this constant evolution.

Flexibility reduces costs because it improves agility, can reduce energy cost by shutting off underutilized equipment, and provides a cost-effective growth strategy to expand the datacenter. To enable flexibility, venders have started selling container datacenters, which house up to a couple thousand servers in a standard 20–40' shipping container. Containers provide flexibility—it is easy to grow a datacenter by deploying new containers.

Resilient

Failures are common in large datacenters. The network should therefore be able to withstand multiple port/switch failures, and failures should have minimal impact on the network. A recent study found that half of all DCN failures involve four or more devices [55]; therefore, traditional 1+1 redundancy is not enough in the datacenter, because more than a single failover path is needed.

As a numerical example, consider a situation where a row of 1200 servers is partitioned from the network, which results in $3 million worth of servers[5] being disconnected until the failure is resolved. This may take under an hour (98% of the time) or over 10 days (0.09% of the time) [55]. If it takes 10 days to bring this row of servers back online, then the datacenter operator would lose at least $27,000 in lost server time (assuming a three year amortization period for the servers). A resilient network lowers the cost of failures, because it prevents servers from being disconnected as a result of a network failure.

Beyond physical resiliency, in settings where multiple tenants share a switching fabric, the network should be resilient to malicious users that launch DoS attacks. At the same time, the network should not punish legitimate users with network-intensive workloads. Both can be achieved if the network provides isolation. Therefore, a resilient cloud DCN architecture provides performance isolation between tenants.

[4] As an example, James Hamilton said that Amazon adds compute capacity to their datacenters every day [63]

[5] Assuming a commodity price of $2,500 per server. High-end servers can cost more than double this.

Amortized Cost	Component	Sub-components
~45%	Servers	CPU, memory, storage systems
~25%	Infrastructure	Power distribution and cooling
~15%	Power draw	Electrical utility costs
~15%	Network	Links, transit, equipment

Table 1.1: Cost breakdown the components of a datacenter. These numbers are taken from [57].

Manageable

Large-scale DCNs have thousands of network devices. This scale makes them difficult to manage, especially if devices have to be manually configured. Manual management of such a network is error-prone and expensive [57]. Instead, the network should be easy to manage and switches should support "plug-and-play" functionality. Therefore, an ideal DCN is self-managed and does not require any manual configuration. A manageable network can reduce operation costs, as it allows cloud datacenter owners to employ fewer employees.

Cost-effective

A datacenter network should minimize capital and operational costs. Network equipment is responsible for slightly under 15% of the capital cost of a typical datacenter (from [57]; see Table 1.1).

Capital costs: is the money spent on infrastructure and equipment. This money goes to buildings, servers, network equipment, power distribution and cooling infrastructure.

Operational costs: are incurred while operating the datacenter. Here, the most expensive costs come from power draw and management.

1.4 Contributions

We make three major contributions in this dissertation:

- We introduce the datacenter network upgrade and expansion problem and design and implement an optimization framework, called LEGUP[6], that designs network topology upgrades for

[6] Short for **leg**acy datacenter network **up**grade framework.

legacy datacenter networks. We show that, for our test scenarios, it is twice as effective at designing high-bandwidth networks than previous approaches.

- We propose a framework, named REWIRE, that uses optimization to design *unstructured* datacenter networks. We explore the design space of unstructured topologies, and we find that unstructured networks have up to an order of magnitude more bisection bandwidth than networks designed by previous approaches, including those found by LEGUP.
- We analyze the overheads of flow management in the datacenter, and find that such functionality has high implementation overheads. We propose two low-overhead datacenter traffic engineering frameworks: Mahout and DevoFlow. Both solutions use a centralized controller to dynamically orchestrate the paths taken by elephant flows. They differ, however, on how to detect elephant flows. Mahout uses the end-hosts for this task, while DevoFlow is an entirely in-network solution. Both frameworks can increase aggregate throughput up to 55% depending on the workload, while sending up to two orders of magnitude fewer control messages than a naive OpenFlow-based implementation.

Together, these contributions significantly reduce the cost of operating a datacenter network. The graph theory and algorithms behind LEGUP and REWIRE enable the physical network infrastructure to be agile, scalable, flexible, resilient, and cost-effective. Our flow management frameworks Mahout and DevoFlow help with the management of the non-standard network designs found by these frameworks, and help maximize agility by increasing network performance. We describe these results in Chapters 3–6. They were originally published as references [34, 35, 36, 37, 38]. However, before introducing these results, we survey the work others have published on datacenter networks in Chapter 2 to help clarify where our work fits in the taxonomy of DCN research.

Chapter 2

Related work

We can divide the DCN design space in a layered hierarchy as shown in Figure 2.1. Most DCN architecture proposals affect multiple layers in this hierarchy. For example, architectures like VL2 [55] and BCube [59] propose novel solutions for topology design, addressing, routing and load balancing. Because a DCN is under control of a single entity, widespread changes like this are acceptable in a DCN architecture. This is in contrast to the Internet at large, where modifications to even a single layer are extremely difficult.

We now motivate the DCN design problem by describing their workloads. Then, we describe the related work for the layers in the taxonomy that are related to the results presented in this dissertation. That is, we describe the related work on topology design, load balancing, and configuration. We end this chapter by providing an overview of the OpenFlow protocol, because we base our traffic engineering solutions in Chapters 5 and 6 on OpenFlow.

2.1 Workloads in datacenter networks

To motivate the design of DCN topologies, we survey measurement studies of DCN workloads. Overall, DCN workloads exhibit a high degree of variance. In a 24 hour time period, there is generally at least an order of magnitude difference between the peak and minimum load in the DCN [66]. The network needs enough capacity to handle the peak load, and it should permit increases in capacity to meet future demand. As a result, DCNs tend to be lightly utilized on average [62]. Additionally, under-provisioned networks constrain job placement schedulers. Such networks do not have enough bandwidth to quickly move jobs to idle servers. As a result, there have been many recent proposals for high-bandwidth DCNs, and especially for networks with full bisection bandwidth [10, 33, 55, 59, 60, 123].

DCN traffic is unpredictable over short time periods. Few detailed studies of datacenter traffic have been published; however, the studies to date indicate that DCNs can exhibit highly variable traffic [17, 18, 55, 76], that is, the traffic matrix (TM) in a DCN shifts frequently and its overall volume (i.e., the sum of its entries) changes dramatically in short time periods.

In Chapters 5 and 6, we consider centralized flow management. Such a routing controller in a DCN needs to respond to traffic fluctuations quickly and effectively. A study by Kandula et al. found that the median inter-flow arrival time of a flows in a 1500-server datacenter was 10^5 flows per second [76], so a centralized scheduler must route 100 flows every millisecond.

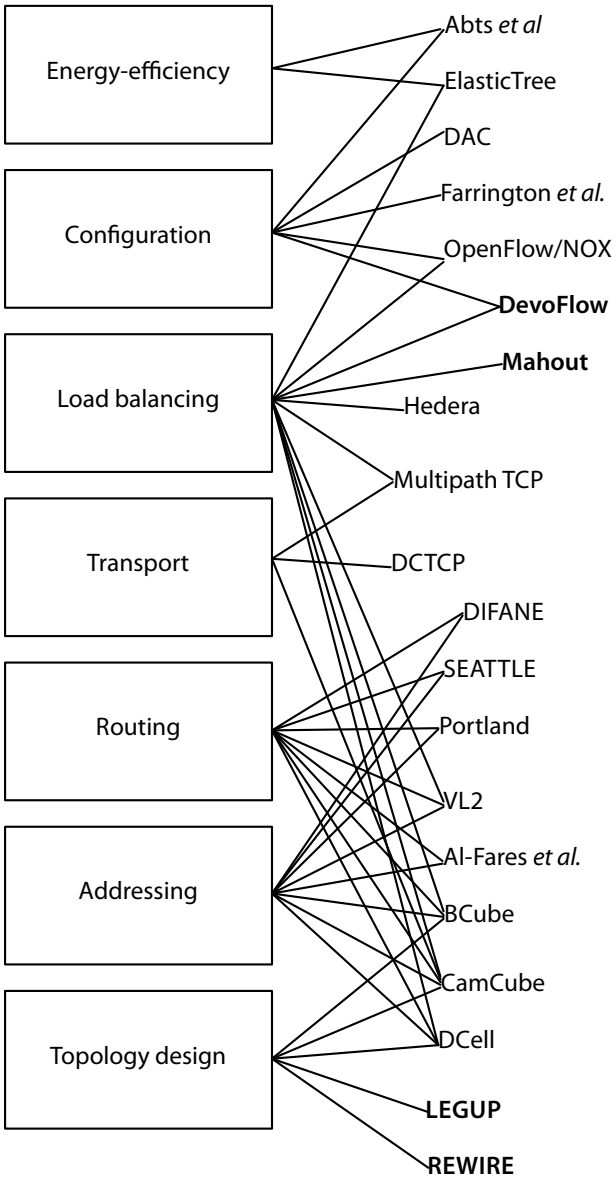


Figure 2.1: Taxonomy of datacenter network solutions. Our contributions are shown in bold typeface.

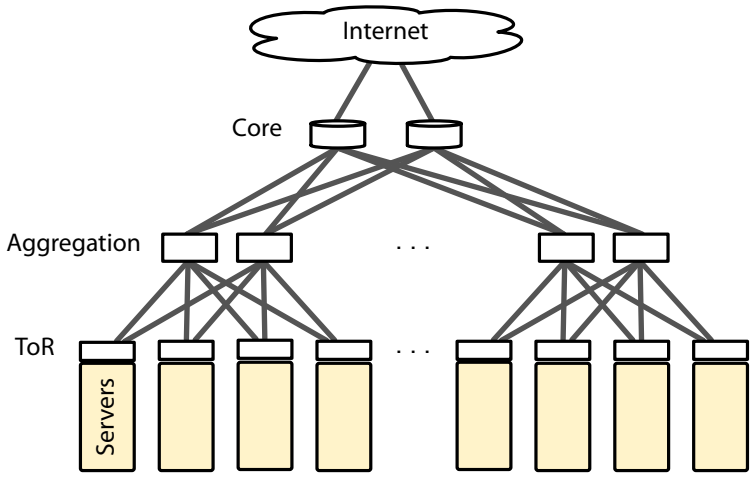


Figure 2.2: A 1+1 redundant tree.

2.2 Topology design

The study of interconnection network topologies dates back to telephone switching networks, where the goal was to interconnect telephone circuits. A variety of topologies have been proposed over the years. Generally, these constructions aim to interconnect hundreds of thousands of endpoints with high bisection bandwidth. For example, the following constructions have been proposed: Clos [30], Beneš [16], de Bruijn [107], flattened butterfly [78], HyperX [7], hypercube [123], DCell [60], and BCube [59]. The general theme of work in this area is to "scale-out", that is, use multiple commodity switches in place of a single high-end switch. The goal of this design pattern is to reduce the cost of the network, since commodity switches are inexpensive.

The traditional DCN topology is a *1+1 redundant tree*. It consists of two disjoint trees, and thus provides a primary as well as a backup path between each pair of servers. The leaves of the trees are called *top-of-rack* (ToR) switches, which connect to 20–80 servers per rack. A typical ToR switch has 48 1 Gbps and 2–4 10 Gbps ports. It uses its 1 Gbps ports to attach to servers, and uses its 10 Gbps ports to connect to two *aggregation switches*. Each aggregation switch connects to ToR switches and "up" to two *core switches*. A 1+1 redundant tree is shown in Figure 2.2. The core switches connect the switching fabric to border routers for Internet connectivity. In this dissertation, we deal only with intra-datacenter communication, and so we ignore the border routers in our analysis. Al-

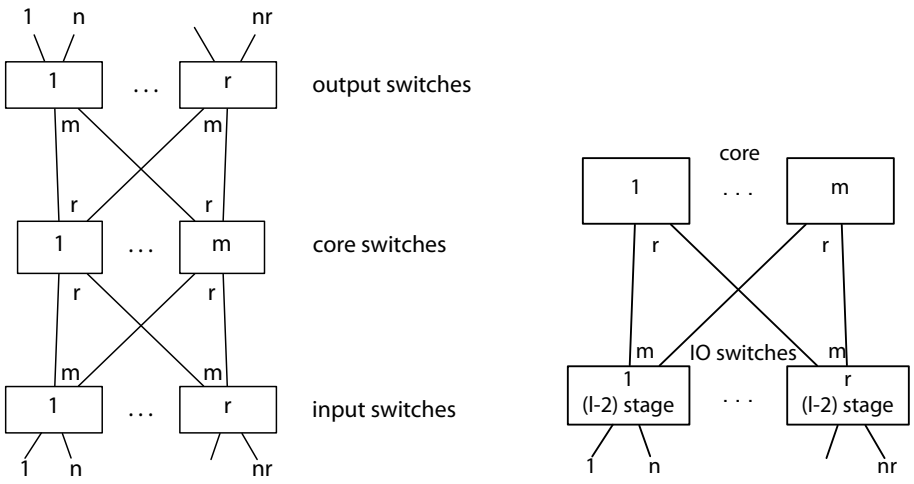


Figure 2.3: At left is a 3-stage, unfolded Clos network. At right is a folded l-stage Clos network. Each IO switch is a subnetwork with $l - 2$ stages.

though widely deployed, especially in enterprise networks, this architecture has two major drawbacks—poor reliability and insufficient bisection bandwidth—besides many other minor problems, as detailed by Greenberg et al. [55, 57].

Although researchers agree that the 1+1 tree is inadequate, determining the ideal DCN topology is still a open research challenge. The ideal topology depends not only on the expected workload, but also the cost structure of switches and cables. Both of these quantities change rapidly over time. The leading candidate, however, is the the Clos network [30], which was proposed in 1954 by Charles Clos to wire telephone switching networks. The HyperX topology is interesting from a theoretical perspective because it generalizes the hypercube and flattened butterfly networks [7]. Several other novel topology constructions have been proposed for the design of DCNs [55, 10, 59, 60, 123, 84]. We describe these topologies below.

2.2.1 Clos network and the fat-tree

The Clos network [30] and its fat-tree configuration [84] have been proposed as DCN topologies by Al-Fares et al. [10] and Greenberg et al. [55]. The benefits of a Clos topology are that all paths between ToR pairs have the same length and it is inexpensive to build using commodity switches.

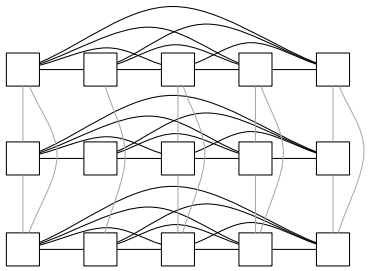


Figure 2.4: A 2-dimensional HyperX topology with $I \equiv (5, 3)$.

In addition, the Clos topology structure is well-suited to *randomized load balancing*, where the path selected for a flow is selected randomly.

Mathematically, a 3-stage Clos network [30], denoted by $C(n, m, r)$, is an interconnection network where nodes are partitioned into *stages*. The goal of the network is to interconnect *inlets* and *outlets*. The Clos network has three stages as shown in Figure 2.2.1. The first stage is called the *input switches*; it consists of r switches, each with n inlets and m uplinks. The *radix* of a switch is the number of ports it has. The radix of each input switch must be at least $n + m$. The second stage is called the *core*. Switches in the core do not connect to any inlets or outlets; instead, there are m core switches, each with radix $2r$. Each core switch connects to each input and output switch. The third stage consists of the *output* switches. There are r output switches, each with n outlets and m downlinks. We refer to the links from a stage to a higher stage as *uplinks* and the links from a stage to a lower stage as *downlinks*.

A *folded Clos network* places input and output layers top of each other, which we use throughout this dissertation. In a folded Clos network, the input and output switches are the same devices, so we refer to them as input/output (IO) switches. Because IO switches do not directly attach to each other, the Clos network is called an *indirect network*.

The recursive nature of Clos network means that we can limit our study to 3-stage Clos networks. An l-stage Clos network is recursively composed of 3-stage Clos networks. In an l-stage Clos network, each input and output switch is replaced by an $(l - 2)$-stage network. An example of a recursive, folded Clos topology is shown in Figure 2.3(b).

2.2.2 HyperX

HyperX [7] is a *direct-connect topology*, that is, IO switches connect to other IO switches. This differs from the Clos network, where ToR switches only connect to aggregation switches—not to each other. HyperX is designed to take advantage of high radix switches (i.e., switches with hundreds of ports), and it generalizes both Hypercubes and flattened butterfly networks, which makes it flexible. Mudigonda et al. have compared the cost of building a HyperX network against the cost to build a Clos network [89]. They found that the two topologies have similar costs, and the lowest-cost option depends on the size of the datacenter and the cost of switches and links.

A HyperX network is constructed as follows. Each HyperX IO switch connects to T inlets/outlets. The switches are arranged in an L-dimensional integer lattice. Each dimension k contains S_k points. Then, each switch is identified by a coordinate vector in this space, $I \equiv (I_1, \ldots, I_L)$ where $0 \leq I_k < S_k$ for each $k = 1, \ldots, L$. Switches in each dimension form a clique. We show a 5×3 HyperX topology in Fig. 2.4.

HyperX has some support for heterogeneous link rates. All links in dimension i have rate K_i, and so the link rates are described by $K \equiv K_1, \ldots, K_L$. HyperX also supports some level of switch radix heterogeneity because of its support for heterogeneous link rates; however, one cannot have arbitrarily heterogeneous switches—their port speeds and radices are prescribed by the topology.

2.2.3 DCell

The DCell [60] topology aims to interconnect a huge number of servers, up to a couple million, with low radix switches (for example, switches with 8 ports). The DCell construction is recursive.DCell(0) is defined as n servers connected to an n-port switch. A DCell(k) is constructed from $n+1$ DCell$(k-1)$ networks. When each subnetwork is treated as a single logical node, DCell(k) forms a clique, that is, each logical node (a subnetwork) has a single edge to each other logical node. These links are realized physically as follows. Assign each server an ID: $< s_k, \ldots, s_0 >$ where s_i is the *level-ID* of the server or subnetwork s_i. A level-ID is an ordering on the servers (for a DCell(0) network) or the logical subnetworks of a DCell(k). Then, at level i, connections are added between servers such that a level-i link connects to a different DCell$(i-1)$, but within the same DCell(i).

A DCell topology typically has twice the bisection bandwidth than a tree, but is unclear how well it performs or much it costs compared to other topologies. In particular, the DCell topology does not provide as much bisection bandwidth as a Clos network.

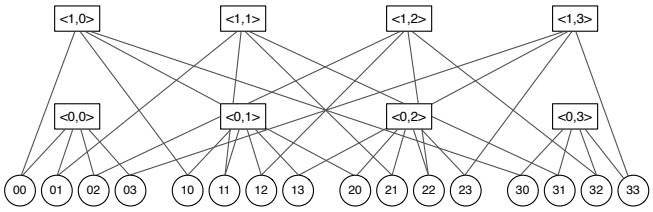


Figure 2.5: A BCube(1) construction with $n = 4$.

2.2.4 BCube

BCube [59] is a DCN architecture targeted at datacenters with up to a few thousand servers. This architecture uses a novel topology, called the BCube topology. An example is shown in Figure 2.5. The BCube topology is a recursive construction. A BCube(0) is n servers connecting to an n-port switch. A BCube(k) is construction from n BCube$(k-1)$ networks and n n-port switches. Switches in the BCube topology only connect to servers, and servers only connect to switches, so BCube is an indirect topology. The switches act as dumb crossbars, and servers relay traffic for each other. This construction provides multiple edge-disjoint end-to-end paths between each server. One side effect is that paths between end-hosts in a BCube network have varying lengths.

This design point of BCube was selected because 40 ft shipping containers make excellent housing for about 1000–3000 servers. A number of commercial container datacenters are available on the market today, for example, Sun's Modular Datacenter [111], HP's POD [69], and IBM's modular datacenter [73]. Because it is common to combine multiple modular datacenters to form a larger datacenter, the authors later extended BCube to network a few dozen container datacenters together [123]. The BCube design point makes many practical issues easier, for example, cabling is less an issue in a $40 \times 10 \times 10$ ft box.

2.2.5 MDCube

MDCube [123] extends BCube to interconnect multiple containers. It uses BCube as the intra-container architecture and MDCube as an inter-container architecture. MDCube interconnects the containers with a generalized hypercube topology.

The MDCube architecture is interesting for a number of reasons. First, it interconnects containers using 10 Gbps ports on BCube's commodity switches. Such switches (with 48 1 Gbps ports and

4 10 Gbps ports) are commodity switches today, and an estimated street price for such a switch is $5,000[1]. MDCube connects the 10 Gb ports in a hypercube topology. Second, MDCube's design creates clusters of 1000–3000 servers with a very high capacity network interconnected by a network with less bandwidth. This is similar to *slimming* a fat-tree [99]. Such a provisioning scheme is beneficial because very few jobs need more than a 1000 servers [125].

2.2.6 Heterogeneous topology constructions

The only construction we are aware of that can connect a heterogeneous set of switches while guaranteeing optimality is that of Rasala and Wilfong [105], who gave a strictly nonblocking[2] construction for networks with heterogeneous IO switches. This traffic model is not applicable to datacenter networks because DCNs are packet switched and buffers prevent "blocking". In their approach, each flow has a unit rate, but flows can be rearranged across crossbars. This model is generalized by the primary traffic model used in this dissertation—the hose model (details were given in Section 1.3). Another limitation of their work is that their topology constructions only connect IO switch sets with two types of switches and they do not support heterogeneous switch port speeds.

2.3 Load balancing

To provide high bisection bandwidth with commodity switches, DCN topologies provide a multiplicity of end-to-end paths. This brings about a challenge: how should flows be scheduled across these multiple paths?

2.3.1 Oblivious load balancing

One load balancing approach is to assign each flow randomly to a path. This zero-overhead approach is known as *oblivious routing* or *randomized routing*. The path taken by a flow from node i to node j is randomly selected from a probability distribution over all i to j paths. Kodialam et al. have shown

[1] There is, however, a premium for the the 10 Gbps ports. A 48-port 1 Gbps switch costs a mere $1,500 on the street as of this writing.

[2] A network with n inputs and n outputs is *strictly nonblocking* if it can setup any n calls independent of the call arrival order. This differs from a network which is *rearrangeably nonblocking*, which can supports n call setups if all calls can be rearranged (i.e., re-routed through the network) when a new call setup is requested.

that oblivious routing can achieve at least 50% of the optimal dynamic routing in the worst-case for any topology [81]. Their model, however, assumes that all end-to-end flows can be fractionally split across multiple paths. This type of multipath routing is not possible on today's switching hardware. Even if it is possible, it may cause out of order delivery problems and interfere with TCP's congestion control mechanisms. In fact, it has been shown that oblivious routing performs poorly on a fat-tree topology when the traffic mix contains many elephant flows. Al-Fares et al. found their flow scheduler, Hedera [9], can increase throughput 113% compared to oblivious routing by performing dynamic flow scheduling.

Throughout this dissertation, we assume the multipath form of oblivious routing. This is one of many models studied in the oblivious routing literature. For example, the celebrated "VPN Theorem" of Goyal et al. assumes single-path routing [54]. See [25] for more details.

2.3.2 Reactive flow scheduling

Hedera [9] introduces a centralized controller that dynamically schedules flows to maximize their aggregate throughput. The controller periodically pulls flow statistics from each switch, which it uses to identify the set of elephant flows (that is, flows which could use more than 10% of server network interface card (NIC) bandwidth if they were not constrained by the network). Such flows are identified using an iterative demand estimation algorithm that uses statistics on all flows as input. An optimized routing of elephant flows is found using simulated annealing. The controller modifies forwarding table entries at the switches to re-route all elephant flows on optimized paths. There are two critical problems with Hedera's architecture. First, they rely on OpenFlow switches. We [86, 38] pointed out that OpenFlow does not scale well, and is ill-suited for current datacenters. Second, their reactive mechanism requires timely statistics to be effective. Hedera's evaluation assumed that DCN flow sizes follow a Pareto distribution, which is not the case in practice—DCN flow sizes actually follow a power law according to recent measurements [76]. We found that Hedera's control loop needs to be faster than 500 ms to achieve near-optimal results[3]. Achieving such a short control loop is not currently possible: it takes too long to pull the statistics from switches and to update the forwarding tables.

[3] This number was found using flow-level simulations of a 1600-server datacenter.

2.3.3 Online scheduling

Another approach to flow scheduling is to use an online scheduler, that is, the path for each flow is selected when the flow is started. An example of online scheduling is to greedily route a flow along the path with least congestion. A challenge here is to process each flow quickly. Flows arrive very quickly in a datacenter with thousands of end-hosts and many flows are latency-sensitive. For example, flow installation using the NOX OpenFlow controller[4] can take up to 10 ms [114]. Recall that, partition-aggregate jobs have deadlines of 10–100 ms [11], so a 10 ms flow setup delay can consume the entire time budget. Therefore, online scheduling is not suitable for latency-sensitive flows.

2.3.4 End-host-based load balancing

An alternative approach is to expose all end-to-end paths to end-hosts, and then adaptively find the best path for a flow. That is, if an end-host s needs to send a flow to end-host t, then s is aware of the available paths (by any mechanism) and so s selects a path at random from the available options. If the selected path does not provide enough bandwidth, then s can probe the other available paths, and *adapt* the flow's routing to place it on the highest-throughput path.

The BCube architecture [59] uses this form of load balancing. End-hosts can compute the available end-to-end paths, because they are encoded in the addressing scheme. Each server is addressed by a label of the form $< l, s_{k-1}, \ldots, s_0 >$ where $0 \leq l \leq k$ is the level of the switch and $s_j \in [0, n-1]$ for $j \in [0, k-1]$. For servers A and B, the Hamming distance of their addresses indicates the length of the shortest paths between them, and the maximum shortest path distance between two servers is $k+1$.

Another architecture that utilizes end-hosts for load balancing is SPAIN [88]. SPAIN provides multipath routing over arbitrary topologies by dividing the network into a set of spanning trees. The set of spanning trees is selected by an algorithm, and each one is implemented by a VLAN. End-hosts then perform adaptive routing over the VLANs.

[4] OpenFlow is described in Section 2.5. The network "operating system" NOX is an open-source, extensible OpenFlow controller. It is available for download at `http://www.noxrepo.org/`.

2.4 Configuration

Configuring a large DCN is expensive and error-prone. There has been some recent work to simplify this process, which we now describe.

2.4.1 Automatic assignment of addresses

The scale of DCNs forces the use of compact routing schemes, which is usually implemented by encoding locality and topology information into the addresses of switches and servers (e.g., [60, 59, 90, 55]). This creates a management challenge: how to automatically configure network addresses?

Chen et al. [27] proposed DAC: a generic and automatic Datacenter Address Configuration system. DAC automates the assignment of IDs to network devices. It begins with a network blueprint which specifies the logical ID (e.g., IP addresses in many architectures encode locality information) of switches and servers. DAC then automatically learns devices IDs (e.g., MAC addresses), and then finds a mapping of logical to device IDs. This mapping is found by modeling the problem as a graph isomorphism problem. The graph isomorphism problem is not known to be in P or NP, but is believed to be a computationally challenging problem. Therefore, Chen et al. propose some heuristics to speed the graph isomorphism search.

An interesting side effect of Chen et al.'s design of DAC is that it can programmatically identify mis-wirings. This is a real problem in todays DCNs because thousands of wires are installed by hand. Unfortunately, in their approach, all servers and switches must be shut off and then turned back on in order for DAC to learn the device IDs. This limitation means that their solution is likely to be of limited use, since production datacenters can rarely turn off all equipment.

2.4.2 Reducing cabling complexity

Cabling is a major issue in datacenter networks. For instance, consider the architecture proposed by Al-Fares et al. [10]. They suggest building a non-oversubscribed DCN for 27K servers using commodity 48-port 1GbE switches. This network provides 27.648 Tb/s of bisection bandwidth. When one considers only the cost of the switches, this network seems like a bargain at $4.3 million[5]; however, it is nearly impossible to build because it would require 1,128 separate cable bundles [48]. Each

[5] 2880 switches, each for $1,500.

bundle is manually routed and installed. Farrington et al. calculate that this would require 226,972 m of cable, which weighs in at over 9,500 kg [48]. If each link costs $50 to install (which is reasonable based on published link prices [89]). This means that this network costs over $11 million to wire—or 72% of the cost of the network is cabling.

To solve the cabling problem, Farrington et al. [48] argue that the ideal DCN topology has all ToR switches connected to a single nonblocking switch. They realize this by designing a switch with 3,456-ports running at 10 Gbps using merchant silicon ASICs (i.e., very low cost ASICs) as their basic building block. Today, ASICs containing 24-ports at 10 Gbps can be purchased for as little as $410. Now, this ASIC is not a fully featured switch—it is only the data-plane and lacks a control-plane. Farrington et al. arrange these ASICs as a fat-tree to build their switch architecture. They find this network architecture costs 52% less than a comparable fat-tree built from commodity switches. The big reduction in costs comes from cabling: Farrington et al.'s design uses only 96 long, cumbersome cables, as compared to the 6,912 such cables used in the fat-tree built from commodity switches.

Another approach to reducing cabling complexity is to build the network with high bandwidth links. For example, VL2 uses 1 Gb NICs on the servers, and 10 Gb links in the switching fabric [55]. Since a single 10 Gb link aggregates ten gigabit links without any oversubscription, the switching fabric can be built with an order of magnitude fewer links. The drawback of this approach is that high bandwidth links are typically more expensive than slower links.

2.5 OpenFlow

OpenFlow [85] is a protocol that aims to separate a network's data-plane from its control-plane. Its goal is to open traditionally closed designs of commercial switches to enable network innovation. OpenFlow has been the basis for many recent research papers (e.g., [8, 64, 83, 95, 109, 23, 94, 115]), as well as for hardware implementations and research prototypes from vendors such as HP, NEC, Arista, and Toroki.

OpenFlow switches form the network's data-plane, and the control-plane is implemented as a distributed system running on commodity servers. A switch's data-plane maintains a flow table where each entry contains a pattern to match and the actions to perform on a packet that matches that entry. OpenFlow defines a protocol for communication between the controller and an Open-

Flow switch to add and remove entries from the flow table of the switch and to query statistics of the flows. In OpenFlow, the control-plane is centralized, unlike traditional networking, which uses a distributed control-plane, with the data-plane and control-planes operating on the same hardware. Upon receiving a packet, if an OpenFlow switch does not have an entry in the flow table or TCAM[6] that matches the packet, the switch encapsulates and forwards the packet to the controller over a secure connection. The controller responds back with a flow table entry and the original packet. The switch then installs the entry into its flow table and forwards the packet according to the actions specified in the entry. The flow table entries expire after a set amount of time, typically 60 seconds. OpenFlow switches maintain statistics for each entry in their flow table. These statistics include a packet counter, byte counter, and duration. The OpenFlow 1.0 specification [3] (the only widely implemented version of OpenFlow currently) defines matching over 12 fields of packet header. The specification defines several actions including forwarding on a single physical port, forwarding on multiple ports, forwarding to the controller, drop, queue (to a specified queue), and defaulting to traditional switching. To support such flexibility, current commercial switch implementations of OpenFlow use TCAMs to store the flow table.

[6] TCAM is an acronym for *ternary content-addressable memory* and is a type of high-performance memory that supports lookups containing "don't care" or wildcard characters.

Chapter 3

LEGUP: Designing Heterogeneous, Tree-like Datacenter Networks

3.1 Introduction

As described in the previous chapter, most current datacenter networks use 1+1 redundancy in a three-level tree topology. This approach cannot provide full agility. This reduces server utilization, because servers cannot be assigned to services rapidly enough. That is, a server is assigned to a specific service, and it is underutilized if load on that service is not high enough. Recent work has addressed this problem by providing enormous bisection bandwidth for up to hundreds of thousands of servers [55, 10, 59, 60, 123, 33]. This allows services to be assigned to servers dynamically. However, these architectures use topologies that assume identical switches, each with a prescribed number of ports. Therefore, adopting these solutions in a legacy datacenter often comes at the cost of replacing nearly all switches in the network and rewiring it. This is wasteful and usually infeasible due to sunk capital costs, downtime, and a slow time to market.

In this chapter, we present LEGUP, an optimization framework to help operators increase network agility and reliability without needing to throw out their existing network devices. However, this results in the creation of heterogeneous datacenter network topologies, which have not been sufficiently studied in the past. Therefore, we first develop the theoretical foundations of heterogeneous Clos networks, that is, we generalize the Clos network [30] to allow support for multiple link rates and switches with differing port counts. Our topology is provably optimal in that it uses the minimal amount of link capacity to feasibly route all hose TMs. Previous work has only considered heterogeneous interconnection networks under a different traffic model [105], which is not applicable to datacenter networks, as discussed in Section 2.2.6. To our knowledge our topology construction is the first topology that achieves optimality under the hose traffic model while supporting switches with heterogeneous rates and numbers of ports.

We then present an optimization framework, called LEGUP, to design network upgrades and expansions for existing datacenters. LEGUP finds networks that are realizable in a highly constrained datacenter. These networks maximize performance by implementing a heterogeneous Clos network from both existing and new switches. Supporting heterogeneous switches allows LEGUP to design upgrades with significantly more agility than existing techniques for the same dollar cost, which includes the cost of new switches and the cost of rewiring the network.

Our key contributions in this chapter are:

- Development of theory to construct optimal heterogeneous Clos topologies (§3.4).
- The LEGUP tool to design datacenter network upgrades and expansions (§3.3). LEGUP reuses

existing networking equipment when possible, minimizes rewiring costs, and selects the location of new equipment.

- We evaluate LEGUP by using it to find network upgrades for a 7,600 server datacenter based on the University of Waterloo's School of Computer Science datacenter (§3.6). LEGUP finds a network upgrade with nearly three times more bisection bandwidth for the same dollar cost as a fat-tree or traditional scale-up upgrades. LEGUP outperforms a fat-tree upgrade even when LEGUP spends half as much money. We also find that when adding servers to a datacenter in an iterative fashion, the network found by LEGUP has 265% more bisection bandwidth than a similarly upgraded fat-tree after the number of servers is doubled.

Before describing these results, we define the problem (§3.2). We end this chapter with a discussion of this work (§3.7).

3.2 Defining the Problem

Designing a datacenter network is a major undertaking. The solution space is enormous due to the huge number of variables, and an ideal network maximizes many objectives simultaneously. Our goal is to automate the task of designing the best network possible given a operator's budget and a model of their datacenter. Ideally, a user of our system need only hire staff to wire the DCN according to the system's output. For the remainder of this section, we describe the datacenter environment and state our assumptions.

3.2.1 Workload assumptions

Throughout this chapter, we assume that an ideal DCN has full agility, as described in Section 1.3. Recall that a network with full agility can feasibly route all hose TMs, which are denoted by $\mathcal{T}$.

Throughout this chapter and the next, we find it more convenient to deal with the ToR-to-ToR traffic matrix, which aggregates the servers connected to a ToR switch into a single entry. Throughout, we use symmetric ingress and egress rates, and we denote the ingress/egress rate of a ToR switch i by $r(i)$. This is called the *rate* of the switch.

Finally, we are primarily concerned with the *design* of high-performance networks, rather than the operation of these networks. Therefore, we assume that the routing and load-balancing schemes

can make full use of a network's available bandwidth, regardless of the TM. In practice, this assumption does not hold, since load-balancing mechanisms are not perfect; however, there are several proposals that provide nearly optimal DCN load-balancing. We defer a detailed discussion of these solutions until Section 3.7.

3.2.2 Switches, links and end-hosts

Most datacenters today run on commodity off the shelf (COTS) servers and switches, which reduces costs. There are many such switches to choose from, each with different features. We allow operators to define the specifications of switches available to add to their datacenter. These specifications must include the number of ports and their forwarding rates for each switch type. They can also include input details about line cards for modular switches. Operators can also specify a *processing delay* for each switch, which indicates the amount of time it takes the switch to forward a packet when it has no other load. This is used to estimate the end-to-end delay over a proposed path.

Links can be optical or copper and can use incompatible connector types. Currently, we do not model the difference in link medium or connectors; instead, we assume that all 1 Gb ports use the same connector and all 10 Gb ports use the same connector and that a 10 Gb port can also operate at 1 Gb. Copper links can pose a problem because they are limited to runs of less than 5–10m when operating at 10 Gbps [89]. It would not be difficult to check for link medium and connector types; however, we do not currently do so.

We assume that the datacenter operator has full control over end-hosts. That is, they can install custom multipath routing protocols, like Multipath TCP [103] and SPAIN [88], on all end-hosts. This assumption does not hold in cloud settings where customers can run their own OS installations. Here, the cloud provider could release a set of OS images that have the required end-host modifications installed. Or, the provider could integrate the changes into their hypervisor.

3.2.3 DCN performance: what's important to applications?

The two most important characteristics of DCN performance are high end-to-end bandwidth and low latency. Because of this, it is important to build a network with high agility, so that the network performs well for all TMs. High agility also keeps end-to-end latencies across the network low, because links will be lightly loaded, and hence queuing delays will be minimal. Minimizing end-to-end latency is important for interactive datacenter jobs, such as search and other partition-aggregate jobs,

where up to hundreds of worker servers perform jobs for a service. As previously described, minimizing latency is critical for this type of service because responses from workers that arrive after a deadline are not included in the final results.

3.2.4 Cost model

Estimating the cost of implementing a proposed DCN design is very challenging. There are two major obstacles to overcome: (1) determining prices of switches—street prices are often a fraction of retail prices and can be difficult to obtain. Further, vendors offer discounts for bulk purchases, so the price of a switch often decreases with volume. And, (2) it is difficult to estimate the cost of cabling a DCN. Bundling cables together in long runs also reduces the cost of wiring long-distance links [102, 89], though we are not aware of any algorithms to compute the cost of such a wiring. Additionally, it may be more expensive to wire irregular topologies compared to regular topologies; however, we do not have any data on this, so we assume per link cost, regardless of the topology.

For tractability, we assume fixed prices for switches. That is, the price of each switch is fixed and does not change based on volume. For cabling, we divide cable lengths into different categories (short, medium and long) and charge for a cable based on its length category. This assumptions can be lifted without any significant changes to our approach.

3.2.5 Placing equipment in a datacenter

A datacenter is a highly constrained environment in terms of equipment placement. We now describe the constraints that must be considered when adding equipment to a datacenter challenging.

First, to add equipment to a datacenter, there must be enough space to house it. Most equipment in the datacenter is positioned in large metal racks. A standard rack is 0.6 m wide, 2 m tall by 1 m deep and is partitioned into 42 rack units (denoted by U). A typical server occupies 1–2U. A small, ToR switch occupies 1U, and large, high-end switches can occupy up to 21U. Therefore, to add a switch to a datacenter, there must be enough free, contiguous rack units to hold it.

Datacenter equipment creates a significant amount of heat, which must be dissipated by a cooling system. Accurately modeling a datacenter cooling system is challenging [66]. Therefore, we assume a simple datacenter thermal model. In our model, the operator can please a thermal limit on each rack. This is an upper bound on the amount of heat that rack's contents may generate. As long as the

equipment in all racks does not generate heat beyond the given limits, then we assume the datacenter can be effectively cooled.

3.3 LEGUP Overview

LEGUP guides operators when upgrading or expanding their datacenter network. To achieve this goal, LEGUP solves a network design optimization problem that maximizes performance subject to a budget and the datacenter's physical constraints. We define DCN performance more precisely next (§3.3.1), and then give details about the inputs, constraints, and outputs of LEGUP (§3.3.2). We end this section by giving an overview of the optimization algorithm used by LEGUP (§3.3.3).

3.3.1 Optimization goals

LEGUP designs upgrades to maximize *network performance*, which we define it to be a weighted, linear combination of agility, flexibility, and reliability. Let the normalized values of these metrics be denoted p_a, p_f, and p_r for agility, flexibility, and reliability respectively. Then, our objective function is:

$$\text{maximize: } \alpha_a p_a + \alpha_f p_f + \alpha_r p_r$$

where α_a, α_f, and α_r are weighting constants. Precise definitions of each metric follows.

Agility measures a network's ability to handle any traffic matrix possible under the hose model, and is precisely defined in Section 1.3. Recall that a network's agility is the greatest constant p_a, such that all TMs in $p_a \cdot \mathcal{T}$ can be feasibly routed, where $\mathcal{T}$ is the set of hose TMs for the network.

Recall the example given earlier: consider a network consisting of two switches, each attached to 48 servers at 1 Gbps and a single 10 Gbps port that connects the switches. The agility of this network is $10/48$. And, if we have n servers attached to the first switch and m attached to the second, then we have the agility of the network is $10/\min\{n, m\}$.

Flexibility measures the number of servers that can be attached to the network without reducing agility beyond a given threshold. We say that a δ attachment point is an unused port such that attaching a 1 unit (in this paper, this is 1 Gbps) uplink device to this port does not decrease the network's agility to less than δ. Then, a network is (p_f, δ)-flexible if it has p_f distinct δ attachment points

when the attachment points are filled according to some rule (e.g., by greedily assigning devices to the attachment point that lowers agility the minimal amount). As an example, again consider our two switch network, except now assume all 48 of each switch's 1 Gbps ports are free. If we take $\delta = 0.5$, then the flexibility of this network is 68, achieved by attaching 48 servers to one switch and 20 to the other. If we attach an additional server to the second switch, then the agility drops to $10/\min\{21, 48\}$ which is less than 0.5.

Reliability is the number of link or switch failures needed to partition the ToR switches, which we denote by p_r. This model corresponds to the failure of a switch or port or a cable cut. As an example, the complete graph on n vertices has a reliability of $n - 1$ because every edge neighboring a vertex must be removed in order to partition the complete graph. The worst case reliability is that of a tree: removing a single node or edge partitions it.

We believe these metrics capture the most relevant features of the network for applications and operators. Applications are primarily interested in high-bandwidth and low-latency service. A network with high agility provides such service, because (1) it provides a large amount of end-to-end bandwidth in all possible network loads and (2) links in the network are likely to be lightly loaded most of the time, which keeps queueing delays short. A network with high flexibility has room to grow: more servers can be added without impacting application performance too much and more capacity can easily added by placing more links between existing switches. Finally, reliability is important in achieving high uptime, and so is of primary importance to operators due to the high cost of downtime.

These three metrics measure distinct aspects of a network. Agility and reliability are related—increased reliability can increase agility because of the additional end-to-end paths—however, two networks can have the same agility with completely different reliability metrics since link speeds can vary by orders of magnitude. Similarly, high agility is a prerequisite to high flexibility, but switches also must have unused ports for a network to be flexible.

To optimize network performance, we need to be able to compute each of these metrics. We have defined them so that they are computable in polynomial-time, and we will describe how to compute each later when describing LEGUP's details in Section 3.5.

3.3.2 Inputs, Constraints, and Outputs

As *input*, we require a budget, a list of available switch types and line cards, and a datacenter model. The budget is the maximum amount of a money that can be spent in the upgrade, and therefore acts as a constraint in the optimization procedure. The available switches are the details and prices of switches that can be purchased. Relevant details for a switch include its ports and their speeds, line card slots (if a modular switch), power consumption, rack units, and thermal output. Details of a line card are its ports, price, and a list of interoperable switches.

Providing a model of the existing datacenter is optional, and even when provided, can include little to complete detail about the datacenter. A complete model includes the full details of the network, the physical arrangement of racks, the contents of each rack, and the power and thermal characteristics of equipment in the racks. Additionally, thermal and power constraints can be included in this description, for example: the equipment in each rack cannot draw more than 10 kW of power. Details of the existing network includes information about its switches and their locations. Therefore, the per rack physical constrains that we model are thermal, power, and free rack units. If details of the existing switches are provided, they will be considered for use in the upgraded network. We have designed LEGUP to find a solution, if one exists, that meets the physical constraints given and will use cables in a cost-effective manner.

As *output*, LEGUP gives a detailed blueprint of the upgraded network. This includes the optimized topology and the new switches and line cards needed to realize the topology. If a datacenter model was included in the input, we also include in the output the rack where each aggregation switch should be placed.

3.3.3 The LEGUP optimization algorithm

We now give a high level overview of the algorithm employed by LEGUP. Recall that the optimization problem solved by LEGUP maximizes the sum of agility, reliability, and flexibility. That is, LEGUP's objective function is:

$$\text{maximize: } \alpha_a p_a + \alpha_f p_f + \alpha_r p_r$$

where p_a, p_f, and p_r is a network's agility, flexibility, and reliability respectively. This is a difficult optimization problem and is made harder by the large number of constraints.

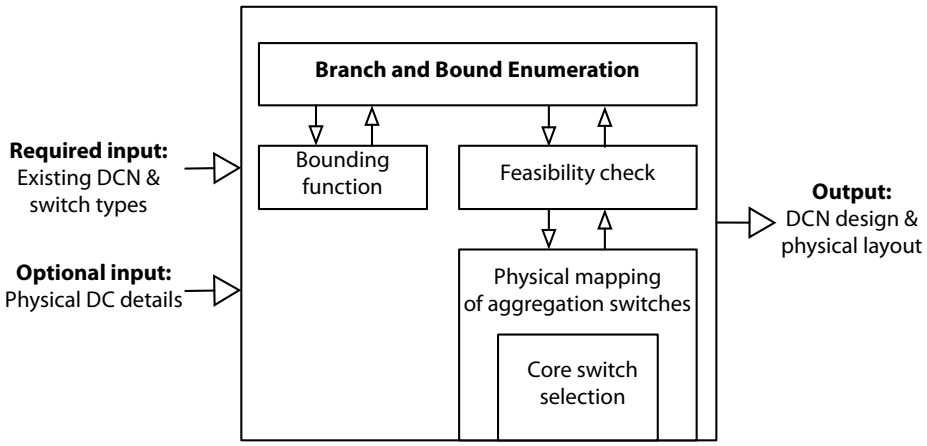


Figure 3.1: The LEGUP optimization algorithm.

We limit the search space by constraining LEGUP to only design tree-like networks. Such topologies are desirable in the datacenter regardless because many DCN load balancing, routing, and addressing solutions require a tree-like network, e.g., [90, 55, 10]. However, the theory of heterogeneous tree-like topologies has not been previously developed. Therefore, we develop the theory of heterogeneous Clos networks, which are tree-like networks, in the next section. The reasoning behind this decision is that that a traditional 1+1 redundant DCN topology is already a Clos network instance (albeit a 1+1 redundant topology is a Clos instance that does not have the agility and reliability typically associated with Clos networks). Despite adding heterogeneous switches, DCN addressing and routing solutions can be used on our topologies with no or minor modifications; we discuss this further in Section 3.7.

Our current implementation of LEGUP does not design the ToR switches. Instead, we assume that the set of ToRs and their hose rates are given as input. The algorithm then upgrades the aggregation and core levels of the network. Given a set of aggregation switches, the optimal set of core switches is restricted in a heterogeneous Clos network, so LEGUP explores the space of aggregation switches using the branch and bound optimization algorithm.

Branch and bound is a general optimization algorithm that finds an optimal solution by enumerating the problem space; however, it achieves efficiency by bounding, and therefore not enumerating, large portions of the problems space that cannot contain an optimal solution. Here, we define

the space of solutions to be the set of all possible arrangements of aggregate switches. This differs from a standard branch and bound implementation because we enumerate over only the aggregation switches, so we must introduce additional steps to find a set of core switches. Figure 3.1 depicts the process. Because of this modification, we cannot guarantee the optimality of solutions found by LEGUP.

In our context, the problem space is all possible sets of aggregation switches given the available switch types given as input. A *candidate solution*, denoted by S, is a set of aggregation switches. The space of all candidate solutions is called the *solution tree*. Each node in the solution tree is labeled by the set of aggregation switches it represents. The root's label is empty. A node has a child for each switch type. The label of a child node is the label of its parent plus the switch type the child represents. Or more precisely, let $S = \{s_1, \ldots, s_k\}$ be a candidate solution where $s_1, \ldots, s_k$ are aggregation switches. Then, the children of S in the solution tree are the candidate solutions with labels $\{S' | S' = S \cup \{s\},\ \forall s \in A\}$, where A is the set of switch types given as input. We say a solution is a *complete solution* when its aggregation switches have enough ports to connect the ToR switches with a spanning tree.

A complete solution only describes the set of aggregation switches in the network and does not account for the core switches nor the physical layout of the network. Given a complete solution, we find the min-cost placement of a solution's aggregation switches to racks (full details of LEGUP's handling of complete solutions are given later in §3.5) and then find the min-cost set of core switches to connect the aggregation switches to. Once this is complete, we add the cost of the core and physical mapping into the cost of the solution to determine if it is still feasible, that is, we check to make sure it is not over-budget. Additionally, we check to make sure no physical constraints (e.g., thermal and power draw) are violated in the physical mapping phase. Unlike standard branch and bound, we continue to branch complete solutions because a solution is complete here whenever it can connect all the ToR switches; however, adding more aggregation switches to a complete solution will always improve its performance (but may violate some constraints).

Before checking for feasibility; however, a candidate solution is bounded to check if it, or any of its children, can be an optimal solution. A candidate is bounded by finding the maximal agility, flexibility, and reliability possible for any solution in its subtree. A candidate solution with a lower performance bound than the optimal complete solution is trimmed, that is, it is not branched because its subtree cannot possibly contain an optimal solution. We delay the details of our particular bounding function until Section 3.5.1.

3.3.4 Why naive solutions aren't enough

To motivate our design of LEGUP, we briefly address the need for algorithms more sophisticated than standard heuristics. We identify three key weaknesses of existing heuristics that LEGUP addresses:

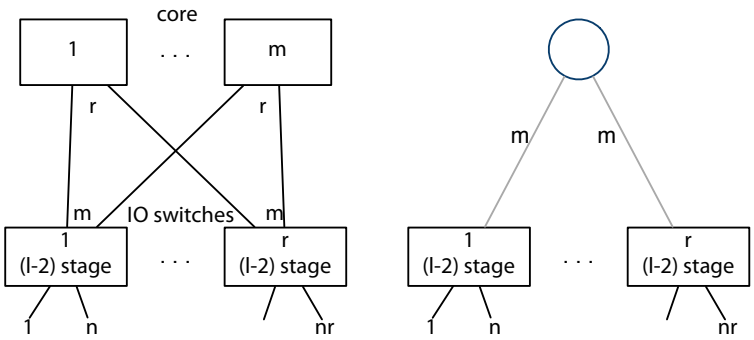
1. Simple heuristics typically don't take physical constraints into account, and therefore might not return a feasible solution. LEGUP finds a feasible solution if one exists.
2. Algorithms that greedily add switches with the minimum bandwidth to price ratio will always reuse existing switches. This might not be the optimal network configuration. LEGUP only reuses switches when it's beneficial to do so.
3. Cabling and switch costs need to be accounted for. We are unaware of any simple algorithms that take both these costs into account.

Our implementation of LEGUP's branch and bound algorithm uses depth-first search of the solution tree. When branching a solution tree node, we order the children so that they are sorted by bandwidth to price ratio. As a result, the first solutions explored by the branch and bound are the solutions that a greedy algorithm considers. We have found this to increase the number of trimmed subtrees dramatically since the first complete solutions tend to have good, though not optimal, performance.

3.4 Theory of Heterogeneous Clos Networks

We now describe the heterogeneous Clos topology, which generalizes the Clos topology [30] to support the use of heterogeneous link rates and differing numbers and rates of ports on switches. We show in this section that our topology can optimally (in terms of link capacity used) connect edge switches with heterogeneous ports and bandwidth demands.

Specifically, we show that our topology needs only as much link capacity as any other indirect network that can feasibly route all hose TMs. An *indirect network* is a network where nodes with positive rate (i.e., $r(i) > 0$) never directly connect to other nodes with positive rate, that is, nodes connect to each other indirectly through switches. A result due to Zhang-Shen and McKeown [127] states that an indirect network with node rates $r(1), \ldots, r(n)$ can feasibly route all hose TMs only if the sum of link capacities in the network is at least $\sum_{1 \leq i \leq n} 2\, r(i)$. We say that any topology that



(a) Physical realization (b) Logical topology

Figure 3.2: An l-stage Clos network. Each IO node here is a subnetwork with $l-2$ stages. In (b), each logical edge represents m physical links and the logical root represents m switches, each with r ports.

matches this lower bound is *optimal*. Our topology matches this bound without requiring homogenous switches and is the first to do so to our knowledge.

This result is developed in Section 3.4.2. First, however, we briefly review the standard, homogenous Clos network.

3.4.1 The Clos network

Recall that a 3-stage Clos network [30] is denoted by $C(n, m, r)$. Full details are given in Section 2.2.1. Throughout this chapter, we deal with folded Clos networks as shown in Figure 3.2(a). We call the switches in the middle stage the *core switches* and switches in the first/third stage as *IO switches*. We refer to the links from a stage to a higher stage as *uplinks* and the links from a stage to a lower stage as *downlinks*.

The recursive nature of Clos's topology (and our heterogeneous Clos topology) means that we only have to describe 3-stage networks, because an l-stage Clos network is recursively composed of 3-stage Clos networks. In an l-stage Clos network, each input and output switch is replaced by an $(l-2)$-stage network. An example is shown in Figure 3.2(a). As such, we always construct 3-stage networks in this section, but our results can be generalized to an l-stage Clos networks in a straightforward manner. (Indeed, we will use the generalization to 5-stage heterogeneous Clos networks

for LEGUP because we want it to design 3-level networks (i.e., networks with core, aggregation and edge layers).)

3.4.2 Constructing heterogeneous Clos networks

We now describe the heterogeneous Clos network construction. This construction relies on the notion of a *logical topology*, which encodes all relevant information of a network in a space-efficient representation. A logical topology can represent multiple network *physical realizations*—each of which is a unique way to physically build the network. Therefore, we separate logical topology design from the problem of finding a physical realization of a logical topology. First, however, we describe the logical topology of a traditional, homogenous Clos network.

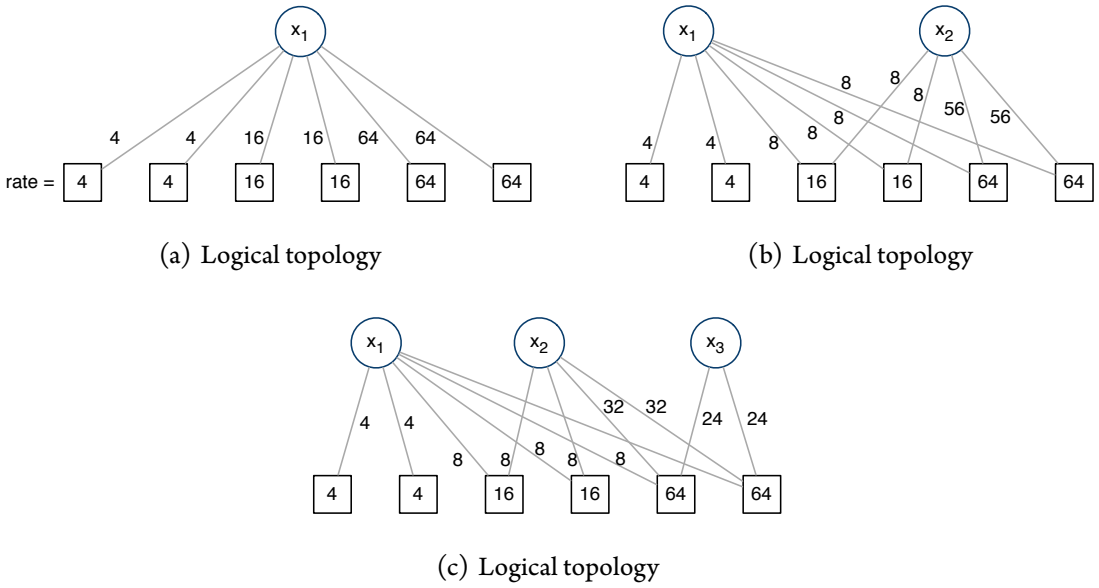
Throughout, we assume that the IO nodes are given and that our goal is to attach them to a set of core switches in an optimal configuration (i.e., the network uses the minimal link capacity necessary and sufficient to feasibly route all hose TMs). When focusing on logical topology design, we make the assumption that a logical node can be realized using the same amount of switching capacity as the logical topology. We then show how to lift this assumption when discussing the physical realizations of a logical topology.

A Clos network's logical topology

The logical topology of a Clos network is a compact representation of the network, as shown in Figure 3.2(b). For a Clos network $C(n, m, r)$, its logical topology T is the network where the core switches are collapsed into a single logical node, denoted by x. That is, x represents the aggregated switching capacity of the m core nodes. We have that T is a tree. Let x be the root of the tree, and x's children be the IO nodes $1, \ldots, r$. A logical edge between x and an IO node i has capacity equal to m, which is denoted by $c(x, i) = m$. Each IO node i has n children, which are servers in our scenario. Therefore, we assign a rate $r(i)$ to each IO node. These rates form the hose traffic model for this network, which are denoted by $\mathcal{T}(r(1), \ldots, r(n))$ or just $\mathcal{T}$ when the context is clear.

To have maximal agility (or equivalently in order to feasibly route all hose TMs), a Clos network must have $m = r(i)$ for all IO nodes i. The network uses $2nm$ link capacity. Note that this is 2 times nm, because we account of the bandwidth in each direction. As the sum of hose rate is $\sum_{i=1}^{n} m = nm$, we have that the Clos topology is optimal by the lower bound of Zhang-Shen and McKeown

[127]. We now show how to build optimal networks when presented with a heterogeneous set of IO nodes.



(a) Logical topology

(b) Logical topology

(c) Logical topology

Figure 3.3: Three optimal logical topologies for the given IO nodes. The numbers in the IO nodes indicate the rate of each node. We have shown one optimal edge capacity assignment for each topology; however, Figures (b) and (c) each have many optimal edge capacity assignments that are not shown.

Heterogeneous logical topology design

We now generalize the logical topology to support heterogeneous IO nodes. That is, we now assume that each IO node i has an arbitrary rate $r(i)$. We let each IO node i have a rate, denoted by $r(i)$, which is its hose rate (e.g., in a homogeneous network, the rate of each IO node is n because each inlet has a unit rate). Each logical edge (i, x) between an IO node i and logical root x has a capacity $c(i, x)$, which is the sum of physical link rates that (i, x) represents.

We assume that the logical topology of a heterogeneous Clos network forms a forest of trees. The leaves of these trees are IO nodes and each root node represents a set of core switches. We denote the IO nodes by $I = \{1, \ldots, k\}$ and the root nodes by $x_1, \ldots, x_l$. The logical topology design

problem is to find a suitable set of root nodes, the neighbors of each root node, and the capacity of the edges between IO nodes and root nodes. In the next section, we will show how to find the set of core switches each logical root node represents.

Our first result characterizes the number of trees that can be in an optimal logical forest for a heterogeneous Clos network with IO nodes I. The following lemma describes the number of allowed logical root nodes and their children given the the rates $r(1), \ldots, r(k)$ of the IO nodes.

Lemma 1. *Let T be a logical topology with IO nodes $I = \{1, \ldots, k\}$, and let $x_1, \ldots, x_l$ be the root nodes of T. Let X_p denote the set of IO nodes neighboring root node x_p such that $X_1 = I$ and $X_1 \supset \cdots \supset X_l$. Whenever all edges of T have positive capacity, we have that T feasibly routes all hose TMs optimally if, for all x_p, such that $2 \leq p \leq l$,*

$$r(i) > \sum_{j \in X_{p-1} - X_p} r(j) \text{ for all } i \in X_p \tag{3.1}$$

and $|X_l - X_{l-1}| \geq 2$.

Proof. Suppose there is some logical topology T that has a root node x such that there is a node $i \in X_{l'}$, where l' is a root node neighboring i with $c(i, x_{l'}) > 0$ and for which Equation 3.1 does not hold. Consider how much capacity the edges $(i, x_1), \ldots, (i, x_{l'-1})$ must have since T can serve all hose TMs: there must be at least $\min\{r(i), \sum_{j \in X_1 - X_{l'}} r(j)\}$ capacity to these nodes otherwise there is a hose TM that T cannot feasibly route. By assumption, $r(i) \leq \sum_{j \in X_1 - X_{l'}} r(j)$, so $r(i)$ is the minimal here. In a logical topology with optimal edge capacity, each IO node has at most $r(i)$ of uplink capacity. However, here, we have that i has $r(i) + c(i, x_{l'}) > r(i)$ uplink capacity, contradicting the optimality of T.

Suppose that $|X_l - X_{l-1}| = 1$. Here, T is non-optimal since the root node x_l has only a single neighbor, so it cannot route traffic to any other IO nodes. Therefore, it should not have positive capacity, since all traffic will need to be routed through $x_1, \ldots, x_{l-1}$ anyhow. □

Examples of the use of Lemma 1 are shown in Figure 3.3. It's important to note that this lemma identifies the available logical forests for a set I of IO nodes, but it does not determine the capacities of each logical edge, i.e., it determines the "shape" of the network, but not the capacity of the links. We will give a result that characterizes the set of optimal edge capacity assignments shortly.

First, however, we note that the following results are implied by this lemma:

- whenever $r(1) = \cdots = r(k)$, the optimal logical topology is a tree, that is, the logical topology has a single root node, and
- no matter the rates of each IO node, a logical topology that is a tree can be optimal, that is, a logical topology can always use fewer root nodes than it's allowed by Lemma 1 and be optimal.

Our next result shows how capacity can be optimally assigned to the logical edges to feasibly route all hose TMs. The intuition underlying this theorem is that the root x_p and its children (the IO nodes) form a disjoint spanning tree. We provision the spanning tree rooted at x_1 first, and then move to the next root node's spanning tree. Every unit of capacity that is provisioned to x_1 is a unit that does not have to be routed through $x_2, \ldots, x_l$, so we subtract the previously allocated capacity from the edges to $x_2, \ldots, x_l$.

Theorem 2. *Let T, $x_1, \ldots, x_l$, $X_1, \ldots, X_l$, and I be as in Lemma 1, and let $X_0 = \emptyset$ and $X_{l+1} = \emptyset$. We have that T can feasibly route all hose TMs using optimal capacity if and only if*

$$c(i, x_p) = \begin{cases} \sum_{j \in X_p - X_{p+1}} r(j) & \text{if } i \in X_{p+1}, \\ r(i) - \sum_{j \in I - X_p} r(j) & \text{otherwise} \end{cases}$$

for all $1 \leq p \leq l$ and all $i \in I$.

Proof. Suppose that T can feasibly route all hose TMs and that Equation 3.2 holds for all edges except (i, x_p). Let l' be the maximum root node such that $i \in X_{l'}$. Because T can feasibly route all hose TMs, we have:

$$\begin{aligned} \sum_{u \in [l']} c(i, x_u) &= \sum_{q \in [l'-1]} \sum_{j \in X_q - X_{q+1}} r(j) \\ &\quad + r(i) - \sum_{j \in X_1 - X_{l'}} r(j) && (3.2) \\ &= \sum_{j \in X_1 - X_{l'}} r(j) + r(i) - \sum_{j \in X_1 - X_{l'}} r(j) && (3.3) \\ &= r(i) && (3.4) \end{aligned}$$

So,

$$\sum_{q \in [l'-1]} \sum_{j \in X_q - X_{q+1}} r(j) = \sum_{j \in X_1 - X_{l'}} r(j) \qquad (3.5)$$

whenever T can feasibly route all hose TM with minimal edge capacity. However, here, we find a contradiction in both possible cases.
Whenever $i \in X_{p+1}$, so $c(i, x_p) < \sum_{j \in X_{p-1} - X_p} r(j)$, the left hand side of Equation 3.5 is less than the right hand side. And otherwise, $c(i, x_p) < r(i) - \sum_{j \in X_1 - X_p} r(j)$, in which case, we cannot make the reduction from Equation 3.3 to Equation 3.4.

To show sufficiency, suppose that Equation 3.2 holds for all edges of T. We construct a multipath routing that feasibly routes any hose TM D_{ij}. Let $i, j \in I$ be IO nodes such that $r(i) \leq r(j)$ and let l' be the max root node where $i, j \in X_{l'}$. When sending to j, let i split its traffic across root nodes $x_1, \ldots, x_{l'}$ such that $D_{ij}/c(i, x_p)$ traffic is routed through x_p, for $1 \leq p \leq l'$, and then x_p forwards this traffic to j on its single edge to j. For any hose TM D, the max traffic i can send is $r(i)$, so the max traffic i places on edge (i, x_p) is $r(i)/c(i, x_p)$. Since Equation 3.2 holds for all edges, we have that

$$\sum_{u \in [l']} c(i, x_u) = r(i)$$

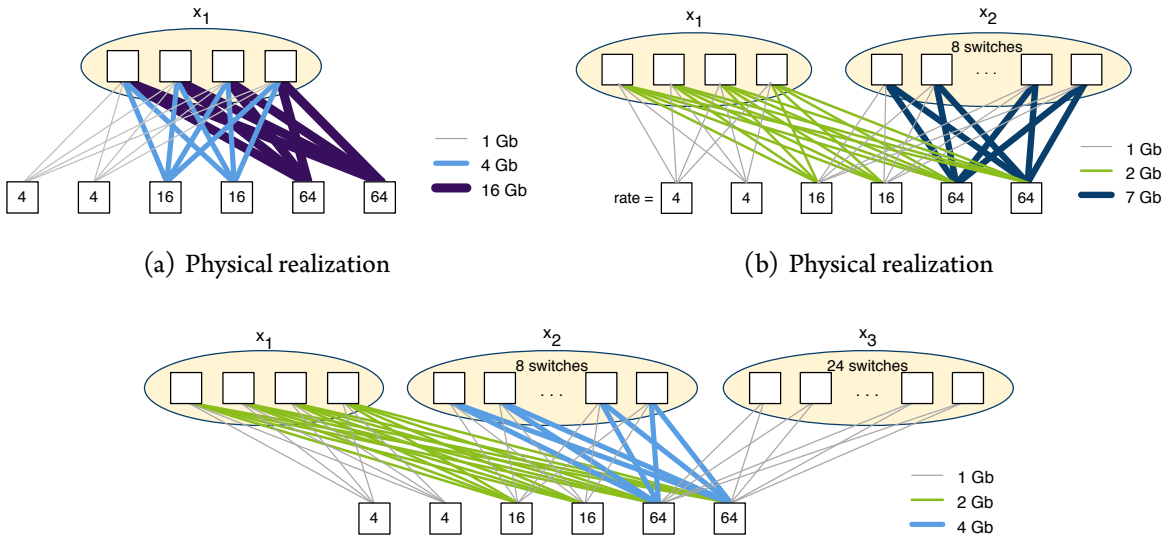
as established in Equations 3.2–3.4 above. Therefore, i can send traffic at rate up to $r(i)$ and never overload a link. Similarly, i cannot overload a link while receiving traffic, because it cannot receive more than $r(i)$ traffic at once. □

Examples of logical edge capacity assignments are shown in Figure 3.3. In this figure, we give a single assignment of edge capacities, however, for the topologies in Figures 3.3(b) and 3.3(c) there are many more optimal assignments. For example, in Figure 3.3(c), the two nodes with $r(i) = 64$ could have $c(i, x_3) = 23$ as long as $c(i, x_2) = 33$ and the logical topology would remain optimal. Following Theorem 2, we can find all other optimal capacity assignments. The theorem shows how to distribute the necessary link capacity across logical edges in such a way that all hose TMs can be feasibly routed.

Physically realizing a logical topology

We now show how to find a physical realization of a logical topology. Here, we are given a logical topology, and we want to find a set of switches that realizes each logical root node.

Each IO node has a set of uplink ports, which may have multiple speeds. To simplify our presentation, we separate IO nodes with multiple uplink port speeds into separate switches, so that each IO node has a single uplink port speed. This does not lead to a loss of generality because we can



(a) Physical realization

(b) Physical realization

(c) Physical realization

Figure 3.4: Examples of physical realizations of the logical topologies shown in Figure 3.3. Here, the thickness and color of each link indicates its capacity, which is shown in the legend next to each network.

recombine the separated switches later. So, each IO node i has a single uplink port speed, denoted by $p(i)$. We assume that an IO node i has at least $\lceil r(i)/p(i) \rceil$ ports; otherwise, no realization that can feasibly route all hose TMs exists because any IO node where this property does not hold will be a bottleneck.

Let x be a logical root node with a set $I(x)$ of IO nodes as its children. We use X to denote the set of switches that make up logical node x. Let $m(i) = \lceil c(i,x)/p(i) \rceil$, where $c(i,x)$ is the capacity of the logical edge (i,x) as before. Here, $m(i)$ is the number of physical uplinks i has to x. We use $P(r)$ to denote the set of all switches of I with $p(i) = r$, and $I(x)$ denotes the set of IO nodes neighboring root x.

Now, we determine how many switches comprise the logical root node X and how many ports each of these switch needs. Let $m_{\text{min}} = \min_{j \in I(x)}\{m(j)\}$. The core switches that realize x and the IO nodes $I(x)$ form a complete bipartite graph, so we have $|X| \leq m_{\text{min}}$. That is, X cannot contain more switches than m_{min}, because otherwise would not be able to form a complete bipartite

subgraph between x and $I(x)$. Each core switch in X must have at least $m_{\text{min}} \cdot |P(r)|$ ports with speed r, for each port speed r, and each $i \in I(x)$ has $\lceil m(i)/m_{\text{min}} \rceil$ uplinks to each switch in X. The following shows this topology is optimal.

Theorem 3. *A physical realization G constructed as described above of a logical tree T with root node x and IO nodes I with $c(i,x)$ minimized according to Theorem 2 can feasibly route all hose TMs.*

Further, if $c(i,x)$ and $m(i)$ are evenly divisible by $p(i)$ and m_{min} respectively for all $i \in I$, then the amount of link capacity used by this physical realization matches the lower bound on link capacity for any indirect network that can feasibly route all hose TMs.

Proof. To show that G can feasibly route all hose TMs, by Theorem 2, it's enough to show that there is a routing which distributes $r(i)/c(i,x)$ traffic over the physical links of the logical edges (i,x) and (x,i) without overloading any physical links. When i sends traffic to x, let each physical uplink carry $p(i)/c(i,x)$ fraction of the traffic, no matter the destination, and then the receiving core switch forwards the traffic to its destination. Then any traffic matrix can be handle as long as i never sends more than:

$$\begin{aligned} r(i) \cdot \sum_{v \in X} p(i)/c(i,x) \lceil m(i)/m_{\text{min}} \rceil &= \\ r(i) \cdot |X| \Big(\Big\lceil \frac{c(i,x)}{m(i)} \Big\rceil / c(i,x) \cdot \lceil m(i)/m_{\text{min}} \rceil \Big) &= \\ r(i) \cdot m_{\text{min}} \big(1/m_{\text{min}}\big) &= \\ = \quad r(i) & \end{aligned}$$

traffic, which i will never exceed in a hose TM. By a similar argument, there is enough link capacity from the physical switches in X to i.

An optimal construction has a total link capacity of $2 \sum_{i \in I} r(i)$. To see that the construction above matches this bound when $c(i,x)$ and $m(i)$ are evenly divisible by $p(i)$ and m_{min} respectively for all $i \in I$, consider the above equations. In this case, each $i \in I$ has $r(i)$ uplink capacity and $r(i)$ downlink capacity. Summing this over all switches in I shows our construction is optimal. □

In the proof of Theorem 3, we explicitly construct an optimal oblivious routing of all hose TMs for a heterogeneous Clos network. This implies that static routing can extract the full bisection bandwidth from a heterogeneous Clos network for any TM as long as our assumption of splittable flows holds. In practice, flows cannot be split, so we return to this in Section 3.7 when discussing operational issues.

3.5 LEGUP Details

We now describe the details of LEGUP's optimization algorithm. Recall that this algorithm solves a maximization problem by performing a branch and bound search of the space of possible aggregation switch sets. In this section, we focus on the handling of complete solutions, that is, the candidate solutions with enough aggregation ports to connect all ToR switches with at least a spanning tree.

Given a complete solution $S = \{s_1, \ldots, s_k\}$, where each s_i represents an aggregation switch, LEGUP does the following. (Each step is explained just below.)

1. Bounds the performance of S (§3.5.1).
2. If S's bound is lower than the best complete solution found so far, S is trimmed and it is not branched.
   Otherwise, the feasibility of S is determined by:
   - selecting a min-cost set of core switches (§3.5.2); and
   - finding a physical mapping of the aggregation switches to the datacenter's racks (§3.5.3).
3. If S is determined infeasible (due to a budget or physical model constraint violation), then it is trimmed.
   Otherwise, the performance of S is computed (§3.5.4), the best complete solution is updated, and S is branched.

Step 1 is used to trim the solution tree. It finds an upper bound of the performance of any solution in S's sub-tree (i.e., the solutions defined by $S \cup A$ for some set of switches A). If the upper bound for any solution in S's sub-tree is less than the performance of the current best complete solution, then we determine the optimal solution is not in S's sub-tree and so we trim the search tree of this sub-tree (as shown in Step 2).

Because a complete solution in our case is a set of aggregation switches, we need to find a set of core switches (assuming we are designing 3-stage networks, which is in the case in this chapter). Additionally, we need to find a placement of the aggregation switches on the datacenter floor. Both a min-cost set of core switches and a placement of aggregation switches are found in Step 2.

Finally, if S is feasible, then it needs to be branched. That is, we need to compute its performance, and then visit its children in the search tree, that is, S needs to be *branched*. This is done in Step 3.

We use α_a, α_f, and α_r to denote the weights of performance metrics agility, flexibility, and reliability respectively. Then, the overall performance of a solution S is $p(S) = p_a\alpha_a + p_f\alpha_f + p_r\alpha_r$

where p_a, p_f, and p_r have been normalized by their maximal values. We show how to find these maximal values in (§3.5.4). Throughout, whenever we use one of these values, we assume it has been normalized.

3.5.1 Bounding a candidate solution

Our bounding function estimates each performance metric individually and then returns the weighted sum of the estimates. Because it is used to trim solutions and we are maximizing performance, it must overestimate the best possible solution in the candidate solution's subtree. Given a complete solution S, we denote the bounds of S for agility, flexibility, and reliability by b_a, b_f, and b_r respectively. Given a candidate solution S, we bound each metric of S as follows.

Agility and flexibility are coupled, so we bound them simultaneously, i.e., we bound $\alpha_a b_a + \alpha_f b_f$. We begin by finding the maximal agility the remaining budget allows. That is, let b_a^{max} denote the maximal agility any network constructed with S and the remaining budget can have. We find b_a^{max} by first greedily adding the switch with the highest sum of port speeds to cost ratio of all the available switch types to S until the cost of S is over-budget (note that this procedure always makes use of any existing switches that are not included in S as they have no cost). Because b_a and b_f can overestimate the performance of any solution that is a descendant of S in the search tree, we do not worry about actually being able to realize the topology, so we aggregate the bandwidth of switches in S, and we use $r(S)$ to denote their aggregate bandwidth, i.e., the sum of their port speeds.

We combine each level of switches into single logical nodes, that is, we create a logical topology with a single core node, single aggregation node, and single ToR node, which form a path ToR to aggregation to core. We find b_a^{max} by determining the maximum possible agility of this logical topology. Let $r(\text{ToR})$, $r(\text{aggr})$ and $r(\text{core})$ be the sum of port speeds of the logical aggregation and core nodes respectively. From Theorem 2, to maximizes b_a^{max}, we have that $r(\text{aggr}) = 2/3\, r(S)$ and $r(\text{core}) = 1/3\, r(S)$. Moreover, we have

$$b_a = \min \left\{ \frac{r(\text{core})}{r(\text{ToR})}, \frac{1/2\, r(\text{aggr})}{r(\text{ToR})} \right\}.$$

We now find a lower bound on b_f, which we denote by $b_f^{\min}$. We have $b_f^{\min} = 1/2 r(\text{aggr}) - r(\text{ToR})$. That is, $b_f^{\min}$ is equal to the amount of spare bandwidth the aggregation and core nodes can handle without decreasing agility.

Algorithm 1 Bound agility and flexibility.

Input: $r(\text{core}), r(\text{aggr}), r(\text{ToR}), b_a^{\max}$, and $b_f^{\min}$
Output: b_a, b_f

begin
$b_a = b_a^{\max}$
$b_f = b_f^{\min}$
$r(cToR) = 0$
until the following does not increase $\alpha_a b_a + \alpha_f b_f$ **do**
    if $r(\text{cToR}) < r(\text{ToR})$ then
        $r(\text{core}) = r(\text{core}) + 1$
        $r(\text{aggr}) = r(\text{aggr}) - 2$
        $r(\text{cToR}) = r(\text{cToR}) + 1$
    else
        $r(\text{aggr}) = r(\text{aggr}) - 1$
        $r(\text{ToR}) = r(\text{ToR}) + 1$
    $b_f = b_f + 1$
    $b_a = \min\{1, \frac{r(\text{core})}{r(\text{ToR})}, \frac{1/2 r(\text{aggr})}{r(\text{ToR})}\}$
end

Because we are jointly optimizing agility and flexibility, we need to maximize their weighted sum, i.e., we should maximize $\alpha_a b_a + \alpha_f b_f$, so we need a procedure to maximize this sum. This procedure is shown in Algorithm 1. There, $r(\text{cToR})$ is the rate of devices attached to the core node. The algorithm attaches 1 unit of capacity at a time to the best location possible. If $r(\text{cToR}) < r(\text{ToR})$ the best location to attach a device is the core because doing so decreases agility less than attaching the device to the aggregation node. We repeat this process until $\alpha_a b_a + \alpha_f b_f$ hits a maximal point, which is guaranteed to be globally optimal because it is the sum of two linear functions (i.e., for a fixed S and budget, the combination of $\alpha_a b_a$ and $\alpha_f b_f$ does not have any local maximal points).

Reliability is bounded by two observations that upper bound S's reliability. We have that b_r is at most:

- 1/2 the number of ports on any $s \in S$; and

- the number of open ports (i.e., the ports not connected to a link) on any ToR switch.

We therefore set b_r to the maximum of these two values.

3.5.2 Finding a set of core switches

To find the min-cost core switches, we need to solve two sub-problems: finding an optimal logical topology (§3.5.2), and then finding the min-cost switches that physically realize that topology (§3.5.2).

Selecting a logical topology

We now show how to select a logical topology. Using our terminology from Section 3.4, the aggregation switches are the IO nodes. Recall that there are multiple optimal logical topologies that connect a set of IO nodes, so we need to use the one with a maximal performance to cost ratio. We observe, however, that a logical topology with k logical core nodes can always be made to have $k-1$ logical core nodes by *stacking* switches, that is, by combining multiple switches with l ports in total into a single switch with at least l ports. Moreover, if no physical realization of a logical topology with k core nodes exists, then there is no physical realization of a logical topology with $k-1$ core nodes. Therefore, we always maximize the number of logical core nodes in accordance with Lemma 1. We set the capacities of each logical edge such that they are minimized according to Theorem 2.

It is important to note that we do not require switches to be stacked, but merely allow it. It can reduce costs (e.g., a 48 port switch generally costs less than two 24 port switches), but in some cases higher radix switches cost more than several lower radix switches, so we only stack core switches when it makes sense from a cost perspective as we show in the next section.

Realizing the logical topology

Once we have a logical topology for a candidate solution S, we need to realize each logical node. The first issue is to determine the ports each aggregation switches should use to connect to ToR switches and what ones should connect to core switches. Here, $1/2$ of the switch's ports should be used as uplinks and the other half should be used as downlinks. We find aggregation switch down ports (i.e., the ports that connect to ToR switches) by iterating through the ToR switches. At each ToR switch, we select one of its free ports to use as an uplink by selecting its free port with the highest

speed such that there is a switch in S with an open port at the same speed or greater. When multiple such switches in S exist, we connect this ToR switch to the $s \in S$ with the most free capacity. We repeat this procedure until either $1/2$ the capacity of each switch in S has been assigned to a ToR switch or until the uplink rate of each ToR switch is equal to its hose traffic rate.

By Theorem 3, the aggregation switches and logical topology dictate the number of core switches and the number and speeds of ports for each core switch. A core candidate solution is therefore infeasible if one of the logical nodes cannot be realized because no switch has enough ports of each rate required (e.g., the aggregation switches may dictate that each core switch has 145 10 Gbps ports when the largest available switch has only 144 such ports)[1]. Assuming that realizing the logical topology F is feasible, let $x_1, \ldots, x_l$ be F's logical root nodes. The switches that realize each x_i are dictated by X_i, the aggregation switches that are x_i's children, so we realize each x_i with the min-cost switch that satisfies its port requirements. This switch assignment is easily found by comparing each x_i's requirements to the available switch types.

We can, however, potentially lower the cost of the core switches by stacking several switches into one physical switch, e.g., if x_i needs to be realized by five 24-port switches, it can also be realized by a single 120-port switch, potentially at a lower cost. This switch stacking problem can be reduced to a generalized cost, variable-sized bin packing problem, which can be approximated by an asymptotic polynomial-time approximation scheme [44]; however, their algorithm is complicated and still too slow for our purposes since it must be executed for every complete solution. Instead, we use the well-known best-fit heuristic [75] to stack core switches, which is known to perform well in practice.

Two issues arise when we stack core switches. First, it is possible to turn a feasible solution infeasible, e.g., after stacking switches, the resulting solution may violate a physical constraint, such as there may not be a rack that has enough free slots for the larger switch. Second, stacking core switches can decrease our reliability metric. Therefore, we save the original set of core switches. If either of these cases occurs, we revert back to the original set of core switches, and then continue.

[1] It is possible to create higher port-count switches by embedding an additional 3-stage Clos network that acts as a single switch—as an example, six 144-port switches can be arranged in a 3-stage Clos network to act as a 288-port switch. We do not support such configurations because it (1) complicates the algorithm, (2) can result it a network where the paths between two ToR switches have different lengths and (3) seems unlikely to achieve any cost reduction.

Algorithm 2 Mapping aggregation switches to racks.

Preprocessing
Input: datacenter model
Output: lists of racks

begin
for each rack **do**
    find the sizes of its contiguous free rack units, and
    the distance to the k nearest ToR switches

Separate the racks into lists $R[u]$ such that the largest
    contiguous free rack units of racks in $R[u]$ is u

Sort each list in increasing order of distance to k ToR switches
end

Mapping
Input: datacenter model M and S
Output: map $S \rightarrow$ racks

begin
// Phase I
for each switch $x \in S$ **do**
    for each aggregation switch $y \in M$ **do**
        find the closeness of x and y
$S' = \emptyset$
for $y \in M$ **do**
    Map the closest $x \in S$ to y
    $S' = S' \cup \{x\}$
// Phase II
for each switch $x \in S - S'$ **do**
    Map x to the first rack in $r \in R[x.U]$ such that no per rack constraints are violated
    Update r's largest contiguous rack units, and move it to
        the appropriate list
end

3.5.3 Mapping aggregation switches to racks and ToR switches

Now that we have determined the set of switches that comprise the aggregation and core levels, we need to place them into racks and connect each ToR switch to aggregation switches. We say that a placement of each aggregation and core switch in a rack is a *mapping* of these switches. We assume that the core switches can be centrally located and that ToR switches are already placed, so we are only concerned with aggregation switches in this section.

Our mapping algorithm takes as input a set of aggregation switches, here this is S, and the datacenter model. If no model is given, then this stage is skipped. If a network blueprint is given but no datacenter model, then the mapping assigns each link a unit cost if it is new or modified. The mapping's goal is to minimize the cost of the physical layout of these aggregation switches subject to the rack, thermal, and power constraints of the datacenter model; here, cost is the length of cables needed to connect the ToR switches to aggregation switches. Even using an Euclidean geometry setting and without our additional constraints, this problem is NP-hard as it can be reduced to a Steiner forest problem variant, see, for example [70]. Additional complications here are that the datacenter model may already have aggregation switches in place and we would like to use the Manhattan distance instead of the Euclidean.

We use a two-phase best-fit heuristic for mapping. The first phase matches aggregation switches to existing switches in the datacenter model, and the second stage finds a best-fit for all aggregation switches not placed in the first phase. To speed up the algorithm, we do some preprocessing. The preprocessing and mapping algorithm details are given in Algorithm 2.

Phase I of our mapping algorithm attempts to replace existing aggregation switches in the datacenter model with a close switch in S. We define closeness as follows for two switches s_1 and s_2. We have $closeness(s_1, s_2) = 0$ if s_1 does not have as many ports as s_2 for any speed, when ports are allowed to operate at any speed less than their line speed, and $closeness(s_1, s_2) = 1$ if s_1 has at least as many ports as s_2 for all speeds, again allowing s_1's ports to operate at less than their max speed (e.g., the closeness of a 24-port 10 Gbps switch and a 24-port 1 Gbps switch is 1).

3.5.4 Computing the performance of a solution

We now address how to compute each of our performance metrics.

Agility can be computed in $O(n)$ time, where n is the number of ToR switches in the input network. Because we have constructed the network in accordance with Lemma 1, a node i with rate

$r(i)$ must have at least $r(i)$ of uplink bandwidth to feasibly route all hose TMs (i.e., for agility to be 1). Specifically, if the uplink bandwidth of all i's uplink ports sums to u, then we have that the network's agility is at most $u/r(i)$. We can therefore determine the upper bound on agility imposed at each ToR and aggregation switch to find the network's agility. Note, however, that this method to compute agility does not work unless the network's logical topology follows Lemma 1. For general networks, a linear programming procedure can compute agility as we show in the next chapter.

In general, reliability can be determined using a standard min-cut algorithm. A heterogeneous Clos network's reliability is bounded by the number of uplinks from a ToR to its aggregation switches and an aggregation switches to its core switches as observed earlier, so we can compute it in linear time, which is faster than any algorithms we are aware of to compute the min-cut of an arbitrary network.

Computing flexibility depends on the rule specified for attaching new devices to the network. In our implementation, we greedily attach devices to the open port that reduces agility the least. Computing flexibility is done by repeating this process until no more unit bandwidth devices can be attached without reducing agility below δ.

Finding the maximal value of each metric: We need to scale each of our performance metrics to a $[0,1]$ range to compare them. We limit the agility of any network to 1[2]. We normalize flexibility and reliability by finding the maximal value of each metric given the budget and using this for normalization. These upper bounds are found applying our bounding function (§3.5.1) on an empty candidate solution with same budget and set of switch types. This upper bounds the agility, flexibility, and reliability of any network that can be constructed given the operator-specified budget and cost model, and hence gives us maximal values of each metric.

3.6 Evaluation

We now evaluate LEGUP by designing networks under several scenarios. We compare it to other methods of constructing datacenter networks. We describe the datacenter used for evaluation first (§3.6.1), and then describe alternative upgrade approaches (§3.6.2). Finally, we study the perfor-

[2]If the agility of a network is greater than 1, then that network can feasibly route each of its hose TMs with a maximum link utilization of less than 1.

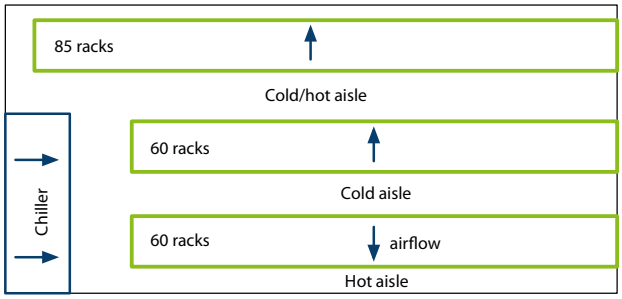


Figure 3.5: Layout of the SCS datacenter. Arrows show the direction of airflow

mance of these approaches with two scenarios: upgrading our datacenter (§3.6.3) and expanding it (§3.6.4).

3.6.1 Input

Datacenter model: To test LEGUP on an existing datacenter, we have modeled the University of Waterloo's School of Computer Science (SCS) datacenter. The servers in this room run services such as web, email, and backup servers, and many are used as compute machines by faculty and students. To make the upgrade problem more like that in a larger datacenter, we have increased the number of racks in the datacenter by a factor of ten and assume that each rack is full. We scaled the network proportionally, keeping the characteristics of the network invariant (such as connectivity between levels of the network's tree-like topology). Our analysis of the SCS datacenter is based on this scaled version. We note that the SCS datacenter network is sufficient for the needs of the school currently, and there are no plans to upgrade it. Despite this, we believe our model is interesting because it is loosely based on a real-world datacenter with real constraints.

The scaled-up SCS datacenter has three rows. These consist of 205 racks, which we assume house a total of 7600 servers, 190 ToR switches, six aggregation switches, and two core routers. These racks are arranged into three rows. Row 1 has 85 racks and rows 2 & 3 each have 60 racks. The rows are arranged as shown in Figure 3.5.

The SCS datacenter has grown organically over time and has never had a clean slate overhaul. As a result, the SCS datacenter is a typical small datacenter with problems such as the following:

ToR switches

Hose uplink rate	Uplinks (1, 10 Gbps)	No. switches
28	8, 2	50
40	8, 4	80
8	8, 0	40
2	2, 0	20

Aggregation switches

Line cards	Line card slots	No. switches
3x 24 1 Gbps, 1x 2 10 Gbps	6	1
4x 4 10 Gbps	6	9

Table 3.1: Existing switches in the scaled-up SCS datacenter model.

Heterogeneous ToR and aggregation switches: Switches have a long lifespan in the datacenter, so the ToR switches are not uniform. Aggregation switches are all HP 5400 series switches, though they do not have identical line cards. We list the details of the datacenter's existing switches in Table 3.1.

Poor air handling: The datacenter has a single chiller and it's located at the end of the rows. Additionally, the hot and cold aisles are not isolated, resulting in less effective cooling. Because of this, hot-running equipment cannot be concentrated at the far end of the rows where it will not receive much cool air from the chiller. We model this by linearly decreasing the allowed amount of heat generated per rack as the racks move away from the chiller. We do not have thermal measurements for all our input switches, so we approximate the thermal output of a switch by its power consumption. Therefore, row 1 (the row with 85 racks) can support up to 18 kW of equipment and the last rack in this row can support only 12 kW; the i^{th} rack in this row can then support equipment drawing $12 + 6/i$ kW of power. The first rack in the other two rows can support up to 22 kW of equipment and the last rack on these row supports up to 12 kW of equipment. The thermal constraint of racks between is again linearly scaled.

The datacenter's current network is arranged as a tree; each ToR switch has a single uplink to an aggregation switch and each aggregation switch has two uplinks to the core routers. We would like to modify the network so that only outbound traffic passes through the core routers. Therefore, all network upgrades must be three-levels, that is, they need to replace these routers with core switches.

Switch model	Ports	Watts	Price ($)
Generic	24 1 Gbps	100	250
	48 1 Gbps	150	1,500
	48 1 Gbps, 4 10 Gbps	235	5,000
	24 10 Gbps	300	6,000
	48 10 Gbps	600	10,000
	144 10 Gbps	5000	75,000
HP 5406zl chassis	n/a	166	2,299
HP line card	24 1 Gbps	160	2,669
HP line card	4 10 Gbps	48	3,700

Table 3.2: Switches used as input in our evaluation. Prices are street and power draw estimates are based on a typical switch of the type for the generic models or manufacturers estimates, except for the HP 5400 line cards, which are estimates based on the watts used per port on the other switches.

Switch and cabling prices: The switches available for use by the upgrade approaches are shown in Table 3.2. Unless otherwise mentioned, we assume that installing or moving links to or from an aggregation switch costs $50 and that links from ToR switches to servers are free to move. Based on our discussions with the datacenter operators, we believe this is a conservative estimate based on link prices and the man-hours needed to install a cable in an existing datacenter. Though LEGUP supports charging for a cable based on its length, we do not use this functionality because we are unable to estimate the lengths of cables used by the fat-tree upgrade approach.

3.6.2 Alternative upgrade approaches

To evaluate the solutions found by LEGUP, we consider two alternative network upgrade approaches. The first method, is the traditional scale-up method. This approach models the method our datacenter operators currently use to upgrade the network. They upgrade line cards in our modular switches as their budget allows by purchasing the line card with the least cost to rate ratio. As they run out of line card slots in the switches, they purchase more of the same switches, and fill them with additional line cards. In our upgrade and expansion scenarios, we want the DCN to have three levels of

switches, so we need to add core switches to our network. To do this, we use the 24-port 10 Gbps switches, and only use 10 Gbps links between aggregation switches and the core.

The second approach we consider is to build a generalized fat-tree using 1 (or occasionally 10) Gbps links following the DCN architecture of Al-Fares et al. [10] and VL2 [55]. Here, we reuse existing ToR switches. We do not explicitly build the fat-tree. Instead, we bound the maximal agility possible by a 3-stage fat-tree given the cost model and budget. This means that we are comparing LEGUP against an optimistic assessment of a fat-tree's performance.

For both these approaches we do not take the physical constraints of the datacenter into account. Therefore, it may not always be possible to construct the networks found this way. In contrast, LEGUP takes the physical constraints (in our case thermal and rack space) into account, and so it is at a disadvantage.

3.6.3 Upgrading the datacenter

We first consider upgrading the SCS datacenter to maximize its performance. For this scenario, we set the weights of each performance metric to be 1 and $\delta = 0.1$. We selected this value of δ because all methods design an upgrade with agility at least 0.1 for all budgets considered. Since we have that δ is less than the agility, it is possible for all methods to design networks with some $p_f > 0$.

The performance achieved by our three upgrade approaches for various budgets is shown in Figure 3.6. As the chart shows, for all budgets, LEGUP finds an upgrade with higher agility and flexibility than the the scale-up or fat-tree approaches. Moreover, LEGUP always finds a network upgrade with more agility and flexibility than the other two approaches even when LEGUP's budget is half as much as their budgets. Because the maximal reliability is two (as limited by the ToR switches with only two uplink ports), all upgrades were able to achieve this for all budgets.

Interestingly, the scale-up approach often outperforms the fat-tree. This is largely due to the high number of cables in the fat-tree, each of which costs $\$50$ to install here. For example, with a budget of $\$100$K, the fat-tree approach can only spend roughly $\$30{,}000$ on switches because $\$70{,}000$ is needed for cabling. By taking advantage of 10 Gbps links, LEGUP and the scale-up approach need an order of magnitude fewer cables, and both approaches reduce cabling costs further by attempting to leave existing switches connected to the same ToR switches.

To investigate the impact of link costs, we performed a sensitivity analysis. We repeated the above experiment as we varied the cost to install a link. We fixed the prices of switches to be the same as

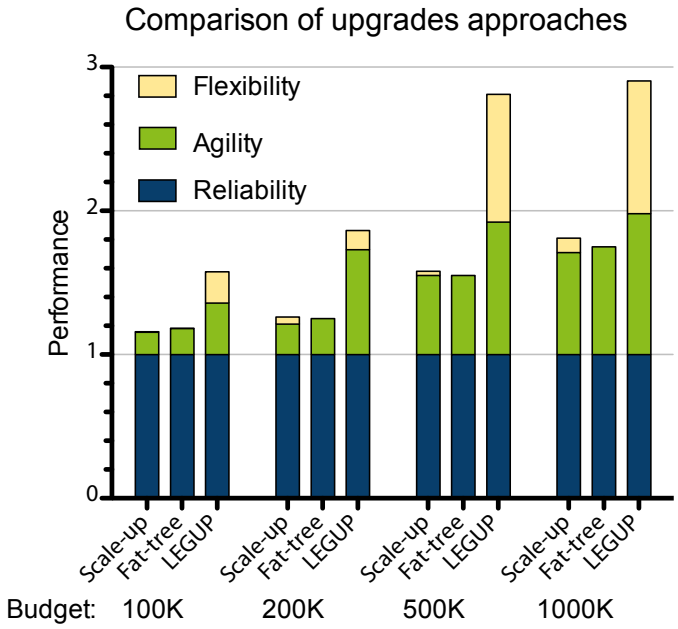


Figure 3.6: Performance of the upgrade approaches for various budgets. Here, we have $\alpha_a = \alpha_f = \alpha_r = 1$ and $\delta = 0.10$.

before and set the budget to $\$200$K. We compare LEGUP to a fat-tree constructed with 1 Gb or 10 Gb links now because of our observation that link costs can be a majority of a network's cost. Again, we set $\alpha_a = \alpha_f = \alpha_r = 1.0$ for this scenario.

The results of this sensitivity analysis are shown in Figure 3.7. As expected, the agility for each approach decreases as link costs increase. However, the performance of the solutions found by LEGUP can *increase* as link costs increase, because our flexibility metric does not depend on the cost of links. This is because of our normalization procedure for flexibility. The normalized flexibility metric is relative based on the inputs, and so its value depends on the cost of links. Therefore, we cannot accurately compare the flexibility of networks found by LEGUP when links cost $\$100$ with networks found with link costs of $\$50$. We can only compare the flexibility of two networks designed under the same cost model. We view this artifact as a weakness in our definition of flexibility and we plan to address it in future work.

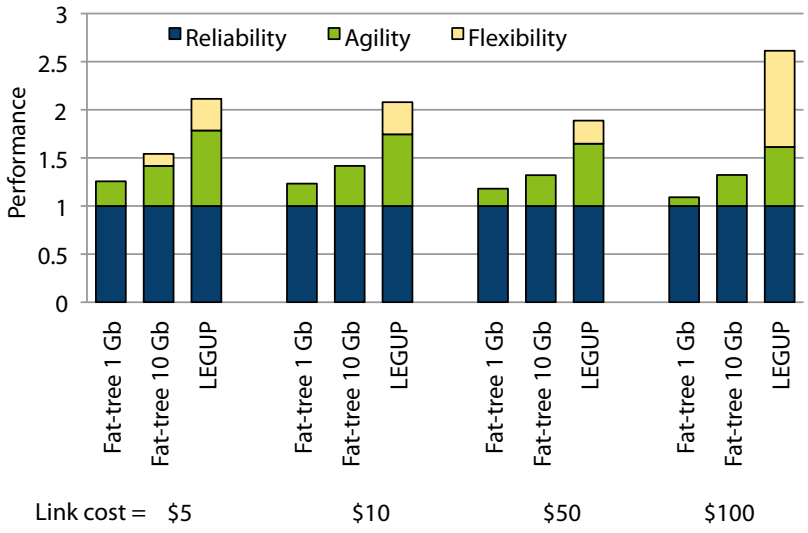


Figure 3.7: Performance of a fat-tree built with 1 Gbps or 10 Gbps links compared to LEGUP with a budget of $200K and various link costs. Throughout, the prices of switches are fixed, and the cost to install a link is varied from 5–100 dollars.

We see that LEGUP significantly outperforms the fat-tree networks under all link costs. Overall, LEGUP's networks have 52–55% more agility than the 10 Gb fat-tree, and the 10 Gb fat-tree performs better than the 1 Gb fat-tree, even when links are very inexpensive. The 10 fat-tree has 350% more agility than the 1 Gb fat-tree when links cost $100 and 60% more agility when links cost $5. We found that the 1 Gb fat-tree used 10–68% of its budget on links and that the 10 Gb fat-tree used 1.5–24% of its budget on links.

3.6.4 Expanding the datacenter

We now consider expanding a datacenter network to accommodate additional servers as they are added over time. Again, we use the SCS datacenter as a starting point, and we add 1200 servers to it at a time and find a network for the expanded datacenter. Each expansion has a budget of $300,000, and uses the network found in the previous iteration as input. This budget was selected because it is 10% of the cost of the servers, assuming a price of $2500 per server; this cost is in line with recent cost breakdowns for servers compared to the network [66, 57]. We do not take the racks' thermal

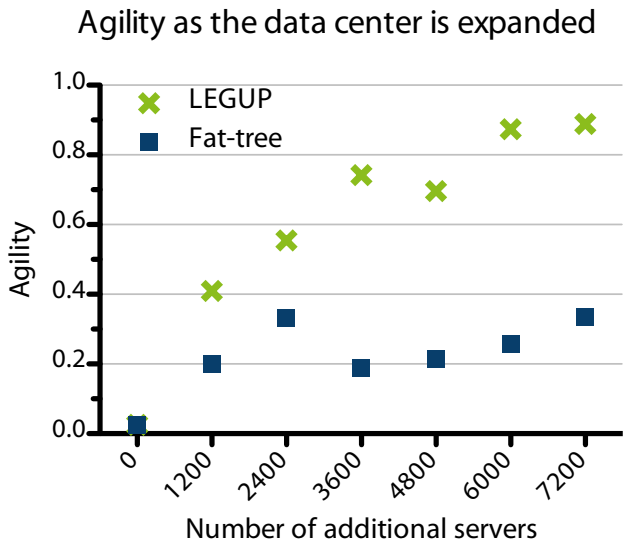


Figure 3.8: Agility as additional racks of servers are added to the datacenter. Each point is found by increasing agility as much as possible given a budget of $\$300{,}000$ and the previous iteration as the existing network.

or constraints into account here because, in reality, the SCS datacenter floor does not have enough room for 1200 more servers. For LEGUP, we set $\alpha_a = 1, \alpha_f = 5, \alpha_r = 1$ and $\delta = 0.10$. We arrived at these settings through experimentation. We found that when we set α_f to less than 2, then agility of the later iterations was lower than when $\alpha_f = 5$ because the solutions found by LEGUP in this case did not plan for growth and thus did not do as well after a few expansion iterations. Because LEGUP assumes that servers connect to a ToR switches, we use 30 switches with 48 1 Gbps and 4 10 Gbps ports as ToR switches for each 1200 server expansion. Doing so uses $\$150{,}000$ of LEGUP's budget each iteration.

The results of our expansion scenario are shown in Figure 3.8. LEGUP significantly outperforms the fat-tree upgrades. The fat-tree approach experiences a drop in agility when the network with 2400 additional servers is expanded by another 1200 servers because the aggregation and core switches of the +2400 server network are all 24-port switches. To accommodate the additional 1200 servers without lowering agility even further, its core switches need to be replaced by 48-port switches. After

this change the amount of agility gained with each addition is less than previously because the 48-port switches are not as good a value as the 24-port switches. LEGUP experiences a similar drop in agility; however, the effect is less pronounced.

3.7 Discussion

Lacking a theoretical foundation to model and analyze heterogeneous tree-like topologies, a datacenter manager has two options to upgrade their network: (1) perform an expensive forklift upgrade, or (2) add additional switches to their network using best practices or other rules of thumb. This second approach would likely either result in a topology with sub-optimal agility for the money because link capacity would not be able to be used optimally. So, even without LEGUP, our theory of heterogeneous Clos networks is useful because it describes topologies that can extract maximal agility from available link capacity, which is useful to guide the addition of switches. Moreover, LEGUP can be used to optimize even homogeneous networks by finding a good rack slots to place new switches.

So far, we have not addressed operational issues that arise when heterogeneity is added to a DCN. We address them now:

Configuration: We have not accounted for the cost of reconfiguring a DCN after modifying its topology. Reconfiguration could be expensive and error-prone, especially if it is performed manually. We expect that this will become less of a issue as datacenter management solutions improve. For instance, PortLand [90] provides "plug-and-play" functionality for DCN switches and NOX can be used to centrally manage a DCN [114]. Both of these solutions can support heterogeneous Clos topologies with minor modifications.

Routing and load balancing: In Section 3.4, we assumed ideal load balancing. This is not achievable in practice because it requires support for splitting individual flows across multiple paths. Nevertheless, close to optimal load balancing on our topologies can be achieved, however. For example, Mudigonda et al.'s SPAIN [88] performs multipath load balancing on arbitrary topologies. Based on their results, we believe SPAIN can extract close to the full bisection bandwidth from our topologies. A second approach is to use *oblivious routing*, where the path for an i-j packet is randomly selected from a probability distribution over all i-j paths. Oblivious routing has been shown to perform well

on Clos networks [55], and it can achieve optimal load balancing on our topologies as well (as implied by our results in Section 3.4, in particular we explicitly construct this routing in the proof of Theorem 3); however, further work is needed to implement and evaluate it on heterogeneous networks.

Chapter 4

REWIRE: Designing Unstructured Datacenter Networks

4.1 Introduction

As previously described, organizations have deployed a considerable amount of datacenter infrastructure in recent years. Much of this growth, however, has come from the expansion and upgrading of existing facilities. For example, a recent survey found that nearly 2/3s of datacenter operators in the U.S. have added datacenter capacity in the past 12–24 months and 36% plan on adding capacity in 2011 [41]. Large datacenter operators such as Amazon and Google report that they add computing power to their datacenters every day [63]. As described in Chapter 2, previous DCN architectures are not flexible enough to support the cost-effective addition of servers.

While LEGUP improves the process, operators still have little guidance when planning and executing a datacenter expansion. Designing a new or updated network is a challenging optimization problem that needs to minimize multiple objectives while meeting many constraints. Most physical data centers designs are unique, so expansions and upgrades must be custom designed for each datacenter (see, e.g., industry white papers [119]). The optimization challenge is to maximize network performance (which includes bisection bandwidth, end-to-end latency and reliability) while minimizing costs and satisfying a large number of constraints.

We propose REWIRE, a optimization framework to design new, upgraded and expanded DCNs. Unlike previous solutions, REWIRE does not place restrictions on the space of topologies considered. Instead, it considers the the space of all topologies feasible under a user-specified datacenter model, and designs networks that simultaneously maximize agility and minimize end-to-end latencies. We find that arbitrary DCN topologies have significant performance benefits compared to previously considered regular topologies. When designing *greenfield* (i.e., new) datacenter networks, REWIRE's networks have at least 500% more bisection bandwidth than a fat-tree constructed with the same budget. REWIRE also significantly outperforms other approaches (included LEGUP) when designing DCN upgrades and expansions. The use of unstructured topologies in the datacenter does create operational and management concerns since most DCN architectures are topology-dependent. However, recent proposals support routing and load balancing on arbitrary DCNs, so this is not a major barrier to adoption. We discuss this further in Sec. 4.4.

Unlike LEGUP (Chapter 3), we now seek a general DCN design framework—one that accepts any network as input and returns arbitrary topologies. This is a challenging optimization problem because of the huge search space. Therefore, REWIRE's network design procedure uses local search to explore the space of all feasible network topologies. This procedure needs to compute the per-

formance of each candidate topology, which involves computing the topology's agility. We are not aware of any previous polynomial-time algorithm to compute the agility (as defined in Section 1.3) of an arbitrary network; however, we show that computing agility is equivalent to solving a linear program (LP) described by Kodialam et al. to compute an optimal *oblivious* routing of the hose TMs [81, 80]. Unfortunately, this LP has $O(n^4)$ variables and constraints, where n is the number of switches in the network, so it is expensive to find even for small networks. To speed this process, we implement an $(1+\epsilon)$-approximation algorithm to compute this LP [82]. We further speed the runtime of this approximation algorithm implementing its bottleneck operation—an all-pairs shortest-path computation—on the GPU using NVIDIA's CUDA framework [98]. Our implementation is 2–23x faster than a high-performance CPU implementation. Additionally, we utilize a heuristic based on the *spectral gap* of a graph, which is the difference between the smallest two eigenvalues of a graph's adjacency matrix. We find that the spectral gap of a graph is a useful heuristic for candidate selection, especially when designing greenfield (newly constructed) DCNs.

The model of a DCN used by REWIRE is the same as LEGUP, and this model is described in Section 3.2.

4.2 REWIRE Algorithm

We now describe the REWIRE framework. The REWIRE algorithm performs *local search*, so it starts with a *candidate solution*, which is a network design that does not violate any constraints. It explores the *search space* of all candidate solutions by moving from one candidate to another by modifying local properties of the solution until a near-optimal solution is found. This local search only optimizes the network's wiring—it does not add switches to the network. Therefore, we end this section by describing how to extend our approach to add new switches to the network as well.

4.2.1 Optimization problem formulation

REWIRE's goal is to find a network with maximal performance, subject to numerous operator-specified constraints.

Optimization objective

REWIRE designs networks to jointly maximize agility while minimizing the worst-case latency between ToR switches, that is, given the fixed scalars α and β, our objective function is:

$$\text{maximize:} \quad \alpha \cdot \text{bw}(G) - \beta \cdot \text{latency}(G)$$

where $\text{bw}(G)$ and $\text{latency}(G)$ are defined as follows:

- **Agility:** is denoted $\text{bw}(G)$ for a network $G = (V, E)$. Recall that the agility of a network depends on the *rate*, $r(i)$, of a node i, which we define as the peak amount of traffic v can initiate or receive at once. For example, a server v with a 1 Gbps NIC has $r(v) =$1 Gbps. For simplification, we aggregate the bandwidth from all servers attached to a ToR switch s at that switch, that is, we let the rate $r(i)$ of a ToR switch i be the sum of the rates of servers directly connected to the switch (e.g., a ToR switch connected to 40 servers, each with a 1 Gbps NIC, has a rate of 40Gbps). Let the bandwidth of a link e be denoted by $w(e)$. The agility of a network G is then:

$$\text{bw}(G) = \min_{S \subseteq V} \frac{\sum_{e \in \delta(S)} w(e)}{\min\{\sum_{i \in S} r(v), \sum_{i \in \overline{S}} r(v)\}}$$

  where $\delta(S)$ is the set of edges with one endpoint in S and another in $\overline{S} = V - S$.

- **Worst-case latency:** is defined as the worst-case shortest-path latency between any pair of ToR switches in the network, where the latency of a path is the sum of queuing, processing, and transmission delays of switches on the path. We assume that the queuing delay at any port in the network is constant because we have no knowledge of network congestion while designing the network.

Both of these metrics have been considered in the design of DCN topologies, e.g., [55, 89]. However, as far as we know, no previous algorithms could compute the agility of an arbitrary network in polynomial-time. Therefore, we propose such an algorithm by combining previous theoretical results in Sec. 4.2.2.

Operator-specified constraints

REWIRE incorporates a wide range of constraints into its optimization procedure. The constraints support by REWIRE are similar to LEGUP; however, we review them here as well. Any constraints placed on the network by REWIRE are provided by the datacenter operator, and are:

- *Budget.* The maximum amount of money the operator will spend on the network.
- *Existing network topology and specifications.* To perform an upgrade or expansion, we need the existing topology. If designing a greenfield network, then REWIRE needs a set of ToR switches given as the existing network because our current design does not attach servers to ToR switches. This input needs to include specifications for all network devices. For switches this includes their neighbors in the network and relevant details such as the number of free ports of each link rate. Our implementation does not support different link types (e.g., copper vs. optical links); however, it would be easy to extend it so that links include the type of connectors on the ends.
- *Link prices and specifications.* We need a price estimate for labor and parts for each link length category.
- *Available switch prices and specifications (optional).* If one would like to add new switches to the network, REWIRE needs as input the prices and specifications of a few switches available on the market. Specifications include number and speeds of ports, peak power consumption, thermal output and the number of rack slots the switch occupies.
- *Datacenter model (optional).* Consists of the following:
  - Physical layout of racks;
  - Description of each rack's contents (e.g. switches, servers, PDUs, number of free slots);
  - Per rack heat constraints; and/or
  - Per rack power constraints.

  The datacenter model places constraints on individual racks. We use these constraints to, for example, restrict the placement of new switches to racks with enough free slots.
- *Reliability requirements (optional).* This places a constraint on the number of links in the min-cut of the network. That is, this is the minimum number of link removals necessary to partition the network into two connected components.

4.2.2 Local Search Approach

REWIRE uses simulated annealing (SA) [79] to search through candidate solutions. SA is a meta-heuristic that attempts to find a globally optimal solution of a given function in a large search space. We denote the search space by $\mathcal{S}$. This is the set of all network topologies limited by the operator-defined constraints. Each topology $S \in \mathcal{S}$ is a candidate solution.

Our algorithm also uses the following variables:

- A real valued energy function $E(S)$ defined $\forall S \in \mathcal{S}$ as $E(S) = -\alpha * \text{bw}(S) + \beta * \text{latency}(S)$.
- The neighboring solutions of S, denoted by $N(S)$. We define $N(S)$ to be the set of all connected networks such that a link has been added or removed from S.
- An initial temperature T_{START}. For each run, we find this using a method due to Kirkpatrick [79].
- A decreasing function $T(k) : \mathbb{Z}^+ \rightarrow \mathbb{R}$ called the *cooling schedule*, where $T(k)$ is the temperature during the kth set of I Metropolis iterations. We chose $T(k) = T_{\text{START}} * 0.93^k$, but note there is extensive theory behind choosing cooling schedules [97, 128].
- A constant I, which the number of inner Metropolis iterations to be preformed (explained below). As $I \rightarrow \infty$, SA finds a guaranteed optimal solution, but the algorithm runtime is unfeasible for very large values of I [19]. For our purposes, we set I to 1000. This value was found by experimentation.
- An initial state $S_0 \in \mathcal{S}$. The selection of an initial candidate solution is described below in Section 4.2.2.

The goal of SA is to find the solution that minimizes E. To find this network, we perform I Metropolis iterations K times as follows, starting at T_{START}. Suppose that S is the candidate solution under consideration by an iteration. Then, the following actions are performed for neighbor selection:

1. We choose random nodes i and j from the nodes of S. We select a random value $R \in \{0, 1\}$.
2. If $R = 1$, we attempt to generate S' by adding a 10 Gbps link between i, j in S subject to the port and budget constraints. If this addition fails, we attempt to add a 1 Gbps link. If either addition is successful, the move is accepted and otherwise rejected. Otherwise, if $R = 0$, we generate S' by attempting to remove a link of random speed between i and j (if one exists), subject to the connectivity constraint. If the link removal fails we reject the move. Otherwise, if $E(s') = E(s)$, the move is accepted unconditionally. If $E(S') > E(S)$ the move is accepted with probability $e^{\frac{-(E(S')-E(s)}{T}}$, known as the *Metropolis criterion*. SA avoids getting caught in local maxima by sometimes taking suboptimal moves. The Metropolis criterion controls this risk as a function of temperature: the limit of the criterion is 1 as $t \rightarrow \infty$ and 0 as $T \rightarrow 0$. When T is high bad moves are likely to be accepted, but when T is close to 0, bad moves are accepted with very low probability.

After I Metropolis iterations are preformed, the temperature is decreased according to the cooling schedule. The process is repeated K times, where K is the smallest integer such that $T(k) < 0.05$.

The SA procedure needs to compute the energy function for each candidate solution. Since we have that $E(S) = -\alpha * \text{bw}(S) + \beta * \text{latency}(S)$, we need to compute the agility of S to find its performance. In the next section, we describe how to find this in polynomial time.

Evaluating a candidate solution

Our definition of performance is the weighted sum of agility and latency, so we now describe how to compute each metric.

Agility: recall that agility is defined on the minimal cut of all cuts of a graph. This is too expensive to compute directly on arbitrary graphs because a graph can have exponentially many cuts. However, we now show that the problem of finding a network's cut bandwidth can be reduced to a maximal flow problem under the hose constraints.

Two-phase routing, proposed by Lakshman et al. [82], is an oblivious routing scheme, meaning that it finds a randomized routing that minimizes the maximum link utilization for any traffic matrix in a polyhedron of traffic matrices. Two-phase routing divides routing into two phases. During phase one, each node forwards an α_i fraction of its ingress traffic to node i. During stage two, nodes forward traffic they received during phase one on to its final destination. We say that $\alpha_1, \ldots, \alpha_n$ are the load-balancing parameters of a graph $G = (V, E)$. The optimal values of the α_i values depends on G and the set of hose TMs $\mathcal{T}(r(1), \ldots, r(n))$ for $V = 1, \ldots, n$. Note also that two-phase routing assume that flows are splittable (i.e., an s-t flow can be routed on multiple paths).

Before describing how to compute a two-phase routing, we show that finding the load-balancing parameters of a network is equivalent to finding the cut bandwidth of a network. We denote a cut of G by $(S, \overline{S})$, where S and $\overline{S}$ are connected components of G and $\overline{S} = V - S$. Let $c(S, \overline{S})$ be the capacity of all edges with one endpoint in S and the other in $\overline{S}$. The following theorem shows how to compute agility of a graph given $\alpha_1, \ldots, \alpha_n$.

Theorem 4 (Curtis and López-Ortiz [37]). *A network $G = (V, E)$ with node rates $r(1), \ldots, r(n)$ and load-balancing parameters $\alpha_1, \ldots, \alpha_n$ can feasibly route all hose TMs $\mathcal{T}(r(1), \ldots, r(n))$ using two-phase routing if and only if, for all cuts $(S, \overline{S})$ of G,*

$$c(S, \overline{S}) \geq \sum_{i \in \overline{S}} \alpha_i \cdot \sum_{i \in S} r(i) + \sum_{i \in S} \alpha_i \cdot \sum_{i \in \overline{S}} r(i)$$

where $\overline{S} = V - S$.

Proof. Necessity is not difficult to show by way of contradiction. We omit the details here; see, e.g., [91] for a proof that necessity holds for any multicommodity flow.

To see sufficiency, let us first define the multicommodity flow problem solved by two-phase routing. Viewed as a multicommodity flow problem, the two-phase routing problem is a set of $2\binom{n}{2} = n(n-1)$ commodities, specified as follows.

$$\begin{aligned} \mathcal{W} = \{((s,i), \alpha_i r_s)\} \quad & \forall s, i \in V \quad \text{Stage 1} \\ \cup \{((i,t), \alpha_i r_t)\} \quad & \forall i, t \in V \quad \text{Stage 2} \end{aligned}$$

Since we have captured all flows between nodes, it's clear that the two-phase routing routing problem with load balancing parameters $\alpha_1, \ldots, \alpha_n$ admits a solution if and only if the multicommodity flow $\mathcal{W}$ has a feasible solution.

Assume that all cuts $(S, \overline{S})$ of G have capacity at least $c(S, \overline{S}) \geq \sum_{i \in \overline{S}} \alpha_i \cdot \sum_{i \in S} r(i) + \sum_{i \in S} \alpha_i \cdot \sum_{i \in \overline{S}} r(i)$. To see that G can serve all hose TMs, we will show that there exists a feasible solution to the multicommodity flow problem $\mathcal{W}$. We need to specify the rate of a commodity, so let $r(k) = r$ for a commodity $k = (s, t, r)$. We denote the set of i's incoming links by $N^-(i)$ and its outgoing links by $N^+(i)$.

A directed graph is called *capacity balanced* if, for all $i \in V$, $N^+(i) + \text{demand}(i) = N^-(i) + \text{supply}(i)$, where $\text{demand}(i)$ is the sum of commodity rates with i as the target and $\text{supply}(i)$ is the sum of commodity rates where i is the source. Nagamochi and Ibaraki [92, 93] have shown that a feasible solution to a multicommodity flow problem $\mathcal{W}$ exists on a capacity balanced network if, for all its cuts $(S, \overline{S})$, $c(S) \geq \sum_{k \in \mathcal{W}_S} r(k)$, where $\mathcal{W}_S = \{(s, t, r) \in \mathcal{W} : s \in S \text{ and } t \in \overline{S}\}$.

Because we We assume $\sum_{k \in \mathcal{W}_S} r(k) = \sum_{i \in \overline{S}} \alpha_i \cdot \sum_{i \in S} r(i) + \sum_{i \in S} \alpha_i \cdot \sum_{i \in \overline{S}} r(i)$ for the multicommodity flow $\mathcal{W}$. Then, if G is a capacity balanced network, then a feasible solution to $\mathcal{W}$ exists. Consider an arbitrary $i \in V$. we have $N^+(i) = N^-(i)$ by definition, since G is bidirectional. In a hose TM D, we have $\sum_{j \in V, j \neq i} D_{ij} = r_i$ and $\sum_{j \in V, j \neq i} D_{ji} = r_i$, so $\text{supply}(i) = \text{demand}(i)$. Therefore, G is a capacity balanced network, and so a feasible solution to $\mathcal{W}$ exists. □

This theorem shows that a multi-commodity flow version of the famous max-flow, min-cut theorem [31] holds for networks using two-phase routing under the hose model. And the theorem shows that any network that can feasibly route all hose TMs has a cut bandwidth of at least 1.

We now show the computation of $\alpha_1, \ldots, \alpha_n$ using the results of Lakshman et al., who proved that the α_i values can be found with linear programming (LP) [82]. We use the following notation. Let f be a *network flow* in the optimization sense. We use f_k to denote flow k where $s(k)$ and is the origin and $t(k)$ is the destination of the flow. Then let $f_k(i,j)$ be the amount of flow placed on edge (i,j) by flow f_k. We denote the outgoing edges from node i by $\delta^+(i)$ its incoming edges by $\delta^-(i)$. The capacity of an edge (i,j) is denoted by $c(i,j)$.

Optimal two-phase routing LP:

$$\min \ \mu$$

Subject to:

$$\sum_{w \in \delta^-(y)} f_k(w,y) = \sum_{z \in \delta^+(y)} f_k(y,z) \ \forall y \neq s(k), t(k) \ \forall k \tag{4.1}$$

$$\sum_{k=1}^{K} f_k(i,j) \leq \mu \cdot c(i,j) \tag{4.2}$$

$$\sum_{j \in \delta^+(i)} f_k(i,j) = \alpha_{s(k)} r(i) + \alpha_{t(k)} r(i) \tag{4.3}$$

$$i = s(k), \forall k$$

$$\sum_i \alpha_i = 1 \tag{4.4}$$

Their LP can be computed in polynomial-time using an LP solver; however, it is computationally expensive because it has $O(n^4)$ constraints and $O(n^4)$ variables. In our initial testing, we found that computing this LP for a network with 200 nodes and 400 (directed) edges needs more than 22GB of memory using IBM's CPLEX solver [72]. Even with only 50 node networks, this LP takes up to several seconds to compute. Because REWIRE's local search approach needs to evaluate thousands of candidate solutions, this LP is not fast enough for our purposes.

To solve these issues, we implemented an approximation algorithm due to Kodialam et al. [82] to compute $\alpha_1, \ldots, \alpha_n$ in polynomial-time. This algorithm finds a solution guaranteed to be within an $(1+\epsilon)$ factor of optimal. The algorithm follows an approach developed by Garg and Koenemann [50]. It augments each node i's value α_i iteratively. At each iteration, the algorithm computes

a weight $w(e)$ for each edge $e \in E$ and then pushes flow to i along the shortest-paths to i based on these weights. The bottleneck operation in this algorithm is computing the shortest-path from each node to each other node given the weights $w(e)$. This operation needs to perform an all-pairs shortest-path (APSP) computation. The best running times we are aware of for an APSP algorithm is $O(n^3)$ (deterministic) [31] and $O(n^2)$ (probabilistic) [101]. Because this operation is the bottleneck, we implemented an APSP solver on a graphics processing unit (GPU). Used this way, the GPU is a powerful, inexpensive co-processor with hundreds of cores.

We implemented a recursive version of the Floyd-Warshall algorithm for our APSP function using CUDA [21, 98]. The algorithm uses generalized matrix multiplication (GEMM) as an underlying primitive. GEMM exhibits a high degree of data parallelism and we can significant speedups by exploiting this attribute. Finally, we perform a parallel reduction to find the maximal path in the distance matrix. Parallel reduction is an efficient algorithm for computing associative operators in parallel. It uses $\Theta(n)$ threads to compute a tree of partial results in parallel. The number of steps is bounded by the depth of the tree which is $\Theta(\log n)$ [68, 65].

Latency: we compute latency(G) by solving an APSP problem on $G = (V, E)$. We set this problem up to model the expected amount of time it takes to process packets in network equipment. We associate with each edge $e \in E$ a weight $w(i, j)$. This represents the expected time to forward packets on that link. We define $w(i, j)$ of an edge (i, j) to be the sum of queuing delays, forwarding time and processing delay at (i, j)'s endpoints. We now have that latency(G) is then the maximum shortest-path between any pair of ToRs for (G, w).

In our model, we assume that the processing delay of each switch is specified by the operator. To estimate queuing delays, we assume that the forwarding time is 1500 Bytes divided by the link rate (1 or 10 Gbps) and that each packet is queued behind two other packets at each switch on its path. Our assumption of uniform queuing delays is not realistic but necessary: since we assume no knowledge of the network load, we cannot accurately determine queuing delays.

Initial candidate selection

Due to the large search space, finding optimized solutions with simulated annealing takes an infeasible amount of time for large networks, especially when there is significant room for improvement in the network. Therefore, we added the ability to seed REWIRE's simulated annealing procedure with a candidate solution. To find a seed candidate, we use a heuristic based on the *spectral gap* of a

graph. Before we define the spectral gap of a graph, we need to introduce a few terms. We consider the matrix L, defined as follows for a graph $G = (V, E)$:

$$L(i,j) = \begin{cases} d(i) & \text{if } i = j, \\ -1 & \text{if } i \text{ and } j \text{ are adjacent,} \\ 0 & \text{otherwise.} \end{cases}$$

The *Laplacian* of $G = (V, E)$ is the matrix:

$$\mathcal{L}(i,j) = \begin{cases} 1 & \text{if } i = j \text{ and } d(i) \neq 0, \\ -\frac{1}{\sqrt{d(i)d(j)}} & \text{if } i \text{ and } j \text{ are adjacent,} \\ 0 & \text{otherwise.} \end{cases}$$

The eigenvalues of $\mathcal{L}$ are said to the be *spectrum* of G, and we denote them by $\lambda_0 \leq \cdots \leq \lambda_{n-1}$. It can be shown that $\lambda_0 = 0$ [28] for a proof. We say that λ_1 is the *spectral gap* of G.

Intuitively, a graph with a "large" spectral gap will be regular (we omit a precise definition of large here—see any text book for details [28]) and the lengths of all shortest-paths between node pairs are expected to be similar. Maximizing a network's spectral gap is not a new objective function in the network design literature (e.g., [42, 113]).

This is not surprising, since As an example, the following lemma shows how the spectral gap correlates with the diameter of a graph.

Lemma 5 ([28]). *Let G be a graph with diameter $D \geq 4$, and let K denote the maximum degree of any vertex of G. Then*

$$\lambda_1 \leq 1 - 2\frac{\sqrt{k-1}}{k}(1 - \frac{2}{D}) + \frac{2}{D}$$

That is, a graph with a large spectral gap has a low diameter.

We therefore modify REWIRE to optionally perform a two stage simulated annealing procedure. In stage 1, its objective function is to maximize the spectral gap. The result from stage 1 is used to seed stage 2, where its objective function is to maximize agility and minimize latency. This way, stage 2 starts with a good solution and can converge quicker. When this two stage procedure is used, we say REWIRE is in *hotstart mode*.

Note that a network with a maximal spectral gap of all candidate solutions does not necessarily mean that the network will also have high agility. The spectral gap metric does *not* take the hose

constraints into account, so it is not directly optimizing for agility. Instead, it creates networks that are well-connected, which tend to have high agility (see [37] for details), but that is not necessarily the case, especially for heterogeneous networks.

4.2.3 Adding switches to the network

REWIRE's simulated annealing procedure does not consider adding new switches to the network—it only optimizes the wiring of a given set of switches. To find network designs with new switches, we run REWIRE on the input network plus a set of new switches that are not attached to any other switch. REWIRE attaches the new switches to the existing network randomly, and then begins its simulated annealing procedure.

While simple, this approach does not scale well. If an operator has input specifications for k new switch types, then we need to run REWIRE $k!$ times to consider all possible combinations of switch types. We believe this could be improved by applying heuristics to select a set of new switches; however, we leave investigation of such heuristics to future work.

4.3 Evaluation

We now present our evaluation of REWIRE. First, we describe the inputs used in the evaluation, then the approaches used for comparison with REWIRE. Finally, we present our results using REWIRE to design greenfield, upgraded and expanded networks.

4.3.1 Inputs

Existing networks

To evaluate REWIRE's ability to design upgrades and expansions of existing networks, we use a scaled-up model of the University of Waterloo's School of Computer Science machine room network (denoted by SCS network) as input. This model is similar to the model we used in the previous chapter. The SCS network has 19 ToR, 2 aggregation and 2 core routers. Each ToR connects to a single aggregation switch with a 1 or 10 Gbps link and both aggregation switches connect to both core switches with 10 Gbps links. The network is composed of a heterogeneous set of switches as

ToR switches

Hose uplink rate	Uplinks (1, 10 Gbps)	No. switches
28	8, 2	5
40	8, 4	8
8	8, 0	4
2	2, 0	2

Aggregation switches

Line cards	Line card slots	No. switches
3x 24 1 Gbps, 1x 2 10 Gbps	6	1
4x 4 10 Gbps	6	1

Table 4.1: Existing switches in the SCS datacenter model.

described in Table 4.1. To scale the network up, we assume that each ToR switch is attached to a full rack of 40 servers, and we assume the hose uplink rates of each ToR switch as described in the table.

To predict the cost of a network design, REWIRE needs the distance between each ToR switch pair. We do not have this data for the SCS network. Therefore, we label each switch with a unique label from $1, \ldots, n$. The distance between switches i and j is then $|i - j|$. The distance from i to the nearest 25% of switches is categorized as "short", the distance to the next 50% is "medium" and then distance to the final 25% is "long". We use these distance categories to estimate the price of adding a link between two switches.

Switches and cabling

We separate the costs of adding a cable into the cost of the cable itself and the cost to install it. Mudigonda et al. report that list prices for 10 Gb cables are between $45–95 for 1m cables and $100–400 for 10m cables depending on the type of cable (copper or optical and its connector types) [89]. Optical cables are more expensive than copper cables, but they are available in longer lengths. To obtain a reasonable estimate of cabling costs without creating too much complexity, we divide cable runs into three groups: short, medium and long lengths. The costs we use are shown in Table 4.2. We also charge an installation fee for each length group (also shown in the table). Whenever an existing cable is moved, we charge the appropriate installation fee given the cable's length.

Table 4.3 shows the costs we assume to buy various switches.

Rate	Short ($)	Medium ($)	Long ($)
Cable costs			
1 Gbps	5	10	20
10 Gbps	50	100	200
Installation and re-wiring costs			
	10	20	50

Table 4.2: Prices of cables and the cost to install or move cables.

Ports	Watts	Price ($)
24 1 Gbps	100	250
48 1 Gbps	150	1,500
48 1 Gbps, 4 10 Gbps	235	5,000
24 10 Gbps	300	6,000
48 10 Gbps	600	10,000

Table 4.3: Switches used as input in our evaluation (prices are the same as in Section 3.6). Prices are representative of street prices and power draw estimates are based on a typical switch of the type according to manufacturers' estimates.

4.3.2 Comparison approaches

We compare REWIRE against the following DCN design solutions.

Fat-tree: was proposed by Leiserson [84], and is a k-ary multi-rooted tree. This is a specific form of the Clos topology [30]. We assume a 3-level fat-tree topology and that all switches in the network must be homogeneous. Building an optimal fat-tree for a set of servers given switch specifications is NP-hard [89], so we upper bound the performance that a fat-tree network with a specified budget could achieve. To do this, we compute the number of ports the fat-tree needs, and bound the cost of switches by the min-cost port of a given rate (e.g., a 1 Gb port costs at least $\$250/24$ and a 10 Gb port costs at least $\$10K/48$). To estimate the cost of cabling, we assume that server to ToR links are free, and that ToR to aggregation switches are medium length and aggregation to core links are long length.

Greedy algorithm: we implemented a greedy heuristic to determine if REWIRE's more sophisticated local search approach is necessary. The algorithm iterates over all pairs of switches as follows. First, it computes the change in agility and latency that would result from adding a 1 Gbps and 10 Gbps between every pair of switches and stores the result. At the end of each iteration, the algorithm adds the link that increases the network's performance the most. If no link changes the agility or latency during an iteration then a random link is added. This iteration continues until the budget is exhausted or no links can be added because all ports are full. Note that this algorithm does not rewire the initial input—it only adds links to the network until the budget is exhausted. This algorithm performs $O(n^2)$ agility computations at each iteration, and hence does not scale to graphs with more than $\sim$40 nodes. Therefore, we do not compare against the greedy algorithm for any network with more than 40 nodes.

LEGUP: as described in Chapter 3.

Random graph: Singla et al. proposed a DCN architecture Jellyfish, which is based on random graph topologies [110]. Random graphs have nice connectivity properties, and they showed that it is less expensive to build a random graph than a fat-tree much of the time. To estimate the performance a random graph can achieve with a specified budget, we determine the expected radix of each ToR switch given number of links one can install with the budget. Then, we compute the expected agility and diameter of the network following Singla et al.'s approach. Note that we use the *expected* agility and diameter, rather than explicitly constructing these networks.

4.3.3 REWIRE settings

REWIRE can operate in several different modes. These are:

- *Spectral gap mode:* sets REWIRE's objective function to maximize the spectral gap of its output.
- *CPLEX or approximation:* sets the method REWIRE uses to compute the agility of a network. In CPLEX mode, REWIRE uses IBM's CPLEX solver [72] to compute the agility exactly; whereas in approximation mode, REWIRE finds the agility of a candidate solution using the FPTAS previously described.
- *Hotstart:* this mode finds a candidate solution in spectral gap mode, which is used as a seed solution to stage 2, where the objective function is changed to our standard definition of per-

formance.

When describing the results from a scenario, we note what mode REWIRE was in.

Finally, REWIRE uses a local search algorithm, so it may not always find an optimal answer, especially if it is not given enough time to run. For all experiments in this paper, we let REWIRE run for 72 hours. This time was derived experimentally. It was typically not enough time for REWIRE to converge to an optimal answer, so we make no claim that our results here are the best that REWIRE can obtain.

4.3.4 Greenfield networks

We begin by evaluating the effectiveness of REWIRE at designing greenfield, that is, new, DCNs. For this scenario, the input to REWIRE is a set of ToR switches (with no links between them). We use REWIRE in approximation mode. Initially, ToR switches are each attached to servers, but no other switches. The total cost of the network is the cost of these ToR switches plus the wiring budget. We experimented with two types of ToR switches. First, we set all ToR switches have 48 1 Gbps ports, where 24 ports attach to servers and the other 24 are left open. Then, we set all ToR switches to have 48 1 Gbps ports and 4 10 Gbps; each ToR switch attaches to 40 servers with 1 Gbps ports. For both experiments, we built networks to connect 3200 servers.

We compare against the fat-tree, LEGUP, and random topology approaches. The results are shown in Figure 4.1. In the chart, the bars show normalized agility (higher is better and a network with full agility has a normalized agility of 1). The table below the bars indicates the diameter (the worst-case hop count between ToR switches) of the network. We do not compare against the greedy heuristic for these experiments because it is not fast enough for networks with more than 40 nodes.

REWIRE significantly outperforms the other approaches for nearly all budgets. The random network has more agility than REWIRE's network when the budget is $5,000; however, REWIRE's solution has less latency (this network has a diameter one hop less than the expected random network's). We weighted agility and latency equally in this experiment, so REWIRE preferred the solution with less bandwidth, but also less latency. This illustrates the flexibility of REWIRE.

In this scenario, LEGUP outperforms the fat-tree. Depending on the budget and ToR assumptions, LEGUP's networks have 50–100% more agility than a same-cost fat-tree.

We also re-ran the REWIRE experiments using its spectral gap mode. We found that the solutions with a maximal spectral gap had the same performance as the solutions found by REWIRE in

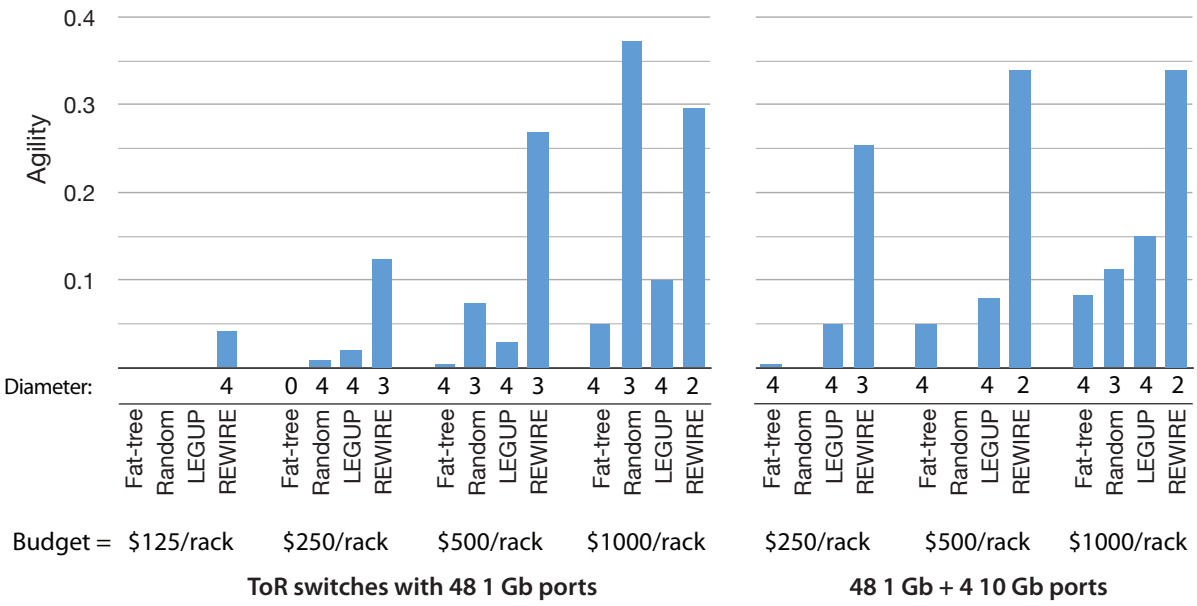


Figure 4.1: Results of designing greenfield networks for 3200 servers using a fat-tree, random graph and REWIRE for two ToR switch types. The results on the left used ToR switches with 48 1 Gbps ports and the results on the right used ToR switches with 48 1 Gbps ports and 4 10 Gbps. Missing bars for the random graph indicate that the network is expected to be disconnected. A network with agility 1 has full agility, and a network with diameter 1 is fully connected.

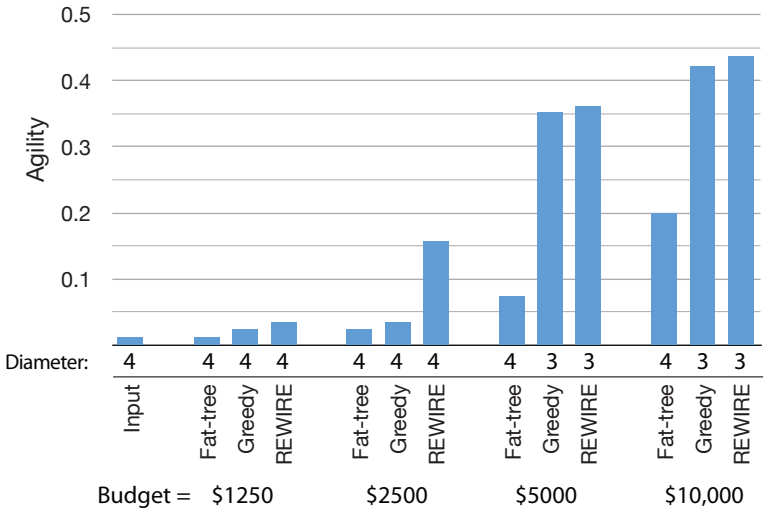


Figure 4.2: Results of upgrading the SCS topology with different budgets and algorithms.

approximation mode. This implies that the spectral gap is good metric when designing greenfield data centers because it seems to maximize agility and it finds networks with very regular topologies, which may reduce the cost of wiring the network.

4.3.5 Upgrading

We now evaluate REWIRE's ability to find upgrades to existing DCNs. To begin, we compared REWIRE to a fat-tree and our greedy algorithm on the SCS network for several budgets. The results are shown in Figure 4.2.

REWIRE significantly outperforms the fat-tree—its networks have 120–530% more agility than a fat-tree constructed with the same budget, and with budgets over $5K, REWIRE's network also has a shorter diameter (by 1 hop) than a fat-tree. REWIRE also outperforms the greedy algorithm for all budgets, though the greedy algorithm performs nearly as well when the budget is $5K or more. This indicates that a greedy approach performs very well in some settings; however, we have generally observed that the greedy algorithm does not perform well when it has a small budget or the input is very constrained and has few open ports. For example, when the budget is $2,500, REWIRE's network has 350% more agility than the network found by the greedy algorithm.

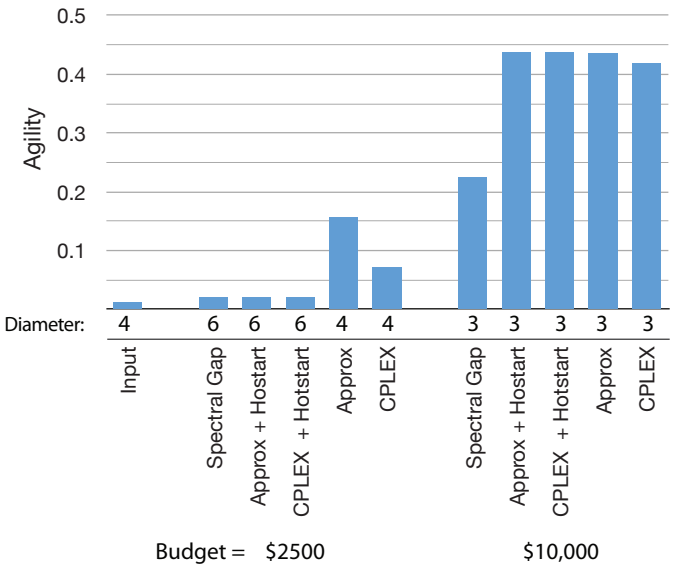


Figure 4.3: Results of upgrading the SCS topology with different REWIRE modes and two budgets.

Next, we compared the various modes of REWIRE for two budgets as shown in Figure 4.3. We observe that the spectral gap and hotstart modes performs poorly when the budget is $2,500. This is likely due to the properties of the spectral gap, which tries to make the network more regular. Because the budget is not large enough to re-wire the network in this regular fashion, optimizing the network's spectral gap creates a candidate solution with poor agility. This problem does not arise when the budget is large enough (as in the case when the budget is $10K), because there is enough money to re-wire the network into this regular structure.

4.3.6 Expanding

We now examine the performance of the algorithms as we expand the datacenter over time by incrementally adding new servers. We tested two expansion scenarios here.

First, we expanded the SCS datacenter by adding 160 servers at a time, until we have added a total of 640 servers to the datacenter. The results are shown in Figure 4.4. For REWIRE and the greedy algorithm, we used ToR switches with 48 1 Gbps and 4 10 Gbps ports, so each ToR attaches to 40 servers. For the 1 Gb fat-tree, we used ToR switches with 24 1 Gbps ports. The budgets shown in the

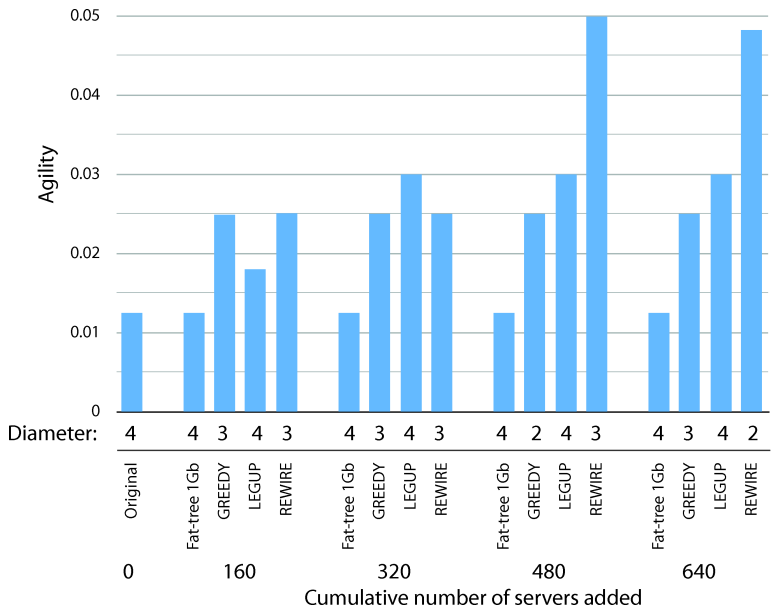


Figure 4.4: Results of iteratively expanding the SCS datacenter.

figure are the budgets for cabling and aggregation and core switches. That is, the budgets shown do not take into account the price of ToR switches.

We found that REWIRE outperforms the fat-tree, the greedy algorithm, and LEGUP in this scenario. The fat-tree is not able to improve the agility of the expanded network beyond the agility of the initial SCS DCN, whereas the greedy algorithm, LEGUP, and REWIRE do. This scenario shows the limitations of the greedy algorithm. After four expansions, REWIRE's network has nearly twice as much agility as the greedy algorithm's network, and LEGUP, despite its topology restrictions, also outperforms the greedy algorithm after the second iteration.

Next, we evaluated REWIRE's performance when constructing, and then expanding a greenfield datacenter. In these experiments, we built the initial DCN with a budget of $40K using LEGUP, REWIRE, or a fat-tree with 1 or 10 Gb links. This initial datacenter contained 1600 servers. Then, we iteratively expanded this datacenter by adding 400 servers at a time. For each expansion, the algorithms were given a total budget of $60K (this includes the cost of ToR switches). The results are shown in Figure 4.5.

Again, REWIRE performed better than either fat-tree configuration and LEGUP. Its initial net-

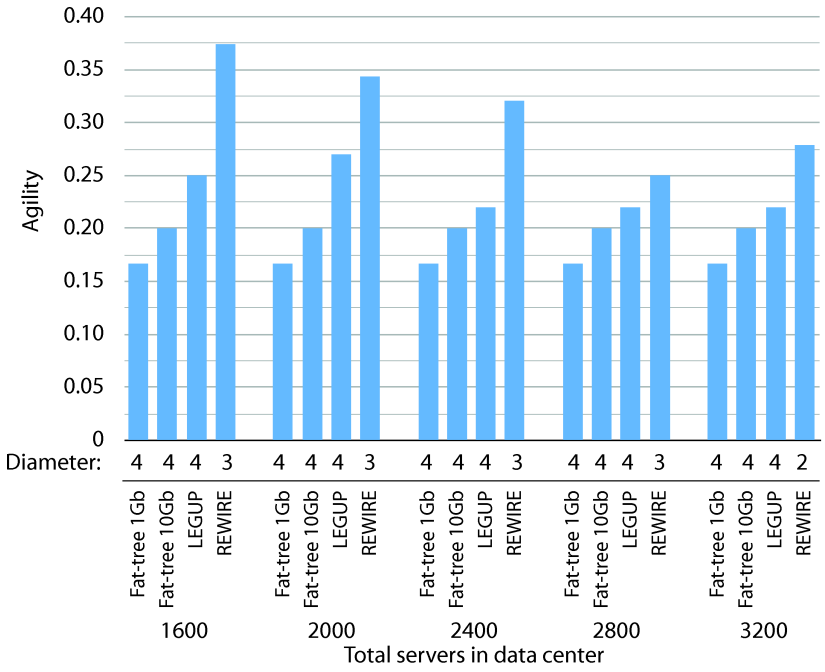


Figure 4.5: Results of iteratively expanding a greenfield network.

work has more agility than the networks designed by the other approaches, and it maintains this lead as the datacenter is iteratively expanded.

4.3.7 Quantitative results

Time is a bottleneck in our network design algorithm. When using a single CPU core, it can take up to two weeks to converge when the input contains 200 switches. To speed this up, we implemented a GPU-based all-pairs shortest-path (APSP) solver. As the APSP computation is the bottleneck operation in REWIRE's operation, we found that doubling the speed of the APSP computations nearly halves REWIRE's total runtime. In summary, we found our GPU-based APSP implementation is 28–244x faster than a naive implementation [31]. As the network size increases, larger speedups are possible.

4.4 Operating an Unstructured DCN

Because of their performance benefits, we advocate the adoption of non-regular topologies in the datacenter. Doing so, however, raises architecture issues. In particular, we need an architecture that provides addressing, routing, load-balancing and cost-effective management on unstructured DCN topologies if they are to be of practical use. We now show how previous work can perform these functions.

Addressing and routing: have been the focus of much recent work due to the difficulty of scaling traditional, distributed control-plane protocols to connect more than a couple thousand servers. Protocols such as SEATTLE [77] and TRILL [116] provide scalable L2 functionality on arbitrary topologies, but do not provide multipath routing, which is needed to fully utilize dense networks. Another option is SPAIN, an architecture for multipath routing on arbitrary topologies [88]. SPAIN performs source-routing and uses VLANs to partition the network into many spanning trees.

Load-balancing: our algorithms assume that a network's bandwidth can be fully exploited by the load-balancing mechanism. This assumption is not valid when using single-path protocols like spanning tree; however, near-optimal load-balancing can be achieved on arbitrary topologies by using

Multipath TCP [103] or SPAIN [88]. Multipath TCP exposes multiple end-to-end paths to end-hosts, and they independently attempt to maximize their bandwidth by performing adaptive load balancing across these paths. This approach can achieve 100% utilization on a fat-tree [103]. Multipath TCP has not yet been evaluated on arbitrary topologies; however, its performance on regular topologies indicates it will be able to fully utilize arbitrary topologies as well. SPAIN [88] performs reactive load balancing at the end-hosts over the various VLANs exposed to each end-host. It has been shown that SPAIN can fully utilize HyperX, FatTree and BCube topologies. Because it performs well on this range of regular topologies, we believe it will also perform well on the topologies REWIRE designs.

Based on the evaluations of SPAIN and Multipath TCP, we believe that these protocols would be able to fully utilize REWIRE's topologies. Alternatively, centralized flow controllers like Hedera [8] and Mahout [34] could be modified to provide near-optimal load-balancing on arbitrary topologies.

Management and configuration: managing a DCN with an irregular topology may be more costly and require more expertise than a vendor-specified DCN architecture. In particular, addressing is more difficult to configure an irregular topology, because we cannot encode topologic locality in the logical ID of a switch (typically a switch's logical ID is its topology-imposed address or label).

To mitigate this issue, we suggest configuring the network using Chen et al.'s generic and automatic datacenter address configuration system (DAC) [27]. An interesting benefit of DAC's design is that it can automatically identify mis-wirings. This operation is especially useful for us because wiring an arbitrary topology may be more difficult than a regular, tree-like topology. Software-defined networking, such as implemented by OpenFlow, is also a promising solution for arbitrary DCN management (see [114] and Chapter 6). We believe these solutions can solve many of the management problems that may arise from the introduction of irregular topologies in the datacenter, and we leave further investigation to future work.

4.5 Discussion

The $(1+\epsilon)$-approximation algorithm we implemented to compute the agility of a network is numerically unstable. At each iteration of its operation, it performs an all-pairs shortest-path computation. To do this, it needs to compare increasingly minute numbers at each successive iteration. We found

it returns incorrect shortest-paths trees after enough iterations because these comparisons are made on numbers less than 10^{-40}. Because of this numerical instability, we could not run the approximation algorithm on inputs larger than 200 nodes and 200 edges. Nor could we run it with very small values of ϵ because the algorithm performs more iterations as ϵ decreases.

We did not explicitly consider designing upgrades or expansions that can be executed with minimal disruption to an existing DCN. However, it is possible to disable REWIRE's support for moving existing network links. Another approach is to modify the cost constraints (e.g., moving a link costs five times more than adding a new one) so that rare, significantly beneficial re-wirings are permissible.

Chapter 5

Datacenter Network Traffic Engineering with Mahout

5.1 Introduction

As previously noted, datacenter switching fabrics need huge amounts of bisection bandwidth to enable the transfer of huge quantities of data between thousands of servers. For example, Hadoop [61] performs an all-to-all transfer of up to petabytes of files during the shuffle phase of a MapReduce job [39]. Further, to better consolidate employee desktop and other computation needs, enterprises are leveraging virtualized datacenter frameworks (e.g., using VMWare [120] and Xen [124, 15]), where timely migration of virtual machines requires high throughput network.

Designing datacenter networks using redundant topologies such as LEGUP's heterogeneous Clos, REWIRE's unstructured topologies, or a fat-tree [30, 10] builds a network with sufficient bisection bandwidth. However, traffic engineering is necessary to fully utilize the available bandwidth in such topologies [8]. A key challenge to traffic engineering here is that the flows come and go too quickly in a datacenter to compute a route for each individually; for example, Kandula et al. report 100K flow arrivals a second in a 1,500 server cluster [76].

For effective utilization of the datacenter fabric, we need to detect *elephant flows*—flows that transfer significant amount of data—and dynamically orchestrate their paths. Datacenter measurement studies show that a large fraction of datacenter traffic is carried in a small fraction of flows [55,76]. These studies report that 90% of DCN flows carry less than 1 MB of data and more than 90% of bytes transferred are in flows greater than 100 MB. Hash-based flow forwarding techniques such as equal-cost multi-path (ECMP) routing [67] works well only for large numbers of short (or *mice*) flows and no elephant flows. For example, Al-Fares et al.'s Hedera shows that dynamically scheduling elephant flows effectively can yield as much as 113% higher aggregate throughput compared to ECMP for some DCN workloads [8].

Existing elephant flow detection methods have limitations that make them unsuitable for datacenter networks. These proposals use one of three techniques to identify elephants: (1) periodic polling of statistics from switches, (2) streaming techniques like sampling or window-based algorithms, or (3) application-level modifications (full details of each approach are given in Section 5.2). We have not seen support for Quality of Service (QoS) solutions take hold, which implies that modifying applications is probably an unacceptable solution. We show that the other two approaches fall short in the datacenter setting due to high monitoring overheads, significant switch resource consumption, or long detection times.

In this Chapter, we assert that the right place for elephant flow detection is at the end-hosts.

To achieve this, we describe Mahout, a low-overhead yet effective traffic management system that uses end-host-based elephant detection. Mahout's design follows the increasingly popular simple-switch/smart-controller model (as in OpenFlow [5]), and so our system is similar to NOX [114], Hedera [8], and DevoFlow (Chapter 6).

Mahout augments this basic design by taking advantage of computational power of datacenter end-hosts. It has low overhead, as it monitors and detects elephant flows at end-hosts via a shim layer in the OS, rather than monitoring at the switches in the network. Mahout does timely management of elephant flows through an in-band signaling mechanism between the shim layer at the end-hosts and the network controller. At the switches, any flow not signaled as an elephant is routed using a randomized load balancing scheme, such as ECMP. Therefore, only elephant flows are monitored and scheduled by the central controller. The combination of end-host elephant detection and in-band signaling eliminates the need for per-flow monitoring in the switches, and hence incurs low overhead and requires few switch resources.

We demonstrate the benefits of Mahout using an analytical evaluation, simulations, and a prototype implementation. We have implemented the end-host shim on Linux, which implements our elephant flow detection algorithm. We have also built a Mahout controller, for setting up switches with default entries and for processing the tagged packets from the end-hosts. Our analytical evaluation shows that Mahout offers one to two orders of magnitude of reduction in the number of flows processed by the controller and in switch resource requirements, compared to Hedera and similar approaches. Our simulations show that Mahout can achieve considerable throughput improvements compared to randomized load balancing techniques, while incurring an order of magnitude lower overhead than Hedera. Finally, experiments with our prototype indicate that our approach can detect elephant flows at least an order of magnitude sooner than network-based approaches.

The key contributions this chapter are: (1) a novel end-host based mechanism for detecting elephant flows in §5.3, (2) design of a centralized datacenter traffic management system that has low overhead yet is effective (§5.3), and (3) simulation and prototype experiments demonstrating the benefits of the proposed design in §5.4–5.5.

5.2 Background

We now describe the relevant background not covered in Chapter 2 on datacenter networks and elephant flow detection.

5.2.1 Datacenter traffic

The heterogeneous mix of applications running in datacenters produces flows that are generally sensitive to either latency or throughput. Latency-sensitive flows are usually generated by network protocols (such as ARP and DNS) and interactive applications. They typically transfer up to a few kilobytes. On the other hand, throughput-sensitive flows, created by, e.g., MapReduce, scientific computing, and virtual machine migration, transfer up to gigabytes. This traffic mix implies that a datacenter network needs to deliver high bisection bandwidth for throughput-sensitive flows without introducing setup delay on latency-sensitive flows.

5.2.2 Identifying elephant flows

The mix of latency- and throughput-sensitive flows in the datacenters means that effective flow scheduling needs to balance visibility and overhead—a one size fits all approach is not sufficient in this setting. To achieve this balance, elephant flows must be identified so that they are the only flows touched by the controller. The following are the previously considered mechanisms for identifying elephant flows:

- *Application-based classification:* This approach requires that applications identify flows they create as a mice or elephant flow. This solution accurately and immediately identifies elephant flows. This is a common assumption for a plethora of research work in network QoS where focus is to give higher priority to latency and throughput-sensitive flows such as voice and video applications [20]. However, this solution is impractical for traffic management in datacenters as each and every application must be modified to support it. If all applications are not modified, an alternative technique will still be needed to identify elephant flows initiated by unmodified applications.

  A related approach is to classify flows based on which application is initiating them. This classifies flows using stochastic machine learning techniques [106] or matching packet header

fields (such as TCP port numbers). While this approach might be suitable for enterprise network management, it is unsuitable for datacenter network management because of the enormous amount of traffic in the datacenter and the difficulty in obtaining flow traces to train the classification algorithms.

- *Collect per-flow statistics:* In this approach, flows are monitored by edge switches. Flow statistics are pulled from edge switches by the controller at regular intervals and are used to classify elephant flows. Hedera [8] and Helios [47] are examples of systems that use such a mechanism. However, this approach does not scale to large networks. First, this consumes significant switch resources: a flow table entry for each flow monitored at a switch. We'll show in Section 5.4 that this requires considerable number of flow table entries. Second, bandwidth between switches and the controller is limited, so much so that transferring statistics becomes the bottleneck in traffic management in datacenter network. As a result, the flow statistics cannot be quickly transferred to the controller, resulting in prolonged sub-par routings.
- *Sampling:* Instead of monitoring each flow in the network, in this approach, a controller samples packets from all ports of the switches using switch sampling features such as sFlow [4]. Only a small fraction of packets are sampled (typically, 1 in 1000) at the switches and only headers of the packets are transferred to the controller. The controller analyzes the samples and identifies a flow as an elephant after it has seen sufficient number of samples from the flow. However, this approach can not reliably detect an elephant flow before it has carried more than 10K packets, or roughly 15 MB [87]. Additionally, sampling has high overhead, since the controller must process each sampled packet.

5.3 Our Solution: Mahout

Mahout's architecture is shown in Figure 5.1. In Mahout, a shim layer at each end-host monitors the flows originating from that host. When this layer detects an elephant flow, it marks subsequent packets of that flow using an in-band signaling mechanism. The switches in the network are configured to forward these marked packets to the Mahout controller. This simple approach allows the controller to detect elephant flows without placing burden on switches or using network bandwidth. The Mahout controller then manages only the elephant flows, to maintain a globally optimal arrangement of them.

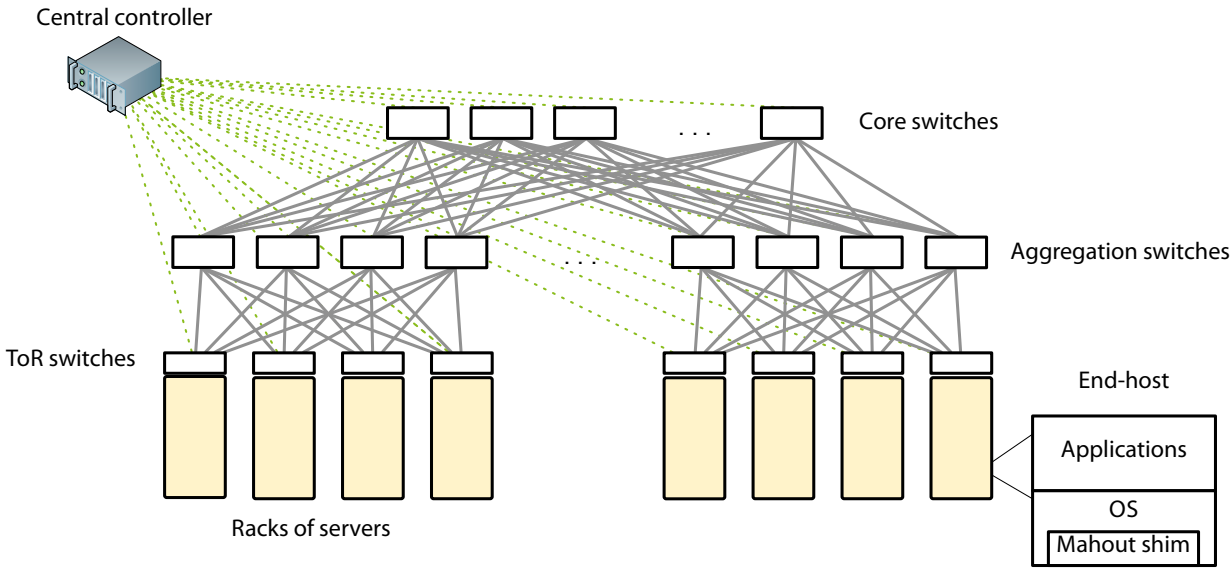


Figure 5.1: Mahout architecture.

Below, we describe Mahout's end-host shim layer for detecting elephant flows, our in-band signaling method to inform the controller about elephant flows, and the Mahout network controller.

5.3.1 Detecting Elephant Flows

An end-host based implementation for detecting elephant flows is better than in-network monitoring/sampling based methods, particularly in datacenters, because: (1) The network behavior of a flow is affected by how rapidly the end-point applications are generating data for the flow. Unlike in-network monitoring, the application's behavior is not biased by congestion in the network. (2) It is possible to augment the end-host OS. This is because datacenters usually are a single administrative domain and end-hosts run uniform software. (3) Mahout's elephant detection mechanism has very little overhead (it is implemented with two if statements) on commodity servers. In contrast, using an in-network mechanism to do fine-grained flow monitoring (such as OpenFlow's stat-pulling mechanism) can be infeasible, even on an edge switch, and even more so on a core switch, especially on commodity hardware. For example, assume that 32 servers are connected to a rack switch. If each server generates 20 new flows per second, with a default flow timeout period of 60 seconds,

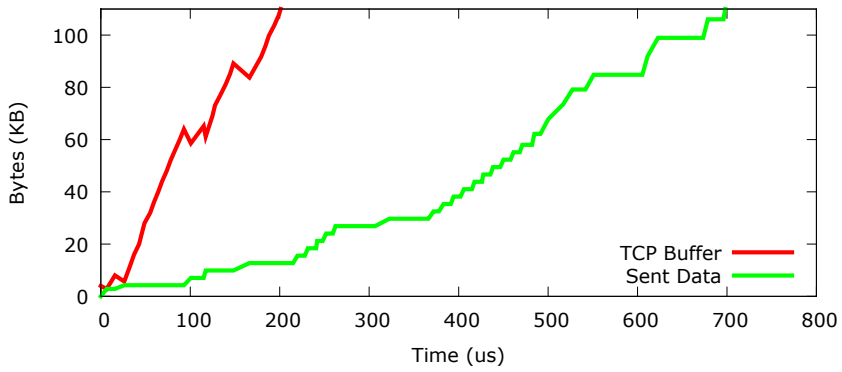


Figure 5.2: Amount of data observed in the TCP buffers vs. data observed at the network layer for a flow.

an edge-switch needs to maintain and monitor 38400 flow entries. This number is infeasible in any of the hardware switch implementations of OpenFlow that we are aware of.

A key idea of the Mahout system is to monitor end-host socket buffers, and thus determine elephant flows before in-network monitoring systems. We demonstrate the rationale for this approach with a micro-benchmark: an *ftp* transfer of a 50 MB file from a host 1 to host 2, connected via two switches all with 1 Gbps links.

In Figure 5.2, we show the cumulative amount of data observed on the network, and in the TCP buffer, as time progresses. The time axis starts when the application first provides data to the kernel. From the graph, one can observe that the application fills the TCP buffers at a rate much higher than the observed network rate. If the threshold for considering a flow as an elephant is 100KB (Figure 2. of [55] shows that more than 85% of flows are less than 100KB), we see that Mahout's end-host shim layer can detect a flow to be an elephant 3x sooner than in-network monitoring. In this experiment there were no other active flows on the network. In further experimental results, presented in Section 5.5, we observe an order of magnitude faster detection when there are other flows.

Mahout uses a shim layer in the end-hosts to monitor the socket buffers. When a socket buffer crosses a chosen threshold, the shim layer classifies the flow as an elephant. This simple approach

Algorithm 3 Pseudocode for end-host shim layer

```
1: When sending a packet
2: if number of bytes in buffer ≥ threshold_elephant then
3:    / * Elephant flow */
4:    if last-tagged-time - now() ≥ T_tagperiod then
5:       / * Set the differentiated services (DS) field to tag as elephant flow */
6:       set DS = 00001100
7:       last-tagged-time = now()
8:    end if
9: end if
```

is implemented by two if statements, as shown in Algorithm 3. It ensures that flows that are bottlenecked at the application layer and not in the network layer. This approach does not necessarily classify long-lived flows as elephants; instead, it identifies flows that are bottlenecked by the network. The intuition is that flows that are bottlenecked by the application need no special management in the network. In contrast, if an application is generating data for a flow faster than the flow's achieved network throughput, the socket buffer will fill up, and hence Mahout will detect this an an elephant flow.

5.3.2 In-band Signaling

Once Mahout's shim layer has detected an elephant flow, it needs to signal this to the network controller. We do this indirectly, by marking the packets in a way that is easily and efficiently detected by OpenFlow switches, and then the switches divert the marked packets to the network controller. To avoid inundating the controller with too many packets of the same flow, the end-host shim layer marks the packets of an elephant flow only once every $T_{tagperiod}$ seconds (we use 1 second in our prototype).

To mark a packet, we repurpose the Differentiated Services Field (DS Field) [96] in the IPv4 header. This field was originally called the IP Type-of-Service (IPToS) byte. The first 6 bits of the DS Field, called Differentiated Services Code Point (DSCP), define the per-hop behavior of a packet. The current OpenFlow specification [3] allows matching on DSCP bits, and most commercial switch implementations of OpenFlow support this feature in hardware; hence, we use the DS Field for signaling between the end-host shim layer and the network controller. Currently, the code point

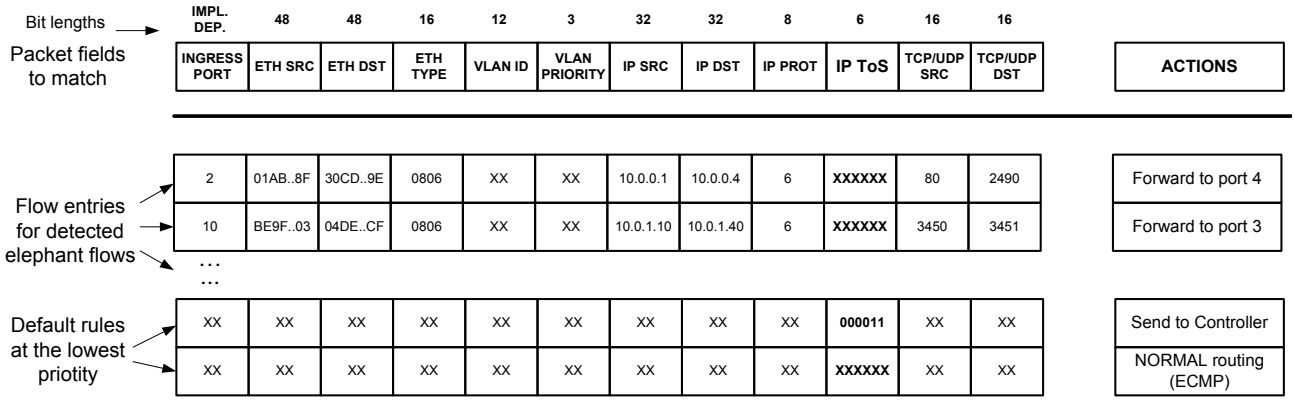


Figure 5.3: An example flow table setup at a switch by the Mahout controller.

space corresponding to $xxx11$ (x denotes a wild-card bit) is reserved for experimental or local usage [71], and we leverage this space. When an end-host detects an elephant flow, it sets the DSCP bits to 000011 in the packets belonging to that flow.

Algorithm 3 shows pseudocode for the end-host shim layer function that is executed when a TCP packet is being sent.

5.3.3 Mahout Controller

At each rack switch, the Mahout controller initially configures two default OpenFlow flow table entries: (i) an entry to send a copy of packets with the DSCP bits set to 000011 to the controller and (ii) the lowest-priority entry to switch packets using NORMAL forwarding action. We set up switches to perform ECMP forwarding by default in the NORMAL operation mode. Figure 5.3 shows the two default entries at the bottom. In this figure, an entry has a higher priority over (is matched before) entries drawn below that entry.

When a flow starts, it matches the lowest-priority (NORMAL) rule, so its packets are forwarded using ECMP. When an end-host classifies a flow as an elephant and marks a packet of that flow, the packet marked with DSCP 000011 matches the other default rule, and the rack switch forwards it to the Mahout controller. The controller then computes the best path for this elephant, and installs a flow-specific forwarding table entry in the switches along this path.

In Figure 5.3, we show a few example entries for the elephant flows. Note that these entries are installed with higher priority than Mahout's two default rules; hence, the packets corresponding to these elephant flows are switched using the actions of these flow-specific entries rather than the actions of the default entries. Also, the DS field is set to wildcard for these elephant flow entries, so that once the flow-specific rule is installed, any tagged packets from the end-hosts are not forwarded to the controller.

Once an elephant flow is reported to the Mahout controller, it needs to be placed on the best available path. We define the *best path for a flow* from s to t as the least congested of all paths from s to t. The least congested s-t path is found by enumerating over all such paths.

To manage the elephant flows, Mahout regularly pulls statistics on the elephant flows and link utilizations from the switches, and uses these statistics to optimize the elephant flows' routes. This is done with the increasing first fit algorithm given in Algorithm 4. Correa and Goemans introduced this algorithm and proved that it finds routings that have at most a 10% higher link utilization than the optimal routing [32]. While we cannot guarantee this bound because we re-route only the elephant flows, we expect this algorithm to perform well because of these theoretical results.

5.3.4 Discussion

DSCP bits In Mahout, the end-host shim layer uses the DSCP bits of the DS field in IP header for signaling elephant flows. However, there may be some datacenters where DSCP may be needed for other uses, such as for prioritization among different types of flows (voice, video, and data) or for prioritization among different customers. In such scenarios, we plan to use VLAN Priority Code Point (PCP) [2] bits for in-band signaling. OpenFlow supports matching on these bits too. As it is unlikely that both of these code point fields (PCP and DSCP) to be in use simultaneously, we expect one of these two options to be unused for most DCNs.

Virtualized Datacenter In a virtualized datacenter, a single server will host multiple guest virtual machines, each possibly running a different operating system. In such a scenario, we have two options. First, we could implement our elephant flow detection algorithm in the virtual switch in the hypervisor. This solution is ideal; however, it requires us to be able to modify the hypervisor. Second, the Mahout shim could be deployed in each of the guest virtual machines. Note that the host operating system will not have visibility into the socket buffers of a guest virtual machine. However,

Algorithm 4 Offline increasing first fit

```
1: sort(F); reverse(F) /* F: set of elephant flows */
2: for f ∈ F do
3:     for l ∈ f.path do
4:         l.load = l.load - f.rate
5:     end for
6: end for
7: for f ∈ F do
8:     best_paths[f].congest = ∞
9:     /* P_st: set of all s-t paths */
10:    for path ∈ P_st do
11:        congest = (f.rate + path.load) / path.bandwidth
12:        if congest < best_path.congest then
13:            best_paths[f] = path
14:            best_paths[f].congest = congest
15:        end if
16:    end for
17: end for
18: return best_paths
```

in cloud computing infrastructures such as Amazon EC2 [12], typically the infrastructure provider makes available a few preconfigured OS versions, which include the paravirtualization drivers to work with the provider's hypervisor. Thus, we believe that it is feasible to deploy the Mahout shim layer in virtualized datacenters, too.

Elephant flow threshold Choosing too low a value for $\texttt{threshold}_{elephant}$ in Algorithm 3 can cause many flows to be recognized as elephants, and hence cause the rack switches to forward too many packets to the controller. When there are many elephant flows, to avoid the controller overload, we could provide a means for the controller to signal the end-hosts to increase the threshold value. However, this would require a out-of-band control mechanism. An alternative is to use multiple DSCP values to denote different levels of thresholds. For example, $xxx11$ can be designated to denote that a flow has more than 100 KB data, $xx111$ to denote more than 1 MB, $x1111$ to denote more than 10 MB, and so on. The controller can then change the default entry corresponding to the tagged

packets (second from bottom in the Figure 5.3) to select higher thresholds, based on the load at the controller. Further study is needed to explore these approaches.

5.4 Analytical Evaluation

In this section, we analyze the expected overhead of detecting elephant flows with Mahout, with flow sampling, and by maintaining per-flow statistics (such as Hedera). We set up an analytical framework to evaluate the number of switch table entries and control messages used by each method. We evaluate each method using an example datacenter, and show that Mahout is the only solution that can scale to support large datacenters.

Flow sampling identifies elephants by sampling an expected 1 out of k packets. Once it has seen enough packets from the same flow, then the flow is classified as an elephant. The number of packets needed to classify an elephant does not affect our analysis in this section, so we ignore it for now. Hedera [8] uses periodic polling for elephant flow detection. Every t seconds, the Hedera controller pulls the per-flow statistics from each switch. In order to estimate the true rate of a flow (i.e., the rate of the flow if its rate is only constrained by its endpoints' NICs and not by any link in the network), the statistics for every flow in the network must be collected. Pulling statistics for all flows using OpenFlow requires setting up a flow table entry for every flow, so each flow must be sent to the controller before it can be started, so we include this cost in our analysis.

We consider a million end-host network for the following analysis. Here, an end-host could be a physical machine or a virtual machine. Our notation and and the assumed values are shown in the Table 5.1.

Hedera [8]: As table entries need to be maintained for all flows, the number of flow table entries needed at each rack switch is $T \cdot F \cdot D$. In our example, this translates to $32 \cdot 20 \cdot 60 = 38,400$ entries at each rack switch. We are not aware of any existing switch with OpenFlow support that can support this many entries in the flow table in the hardware—for example, HP ProCurve 5400zl switches support up to 1.7K OpenFlow entries per linecard. It is unlikely that any switch in the near future will support so many table entries given the expense of high-speed memory.

The Hedera controller needs to handle $N \cdot F$ flow setups per second, or more than 20 million requests per second in our example. A single NOX controller can handle only 30,000 requests per second [114]; hence one needs 667 controllers to just handle the flow setup load, assuming that the

Parameter	Description	Value
N	Num. of end-hosts	2^{20} (1M)
T	Num. of end-hosts per rack switch	32
S	Num. of rack switches	2^{15} (32K)
F	Avg. new flows per second per end-host	20 [114]
D	Avg. duration of a flow in the flow table	60 seconds
c	Size of counters in bytes	24 [3]
r_{stat}	Rate of gathering statistics	1-per-second
p	Num. of bytes in a packet	1500
f_m	Fraction of mice	0.99
f_e	Fraction of elephants	0.01
r_{sample}	Rate of sampling	1-in-1000
h_{sample}	Size of packet sample (bytes)	60

Table 5.1: Parameters and typical values for the analytical evaluation

load can be perfectly distributed.

The rate at which the controller needs to process the statistics packets is

$$= \frac{c \cdot T \cdot F \cdot D}{p} \cdot S \cdot r_{stat}$$

In our example, this implies $(24 \cdot 38400)/1500 \cdot 2^{15} \cdot 1 \approx 20.1M$ control packets per second. Assuming that NOX controller can handle these packets at the rate it can handle the flow setup requests (30,000 per second), this translates to needing 670 controllers just to process these packets. Or, if we consider only one controller, then the statistics can be gathered only once every 670 seconds (≈ 11 minutes).

Sampling: Sampling incurs the messaging overhead of taking samples, and then installs flow table entries when an elephant is detected. The rate at which the controller needs to process the sampled packets is

$$= \text{throughput} \cdot r_{sample} \cdot \frac{\text{bytes per sample}}{p}$$

We assume that each sample contains only a 60 byte header and that headers can be combined into 1500 byte packets, so there are 25 samples per message to the controller. The aggregate through-

put of a datacenter network changes frequently, but if 10% of the hosts are sending traffic, the aggregate throughput (in Gbps) is $0.10 \cdot N$. We then find the messaging overhead of sampling to be around 550K messages per second, or if we bundle samples into packets (i.e., 25 samples fit in a 1500 byte packet), then this drops to 22K messages per second.

At first blush, this messaging overhead does not seem like too much overhead; however, as the network utilization increases, the messaging overhead can reach 3.75 million (or 150K if there are 25 samples per packet) packets per second. Therefore, sampling incurs the highest overhead when load balancing is most needed. Decreasing the sampling rate reduces this overhead but adversely impacts the effects of flow scheduling since not all elephants are detected.

We expect the number of elephants identified by sampling to be similar to Mahout, so we do not analyze the flow table entry overhead of sampling separately.

Mahout: Because elephant flow detection is done at the end-host, switches contain flow table entries for elephant flows only. Also, statistics are only gathered for the elephant flows. So, the number of flow entries per rack switch in Mahout is $T \cdot F \cdot D \cdot f_e = 384$ entries. The number of flow setups that the Mahout controller needs to handle is $N \cdot F \cdot f_e$, which is about 200K requests per second, which needs 7 controllers. Also, the number of packets per second that need to be processed for gathering statistics is a f_e fraction of the same in case of Hedera. Thus 7 controllers are needed for gathering statistics at the rate of once per second, or the statistics can be gathered by a single controller at the rate of once every 7 seconds.

5.5 Experiments

5.5.1 Simulations

Our goal is to compare the performance and overheads of Mahout against the competing approaches described in the previous section. To do so, we implemented a flow-level, event-based simulator that can scale to a few thousand end-hosts connected using Clos topology [30]. We now describe this simulator and our evaluation of Mahout with it.

Methodology

We simulate a datacenter network by modeling the behavior of flows. The network topology is modeled as a capacitated, directed graph and forms a three-level Clos topology. All simulations here are of a 1,600 server datacenter network. The network has an agility of 0.20, which means that it has 320 Gb of bisection bandwidth. All servers have 1 Gbps NICs and links have 1 Gbps capacity. Our simulation is event-based, so there is no discrete clock—instead, the timing of events is accurate to floating-point precision. Input to the simulator is a file listing the start time, bytes, and endpoints of a set of flows (our workloads are described below). When a flow starts or completes, the rate of each flow is recomputed.

We model the OpenFlow protocol only by accounting for the delay when a switch sets up a flow table entry for a flow. When this occurs, the switch sends the flow to the OpenFlow controller by placing it in its OpenFlow queues. This queue has 10 Mbps of bandwidth (this number was measured from an OpenFlow switch [86]). This queue has infinite capacity, so our model optimistically estimates the delay between a switch and the OpenFlow controller since a real system drops arriving packets if one of these queues is full, resulting in TCP timeouts. Moreover, we assume that there is no other overhead when setting up a flow, so the OpenFlow controller deals with the flow and installs flow table entries instantly.

We simulate three different schedulers: (1) an offline scheduler that periodically pulls flow statistics from the switches, (2) a scheduler that behaves like the Mahout scheduler, but uses sampling to detect elephant flows, and (3) the Mahout scheduler as described in Sec. 5.3.3.

The stat-pulling controller behaves like Hedera [8] and Helios [47]. Here, the controller pulls flow statistics from each switch at regular intervals. The statistics from a flow table entry are 24 bytes, so the amount of time to transfer the statistics from a switch to the controller is proportional to the number of flow table entries at the switch. When transferring statistics, we assume that the CPU-to-controller rate is the bottleneck, not the network or OpenFlow controller itself. Once the controller has statistics for all flows, it computes a new routing for elephant flows and reassigns paths instantly. In practice, computing this routing and inserting updated flow table entries into the switches will take up to hundreds of milliseconds. We allow this to be done instantaneously to find the theoretical best achievable results using an offline approach. The global re-routing of flows is computed using the increasing best fit algorithm described in Algorithm 4. This algorithm is simpler than the simulated annealing employed by Hedera; however, we expect the results to be similar, since this heuristic is

likely to be as good as any other (as discussed in Sec. 5.3.3)

As we are doing flow-level simulations, sampling packets is not straightforward since there are no packets to sample from. Instead, we sample from flows by determining the amount of time it will take for k packets to traverse a link, given its rate, and then sample from the flows on the link by weighting each flow by its rate. To simulate packet sampling, we begin by calculating the amount of time it will take for s packets to traverse a link given the rate of flows on that link, where $1/s$ is the sampling rate and each packet is assumed to be 1500bytes. We find the time this occurs, and at that time, we sample from the flows traverse the link, weighting each flow by its rate. For example, if there are three flows on a link with rates $r_1, r_2,$ and r_3, then we sample from flow i with probability $(r_1 + r_2 + r_3)/r_i$. We found, however, that performing this sampling took too long when the sample rate was high (e.g., 1 out of 100 packets) because it requires over 1.4 million samples per second, and each sample takes time proportional to the number of flows on the link to obtain. Therefore, we did not perform sampling directly, and instead, precomputed the distribution of times needed to correctly identify a flow as an elephant for different sampling rates by performing the sampling on a single switch while running our traffic workload (described just below). So, when a new flow is started, we draw a time from this distribution, and if the flow is still active after that amount of time, we label it as an elephant.

Workloads We simulate background traffic modeled on recent measurements [76] and add traffic modeled on MapReduce traffic to stress the network. We assume that the MapReduce job has just gone into its shuffle phase. In this phase, each end-host transfers 128 MB to each other host. Each end-host opens a connection to at most five other end-hosts simultaneously (as done by default in Hadoop's implementation of MapReduce). Once one of these connections completes, the host opens a connection to another end-host, repeating this until it has transferred its 128 MB file to each other end-host. The order of these outgoing connections is randomized for each end-host. For all simulations described here, we used 250 randomly selected end-hosts in the shuffle load. The reduce phase shuffle begins three minutes after the background traffic is started to allow the background traffic to reach a steady state, and measurements shown here are taken for five minutes after the reduce phase began.

We added background traffic following the macroscopic flow measurements collected by Kandula et al. [55, 76] to the traffic mix because datacenters run a heterogeneous mix of services simultaneously. They give the fraction of correspondents a server has within its rack and outside of its

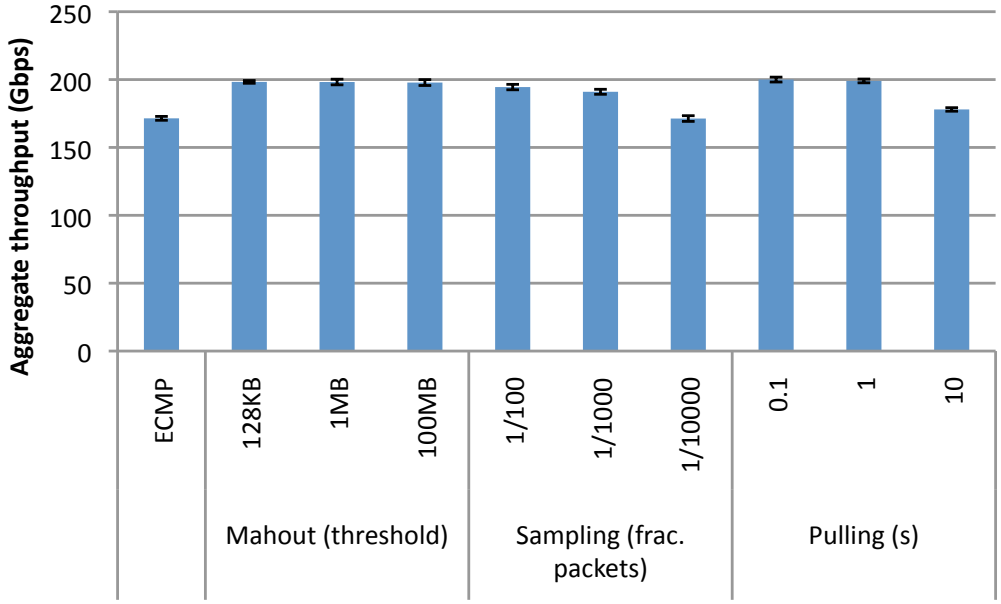


Figure 5.4: Throughput results for the schedulers with various parameters. Error bars on all charts show 95% confidence intervals.

rack over a ten second interval. We follow this distribution to decide how many inter- and intra-rack flows a server starts over ten seconds; however, they do not give a more detailed breakdown of flow destinations than this, so we assume that the selection of a destination host is uniformly random across the source server's rack or the remaining racks for an intra- or inter-rack flow respectively. We select the number of bytes in a flow following the distribution of flow sizes in their measurements as well. Before starting the shuffle job, we simulate this background traffic for three minutes. The simulation ends whenever the last shuffle job flow completes.

Metrics To measure the performance of each scheduler, we tracked the aggregate throughput of all flows. This the the sum of rates of all flows in the network. We measure overhead as before in Section 5.4, i.e., by counting the number of control messages and the number of flow table entries at each switch. All numbers shown here are averaged from ten runs.

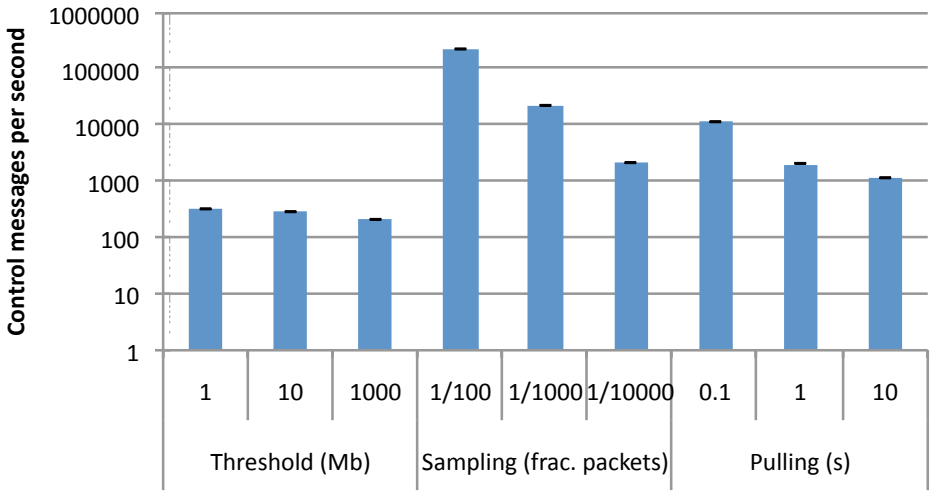


Figure 5.5: Number of packets sent to controller by various schedulers. Here, we bundled samples together into a single packet (there are 25 samples per packet)—each bundle of samples counts as a single controller message.

Results

The per-second aggregate throughput for the various scheduling methods is shown in Figure 5.4. We compare these schedulers to static load balancing with equal-cost multipath (ECMP), which uniformly randomizes the outgoing flows across a set of ports [8]. We used three different elephant thresholds for Mahout: 128 KB, 1 MB, and 100 MB, and flows carrying at least this threshold of bytes were classified as an elephant after sending 2, 20, or 2000 packets respectively. As expected, controlling elephant flows extracts more bisection bandwidth from the network—Mahout extracts 16% more bisection bandwidth from the network than ECMP and the other schedulers obtain similar results depending on their parameters.

Hedera's results found that flow scheduling gives a much larger improvement over ECMP than our results (up to 113% on some workloads) [8]. This is due to the differences in workloads. Our workload is based on measurements [76], whereas their workloads are synthetic. We have repeated our simulations using some of their workloads and find similar results: the schedulers improve throughput by more than 100% compared to ECMP on their workloads.

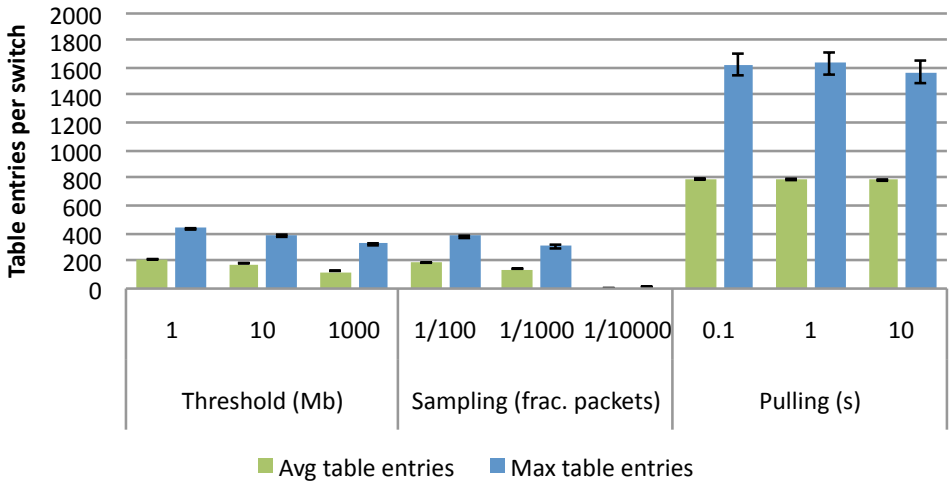


Figure 5.6: Average and maximum number of flow table entries at each switch used by the schedulers.

We examine the overhead versus performance tradeoff by counting the maximum number of flow table entries per rack switch and the number of messages to the controller. These results are shown in Figures 5.5 and 5.6

Mahout has the least overhead of any scheduling approach considered. Pulling statistics requires too many flow table entries per switch and sends too many packets to the controller to scale to large datacenters; here, the stat-pulling scheduler used nearly 800 flow table entries per rack switch on average no matter how frequently the statistics were pulled. This is more than seven times the number of entries used by the sampling and Mahout controllers, and makes the offline scheduler infeasible in larger datacenters because the flow tables will not be able to support such a large number of entries. Also, when pulling stats every 1 sec., the controller receives 10x more messages than when using Mahout with an elephant threshold of 100 MB.

These simulations indicate that, for our workload, the value of $\texttt{threshold}_{elephant}$ affects the overhead of the Mahout controller, but does not have much of an impact on performance (up to a point: when we set this threshold to 1 GB (not shown on the charts), the Mahout scheduler performed no better than ECMP). The number of packets to the Mahout controller goes from 328 per sec. when the elephant threshold is 128 KB to 214 per sec. when the threshold is 100 MB, indicating that tuning it can reduce controller overhead by more than 50% without affecting the scheduler's performance.

Even so, we suggest making this threshold as small as possible to save memory at the end-hosts and for quicker elephant flow detection (see the experiments on our prototype in the next section). We believe a threshold of 200–500 KB is best for most workloads.

5.5.2 Prototype & Microbenchmarks

We have implemented a prototype of the Mahout system; however, this work was primarily performed by Wonho Kim, so we omit the details here. They are published in reference [34]. The shim layer is implemented as a kernel module inserted between the TCP/IP stack and device driver, and the controller is built on NOX [114]. We evaluated our prototype, and overall, we showed that it correctly classifies elephant flows in under 2 ms, while it takes a Hedera-like controller at least 180 ms to detect elephant flows.

Chapter 6

Traffic Engineering with DevoFlow

6.1 Introduction

In the previous chapter, we showed how end-hosts could be leveraged to lower the overheads involved with traffic engineering in the datacenter. We now evaluate the effectiveness of using DevoFlow [38], an alternative approach to DCN traffic engineering. DevoFlow is a flow-based networking framework for cost-effective, scalable flow management that does not require end-host modifications.

Flow-based networking is based on a separation between the network's control-plane and its data-plane. The control-plane is implemented as a distributed system running on commodity servers. This controller orchestrates network flows by managing the state of switch flow tables. This model is very flexible, since software controllers can be quickly updated, unlike vertically integrated network devices. This approach to networking is called *software-defined networking* (SDN), because the behavior of the network is defined by software.

The OpenFlow protocol [3, 85] is a widely-deployed implementation of SDN. It uses a centralized controller that makes decisions on a per-flow basis. This model supports fine-grained, flow-level control of Ethernet switches. Such control is desirable because it enables (1) correct enforcement of flexible policies without carefully crafting switch-by-switch configurations, (2) visibility of all flows, allowing for near-optimal management of network traffic, and (3) simple and future-proof switch design.

DevoFlow [38] switch implementation mechanisms to help scale SDN applications. In this chapter, we demonstrate that high-performance flow management requires devolving control of most flows back to the switches, while the controller maintains control over only targeted *elephant flows*. DevoFlow is allows aggressive use of wild-carded OpenFlow rules—thus reducing the number of switch-controller interactions and the number of TCAM entries—through new mechanisms to efficiently detect elephant flows. DevoFlow also introduces mechanisms to allow switches to make local routing decisions, independent of the controller.

We evaluate the effectiveness of DevoFlow. Through simulations we find that it can load-balance datacenter traffic as well as fine-grained solutions, but with far less overhead: DevoFlow uses 10–53 times fewer flow table entries at an average switch, and uses 10–42 times fewer control messages.

DevoFlow overview

Our goal in designing DevoFlow is to enable cost-effective, scalable flow management. Our design principles are:

- *Keep flows in the data-plane* as much as possible. Involving the control-plane in all flow setups creates too many overheads in the controller, network, and switches.
- *Maintain enough visibility over network flows* for effective centralized flow management, but otherwise provide only aggregated flow statistics.
- *Simplify the design and implementation of fast switches* while retaining network programmability.

DevoFlow attempts to resolve two dilemmas — a control dilemma:

- Invoking the OpenFlow controller on every flow setup provides good start-of-flow visibility, but puts too much load on the control plane and adds too much setup delay to latency-sensitive traffic, and
- Aggressive use of OpenFlow flow-match wildcards or hash-based routing (such as equal-cost multipath (ECMP) routing) reduces control-plane load, but prevents the controller from effectively managing traffic.

and a statistics-gathering dilemma:

- Collecting OpenFlow counters on lots of flows, via the pull-based Read-State mechanism, can create too much control-plane load, and
- Aggregating counters over multiple flows via the wild-card mechanism may undermine the controller's ability to manage specific elephant flows.

We resolve these two dilemmas by pushing responsibility over most flows to switches and adding efficient statistics collection mechanisms to identify significant flows, which are the only flows managed by the central controller. This is described in Section 6.3.3.

Our work here derives from a long line of related work that aims to allow operators to specify high-level policies at a logically centralized controller, which are then enforced across the network without the headache of manually crafting switch-by-switch configurations [22, 24, 56, 58]. This separation between forwarding rules and policy allows for innovative and promising network management solutions such as NOX [58, 114] and other proposals [64, 95, 121], but these solutions may not be realizable on many networks because the flow-based networking platform they

are built on—OpenFlow—is not scalable. We made this observation in the previous chapter, and it has been made by others as well. However, other research has focused on scaling the controller, e.g., Onix [83], Maestro [23], and a devolved controller design [112]. We find that the controller can present a scalability problem, but that switches may be a greater scalability bottleneck. Removing this bottleneck requires minimal changes: slightly more functionality in switch ASICs and more efficient statistics-collection mechanisms.

Note that this chapter presents work that was performed while the author was an intern at HP Labs, Palo Alto. This work was collaborative with several others, so the results are sketched here, with detailed presentation only of the results where this dissertation's author was the primary contributor. The full version of this work can be found as reference [38].

6.2 OpenFlow Overheads

Flow-based networking involves the control-plane more frequently than traditional networking, and therefore the bandwidth and latency of communication between a switch and the central controller is a potential performance bottleneck. The latency imposed on traffic to the controller can be on the order of several milliseconds for DCNs and a full RTT for WANs. Flow-based network imposes overheads on switch implementation, which can be broken down into implementation-imposed and implementation-specific overheads. We sketch the overheads of OpenFlow here. The interested reader can find the full details in reference [38]. To explore them, we experimented with HP's OpenFlow implementation on the HP ProCurve 5406 zl switch [1]. This switch is designed with a central CPU for management functions and an ASIC on each line card. The implementation-imposed overheads of OpenFlow can be summarized as follows.

- **Flow setup overheads:** The bandwidth between the data- and control-planes of a switch and its controller has finite capacity. This can limit the rate of flow setups—the best implementations we know of can set up only a few hundred flows per second. To estimate the flow setup rate of the ProCurve 5406 zl, we attached two servers to the switch and opened a connection from one server to the other as soon as the previous connection was established. We found that the switch completes roughly 275 flow setups per second. This number is in line with what others have reported [109].

  However, this rate is insufficient for flow setup in a high-performance network. The median

inter-arrival time for flows at datacenter server is less than 30 ms [76], so we expect a rack of 40 servers to initiate approximately 1300 flows per second—far too many to send each flow to the controller.

- **Gathering flow statistics:** Global flow schedulers need timely access to statistics. If a few, long-lived flows constitute the majority of bytes transferred, then a scheduler need only collect flow statistics every several seconds; however, this is not the case in high-performance networks, where even the longest-lived flows last only a few seconds [76]. Overall, based on the expected and measured workloads of DCNs, we found that OpenFlow's current statistics-gathering mechanisms are not scalable.

  This is primarily because OpenFlow provides pull-based statistics, where counters for each flows are collected from the switches by the controller. This type of flow statistics can be used for flow scheduling if they can be collected frequently enough. The evaluation of one flow scheduler, Hedera [8], indicates that a 5 sec. control loop (the time to pull statistics from all access switches, compute a re-routing of elephant flows, and then update flow table entries where necessary) is fast enough for near-optimal load balancing on a fat-tree topology; however, their workload is based on flow lengths following a Pareto distribution. Recent measurement studies have shown datacenter flow sizes do not follow a Pareto distribution [17, 55]. Using a workload with flow lengths following the distribution of flow sizes measured in [55], we find that a 5 sec. statistics-gathering interval can improve utilization only 1–5% over randomized routing with ECMP (details are in §6.4). This is confirmed by Raiciu et al., who found that the Hedera control loop needs to be less than 500ms to perform better than ECMP on their workload [104].
- **Switch state size:** A limited number of flow entries can be supported in hardware. The 5406 zl switch hardware can support about 1500 OpenFlow rules, whereas the switch can support up to 64000 forwarding entries for standard Ethernet switching. One reason for this wide disparity is that OpenFlow rules are stored in a TCAM, necessary to support OpenFlow's wildcarding mechanism, and TCAM entries are an expensive resource, whereas Ethernet forwarding uses a simple hash lookup in a standard memory. It is certainly possible to increase the number of TCAM entries, but only at the expense of space, power, and money.

  Because OpenFlow rules are per-flow, rather than per-destination, each directly-connected host will typically require an order of magnitude more rules. Use of wildcards could reduce this ratio, but this is often undesirable as it reduces the ability to implement flow-level policies (such as multipathing) and flow-level visibility.

- **Controller overheads:** Involving the controller in all flows creates a potential scalability problem: any given controller instance can support only a limited number of flow setups per second. For example, Tavakoli *et al.* [114] report that one NOX controller can handle "at least 30K new flow installs per second while maintaining a sub-10 ms flow install time ... The controller's CPU is the bottleneck." Kandula *et al.* [76] found that 100K flows arrive every second on a 1500-server cluster, implying a need for multiple OpenFlow controllers.

  Recently, researchers have proposed more scalable OpenFlow controllers. Maestro [23] is a multi-threaded controller that can install about twice as many flows per second as NOX, without introducing additional latency. Others have worked on distributed implementations of the OpenFlow controller (also valuable for fault tolerance) These include HyperFlow (Tootoonchian and Ganjali [115]) and Onix (Koponen et al. [83]). These distributed controllers can only support global visibility of rare events such as link-state changes, and not of frequent events such as flow arrivals. As such, they are not yet suitable for applications, such as Hedera [8], which need a global view of flow statistics.

6.3 DevoFlow

We now sketch the design of DevoFlow, which avoids the overheads described above by introducing mechanisms for efficient devolved control (§6.3.1) and statistics collection (§6.3.2). Then, we end this section by discussing how to use DevoFlow to reduce use of the control-plane in traffic engineering (§6.3.3).

6.3.1 Mechanisms for devolving control

We introduce two new mechanisms for devolving control to a switch, *rule cloning* and *local actions*.

Rule cloning: Under standard OpenFlow, all packets matching a rule are treated as a single flow, so the controller is not invoked for each *microflow* arrival, where a microflow is an end-to-end stream that is uniquely identified by source IP address, destination IP address, transport protocol (such as TCP), source port number, and destination port number. To gain greater visibility over each microflow, rule cloning duplicates a wildcard rule for a microflow, and insert a copy into the exact match flow table, with the relevant details of the microflow filled out. This allows counters to be col-

lected on each microflow matching the wildcard rule.

Local actions: Certain flow-setup decisions might require decisions intermediate between the heavyweight "invoke the controller" and the lightweight "forward via this specific port" choices offered by standard OpenFlow. In DevoFlow, we envision rules augmented with a small set of possible "local routing actions" that a switch can take without paying the costs of invoking the controller. If a switch does not support an action, it defaults to invoking the controller, so as to preserve the desired semantics.

Examples of local actions include multipath support and rapid re-routing:

- **Multipath support** gives the switch a choice of several output ports for a wildcard forwarding rule. The switch can then select, randomly from a probability distribution or round-robin, between these ports on each microflow arrival. Note that the output port for a microflow remains fixed for the duration of that microflow. This keeps the microflow's packets on the same path to prevent out of order packet delivery.

  This functionality is similar to equal-cost multipath (ECMP) routing; however, multipath wildcard rules provide more flexibility. ECMP (1) uniformly selects an output port uniformly at random and (2) requires that the cost of the multiple forwarding paths to be equal, so it load balances traffic poorly on irregular topologies. As an example, consider a topology with two equal-cost links between s and t, but the first link forwards at 1 Gbps whereas the second has 10 Gbps capacity. ECMP splits flows evenly across these paths, which is clearly not ideal since one path has 10 times more bandwidth than the other.

  DevoFlow solves this problem by allowing a clonable wildcard rule to select an output port for a microflow according to some probability distribution. This allows implementation of *oblivious routing* (see, e.g., [45, 81]), where a microflow follows any of the available end-to-end paths according to a probability distribution. Oblivious routing would be optimal for our previous example, where it would route $10/11^{\text{th}}$ of the microflows for t on the 10 Gbps link and $1/11^{\text{th}}$ of them on the 1 Gbps link.
- **Rapid re-routing** allows switches to specify fallback rules if an output port fails.

6.3.2 Efficient statistics collection

DevoFlow provides three different ways to improve the efficiency of OpenFlow statistics collection.

Sampling: the sFlow protocol [4] allows a switch to report the headers of randomly chosen packets to a monitoring node—which could be the OpenFlow controller. This imposes almost no extra load on the network because only packet headers are sent to the monitoring node.

Triggers and reports: allows the controller to place push-based *triggers* on counters. When the counter reaches a specified threshold, a report is sent to the controller.

Approximate counters: can be maintained for all microflows that match a wildcard forwarding table rule. Such counters maintain approximate, space-efficient statistics for all microflows forwarded by this rule. Approximate counters can be implemented using streaming algorithms [46, 52, 53], which are generally simple, use very little memory, and identify the flows transferring the most bytes with high accuracy. For example, Golab et al.'s algorithm [53] correctly classifies 80–99% of the flows that transfer more than a threshold k of bytes. Implementing approximate counters in the ASIC is more difficult than DevoFlow's other mechanisms; however, they provide a more timely and accurate view of the network and can keep statistics on microflows without creating a table entry per microflow.

6.3.3 Using DevoFlow for flow scheduling

All existing OpenFlow applications work unmodified with the introduction of DevoFlow; however, DevoFlow enables scalable implementation of these solutions by reducing the number of flows that interact with the control-plane. Scalability relies on a finding a good definition of "significant flows" in a particular domain. These flows should represent a small fraction of the total flows, but should be sufficient to achieve the desired results. As an example, we show how to schedule flows for traffic engineering with DevoFlow.

Flow scheduling does not scale well if the scheduler relies on visibility over all flows, as is done in Hedera [8] because maintaining this visibility via the network is too costly, as discussed in §6.2 showed. Instead, we maintain visibility only over elephant flows, which is all that a system such as Hedera actually needs. While Hedera defines an elephant as a flow using at least 10% of a NIC's

bandwidth, we define one as a flow that has transferred at least a threshold number of bytes X. A reasonable value for X is 1–10MB.

Our solution starts by initially routing incoming flows using DevoFlow's multipath wildcard rules; this avoids involving the control-plane in flow setup. We then detect elephant flows as they reach X bytes transferred. We can do this using using any combination of DevoFlow's statistics collection mechanisms. For example, we can place triggers on flow table entries, which generate a report for a flow after it has transferred X bytes; We could also use sampling or approximate counters; we evaluate each approach in §6.4.

Once a flow is classified as an elephant, the detecting switch or the sampling framework reports it to the DevoFlow controller. The controller finds the least congested path between the flow's endpoints, and re-routes the flow by inserting table entries for the flow at switches on this path.

The new route can be chosen, for example, by the decreasing best-fit bin packing algorithm of Correa and Goemans [32]. The algorithm's inputs are the network topology, link utilizations, and the rates and endpoints of the elephant flows. Its output is a routing of all elephant flows. Correa and Goemans proved that their algorithms finds routings with link utilizations at most 10% higher than the optimal routing, under a traffic model where all flows can be rearranged. We cannot guarantee this bound, because we only rearrange elephant flows; however, their theoretical results indicates their algorithm will perform as well as any other heuristic for flow scheduling.

Finally, we note that this architecture uses only edge switches to encapsulate new flows to send to the central controller. The controller programs core and aggregation switches reactively to flow setups from the edge switches. Therefore, the only overhead imposed is cost of installing flow table entries at the the core and aggregation switches—no overheads are imposed for statistics-gathering.

6.4 Evaluation

In this section, we present our simulated evaluation of DevoFlow. We show that it achieves the same performance as fine-grained, OpenFlow-based flow scheduler, but with far less overhead.

Algorithm 5 — Flow rate computation.

Input: set of flows F and a set of ports $\mathcal{P}$
Output: a rate $r(f)$ of each flow $f \in F$

begin
Initialize: $F_a = \emptyset; \forall f, r(f) = 0$
Define: P.used() $= \sum_{f \in F_a \cap P} r(f)$
Define: P.unassigned_flows() $= P - (P \cap F_a)$
while $\mathcal{P} \neq \emptyset$ **do**
    Sort $\mathcal{P}$ in ascending order, where the sort key
        for P is (P.rate$-P$.used())$/|P$.unassigned_flows()$|$
    $P = \mathcal{P}$.pop_front()
    for each $f \in P$.unassigned_flows() **do**
        $r(f) = (P.\text{rate} - P.\text{used}())/|P.\text{unassigned\_flows}()|$
        $F_a = F_a \cup \{f\}$
end

6.4.1 Simulation methodology

To evaluate DevoFlow on a large-scale network, we implemented a flow-level datacenter network simulator. This fluid model captures the overheads generated by each flow and the coarse-grained behavior of flows in the network. The simulator is event-based, and whenever a flow is started, ended, or re-routed, the rate of all flows is recomputed using the algorithm shown in Algorithm 5. This algorithm works by assigning a rate to flows traversing the most-congested port, and then iterating to the next most-congested port until all flows have been assigned a rate.

We represent the network topology with a capacitated, directed graph. For these simulations, we used two topologies: a three-level Clos topology [30] and a two-dimensional HyperX topology [7]. In both topologies, all links were 1 Gbps, and 20 servers were attached to each access switch. The Clos topology has 80 access switches (each with 8 uplinks), 80 aggregation switches, and 8 core switches. The HyperX topology is two-dimensional and forms a 9×9 grid, and so has 81 access switches, each attached to 16 other switches. Since the Clos network has 8 core switches, it is 1:2.5

oversubscribed; that is, its bisection bandwidth is 640 Gbps. bandwidth. The HyperX topology is 1:4 oversubscribed and thus has 405 Gbps of bisection bandwidth.

The Clos network has 1600 servers and the HyperX network has 1620. We sized our networks this way for two reasons: first, so that the Clos and HyperX networks would have nearly the same number of servers. Second, our workload is based on the measurements of Kandula *et al.* [76], which are from a cluster of 1500 servers. We are not sure how to scale their measurements up to much larger data centers, so we kept the number of servers close to the number measured in their study.

We simulate the behavior of OpenFlow at switches by modeling (1) switch flow tables, and (2) the limited data-plane to control-plane bandwidth. Switch flow tables can contain both exact-match and wildcard table entries. For all simulations, table entries expire after 10 seconds. When a flow arrives that does not much a table entry, the header of its first packet is placed in the switch's data-plane to control-plane queue. The service rate for this queue follows our measurements described in Section 6.2, so it services packets at 17Mbps. This queue has finite length, and when it is full, any arriving flow that does not match a table entry is dropped. We experimented with different lengths for this queue, and we found that when it holds 1000 packets, no flow setups were dropped. When we set its limit to 100, we found that fewer than 0.01% of flow setups were dropped in the worst case. For all results shown in this paper, we set the length of this queue to 100; we restart rejected flows after a simulated TCP timeout of 300 ms.

Finally, because we are interested in modeling switch overheads, we do not simulate a bottleneck at the OpenFlow controller; the simulated OpenFlow controller processes all flows instantly. Also, whenever the OpenFlow controller re-routes a flow, it installs the flow-table entries without any latency.

Workloads

We consider two workloads in our simulations: (1) a MapReduce job that has just gone into its shuffle stage, and (2) a workload based on measurements, by Kandula et al. at Microsoft Research (MSR) [76], of a 1500-server cluster.

The MapReduce-style traffic is modeled by randomly selecting n servers to be part of the reduce-phase shuffle. Each of these servers transfers 128 MB to each other server, by maintaining connections to k other servers at once. Each server randomizes the order it connects to the other servers, keeping k connections open until it has sent its payload. All measurements we present for this shuffle

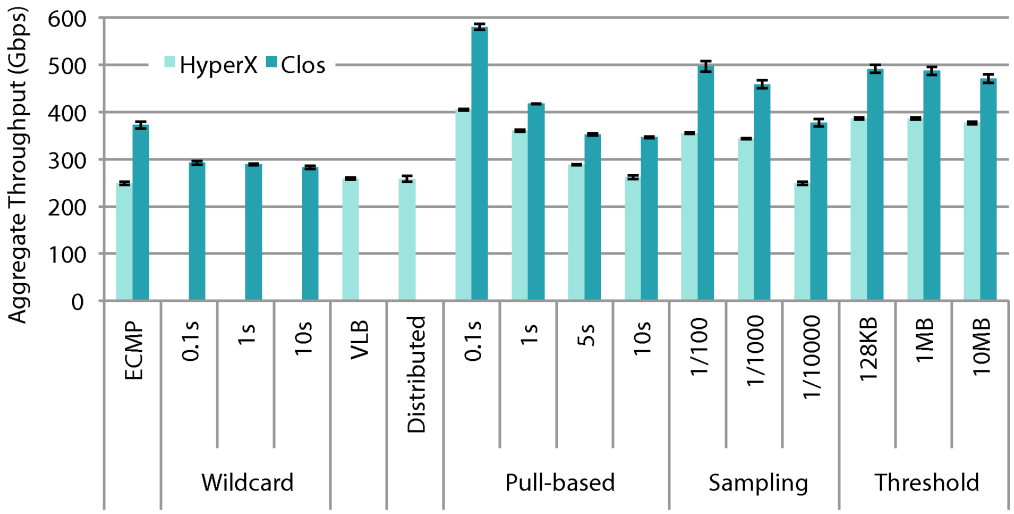


Figure 6.1: Throughput achieved by the schedulers for the shuffle workload with $n = 800$ and $k = 5$. OpenFlow-imposed overheads are not modeled in these simulations. All error bars in this paper show 95% confidence intervals for 10 runs.

workload are for a one-minute period that starts 10 sec. after the shuffle begins.

In our MSR workload, we generated flows based on the distributions of flow inter-arrival times and flow sizes in [76]. We attempted to reverse-engineer their actual workload from only two distributions in their paper. In particular, we did not model dependence between sets of servers. We pick the destination of a flow by first determining whether the flow is to be an inter- or intra-rack flow, and then selecting a destination uniformly at random between the possible servers. For these simulations, we generated flows for four minutes, and present measurements from the last minute.

Additionally, we simulated a workload that combines the MSR and shuffle workloads, by generating flows according to both workloads simultaneously. We generated three minutes of MSR flows before starting the shuffle. We present measurements for the first minute after the shuffle began.

Schedulers

We compare static routing with ECMP to flow scheduling with several schedulers.

The DevoFlow scheduler: behaves as described in Sec. 6.3, and collects statistics using either sampling or threshold triggers on multipath wildcard rules. The scheduler might re-reroute a flow after it has classified the flow as an elephant. New flows, before they become elephant flows, are routed using ECMP regardless of the mechanism to detect elephant flows. When the controller discovers an elephant flow, it installs flow-table entries at the switches on the least-congested path between the flow's endpoints. We model queueing of a flow between the data-plane and control-plane before it reaches the controller; however, we assume instantaneous computation at the controller and flow-table installations.

For elephant detection, we evaluate both sampling and triggers.

Our flow-level simulation does not simulate actual packets, which makes modeling of packet sampling non-trivial. In our approach:

1. We estimate the distribution of packets sent by a flow before it can be classified, with less than a 10% false-positive rate, as an elephant flow, using the approach described by Mori et al. [87].
2. Once a flow begins, we use that distribution to select how many packets it will transfer before being classified as an elephant; we assume that all packets are 1500 bytes. We then create an event to report the flow to the controller once it has transferred this number of packets.

Finally, we assume that the switch bundles 25 packet headers into a single report packet before sending the samples to the controller; this reduces the packet traffic without adding significant delay. Bundling packets this way adds latency to the arrival of samples at the controller. For our simulations, we did not impose a time-out this delay. We bundled samples from all ports on a switch, so when a 1 Gbps port is the only active port (and assuming it's fully loaded), this bundling could add up to 16 sec. of delay until a sample reaches the controller, when the sample rate is 1/1000 packets.

Fine-grained control using statistics pulling: simulates using OpenFlow in active mode. Every flow is set up at the central controller and the controller regularly pulls statistics, which it uses to schedule flows so as to maximize throughput. As with the DevoFlow scheduler, we route elephant flows using Correa and Goeman's bin-packing algorithm [32]. Here, we use Hedera's definition of an elephant flow: one with a demand is at least 10% of the NIC rate [8]. The rate of each flow is found using Algorithm 5 on an ideal network; that is, each access switch has an infinite-capacity uplink to a single, non-blocking core switch. This allows us to estimate the demand of each flow when flow rates are constrained only by server NICs and not by the switching fabric.

Following the OpenFlow standard, each flow table entry provides 88 bytes of statistics [3]. We

collect statistics only from the access switches. The ASIC transfers statistics to the controller at 17 Mbps, via 1500-byte packets. The controller applies the bin-packing algorithm immediately upon receiving a statistics report, and instantaneous installs a globally optimized routing for all flows.

Wildcard routing: performs multipath load balancing possible using only wildcard table entries. This controller reactively installs wildcard rules to create a unique spanning tree per destination: all flows destined to a server are routed along a spanning tree. When a flow is set up, the controller computes the least-congested path from the switch that registered the flow to the flow's destination's spanning tree, and installs the rules along this path. We simulated wildcard routing only on the Clos topology, because we are still developing the spanning tree algorithm for HyperX networks.

Valiant load balancing (VLB): balances traffic by routing each flow through an intermediate switch chosen uniformly at random; that switch then routes the flow on the shortest path to its destination [117]. On a Clos topology, ECMP implements VLB.

Distributed greedy routing: routes each flow by first greedily selecting the least-congested next-hop from the access switch, and then using shortest-path routing. We simulate this distributed routing scheme only on HyperX networks.

6.4.2 Performance

We begin by assessing the performance of the schedulers, using the aggregate throughput of all flows in the network as our metric. Figure 6.1 shows the performance of the schedulers under various settings, on a shuffle workload with $n = 800$ servers and $k = 5$ simultaneous connections/server. This simulation did *not* model the OpenFlow-imposed overheads; for example, the 100ms pull-based scheduler obtains all flow statistics every 100ms, regardless of the switch load.

We see that DevoFlow can improve throughput compared to ECMP by up to 32% on the Clos network and up to 55% on the HyperX network. The scheduler with the best performance on both networks is the pull-based scheduler when it re-routes flows every 100 ms. This is not entirely surprising, since this scheduler also has the highest overhead. Interestingly, VLB did not perform any better than ECMP on the HyperX network.

To study the effect of the workload on these results, we tried several values for n and k in the shuffle workload and we varied the fraction of traffic that remained within a rack on the MSR workload. These results are shown in Figure 6.2 for the Clos topology and Figure 6.3 for the HyperX network. Overall, we found that flow scheduling improves throughput for the shuffle workloads, even when

the network has far more bisection bandwidth than the job demands.

For instance, with $n = 200$ servers, the maximum demand is 200 Gbps. Even though the Clos network has 640 Gbps of bisection bandwidth, we find that DevoFlow can increase performance of this shuffle by 29% over ECMP. We also observe that there was little difference in performance when we varied k.

Flow scheduling did not improve the throughput of the MSR workload. For this workload, regardless of the mix of inter- and intra-rack traffic, we found that ECMP achieves 90% of the optimal throughput[1] for this workload, so there is little room for improvement by scheduling flows. We suspect that a better model than our reverse-engineered distributions of the MSR workload would yield different results.

Because of this limitation, we simulated a combination of the MSR workload with a shuffle job. Here, we see improvements in throughput due to flow scheduling; however, the gains are less than when the shuffle job is ran in isolation.

6.4.3 Overheads

We used the MSR workload to evaluate the overhead of each approach because, even though we do not model the dependence between servers, we believe it gives a good indication of the rate of flow initiation. Figure 6.4 shows, for each scheduler, the rate of packets sent to the controller while simulating the MSR workload.

Load at the controller should scale proportionally to the number of servers in the datacenter. Therefore, when using an OpenFlow-style pull-based scheduler that collects stats every 100ms, in a large datacenter with 160K servers, we would expect a load of about 2.9M packets/sec., based on extrapolation from Figure 6.4. This would drop to 775K packets/sec. if stats are pulled once per second. We are not aware of any OpenFlow controller that can handle this message rate; for example, NOX can process 30K flow setups per second [114]. A distributed controller might be able to handle this load (which would require up to 98 NOX controllers, assuming they can be perfectly distributed and that statistics are pulled every 100 ms), but it might be difficult to coordinate so many controllers.

Figure 6.5 shows the number of flow table entries at any given access switch, for the MSR workload and various schedulers. For these simulations, we timed out the table entries after 10 sec. As expected, DevoFlow does not require many table entries, since it uses a single wildcard rule for all mice

[1] We found the optimal throughput by attaching all servers to a single non-blocking switch.

flows, and stores only exact-match entries for elephant flows. This does, however, assume support for the multipath routing wildcard rules of DevoFlow. If rule cloning were used instead, DevoFlow would use the same number of table entries as the pull-based OpenFlow scheduler because it would clone a rule for each flow. The pull-based scheduler uses an order of magnitude more table entries, on average, than DevoFlow.

We estimated the amount bandwidth required between a switch's data-plane and control-plane when statistics are collected with a pull-based mechanism. Figure 6.6 shows the bandwidth needed so that the 95^{th} and 99^{th} percentile flow setup latencies on the MSR workload are less than 2ms. Here, we assume that the only latency incurred is in the queue between the switch's data-plane and control-plane; we ignore any latency added by communication with the controller. That is, the figure shows the service rate needed for this queue, in order to maintain a waiting time of less than 2 ms in the 95^{th} and 99^{th} percentiles. The data to control-plane bandwidth sufficient for flow setup is directly proportional to this deadline, so a tighter deadline of 1 ms needs twice as much bandwidth to meet.

The scale on the right of the chart normalizes the required data-to-control-plane bandwidth to a switch's total forwarding rate (which in our case is 28 Gbps, because each ToR switch has 28 gigabit ports). For fine-grained (100 ms) flow management using OpenFlow, this bandwidth requirement would be up to 0.7% of its total forwarding rate. Assuming that the amount of control-plane bandwidth needed scales with the forwarding rate, a 144-port 10 Gbps switch needs just over 10 Gbps of control-plane bandwidth to support fine-grained flow management. We do not believe it is cost-effective to provide so much bandwidth, so DevoFlow's statistics-collection mechanisms are the better option because they are handled entirely within the data-plane.

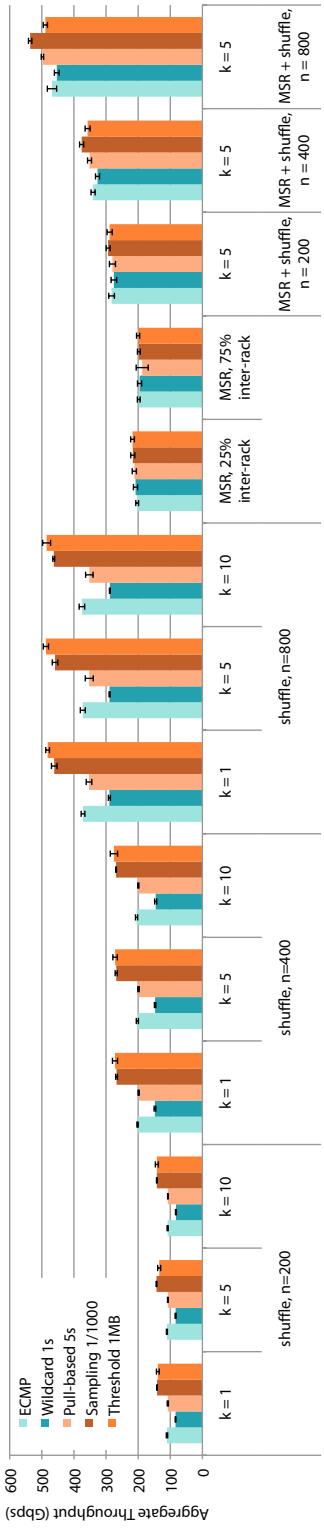


Figure 6.2: Aggregate throughput of the schedulers on the Clos network for different workloads. For the MSR plus shuffle workloads, 75% of the MSR workload-generated flows are inter-rack.

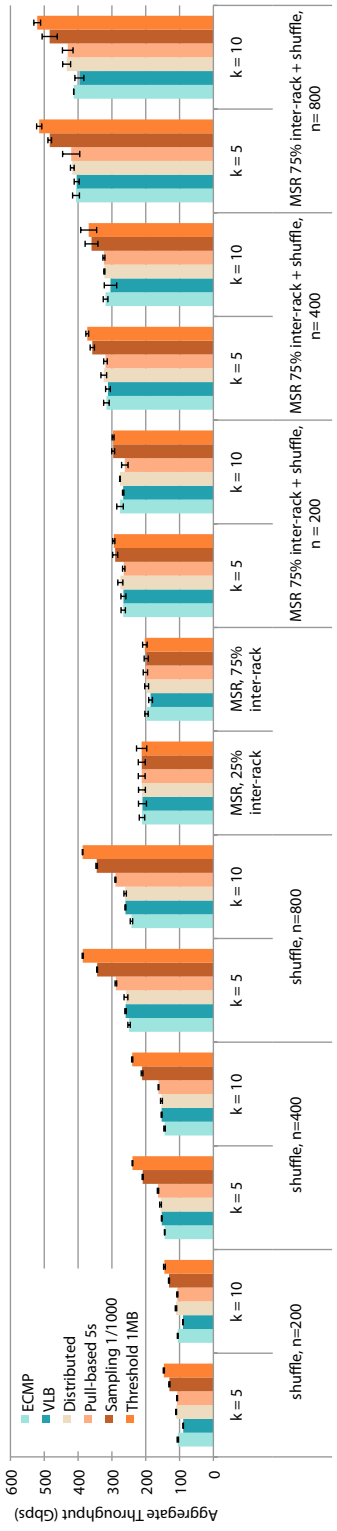


Figure 6.3: Aggregate throughput of the schedulers on the HyperX network for different workloads.

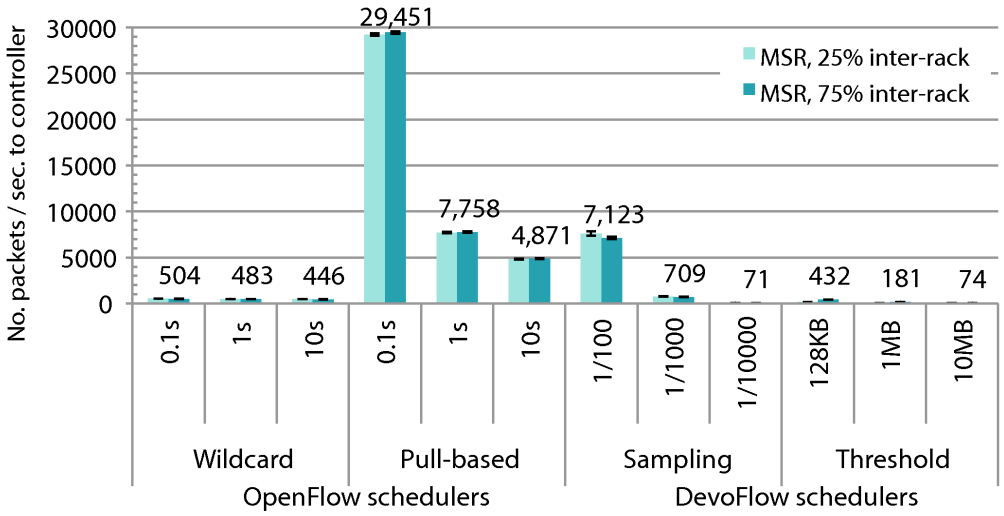


Figure 6.4: The number of packet arrivals per second at the controller using the different schedulers on the MSR workload.

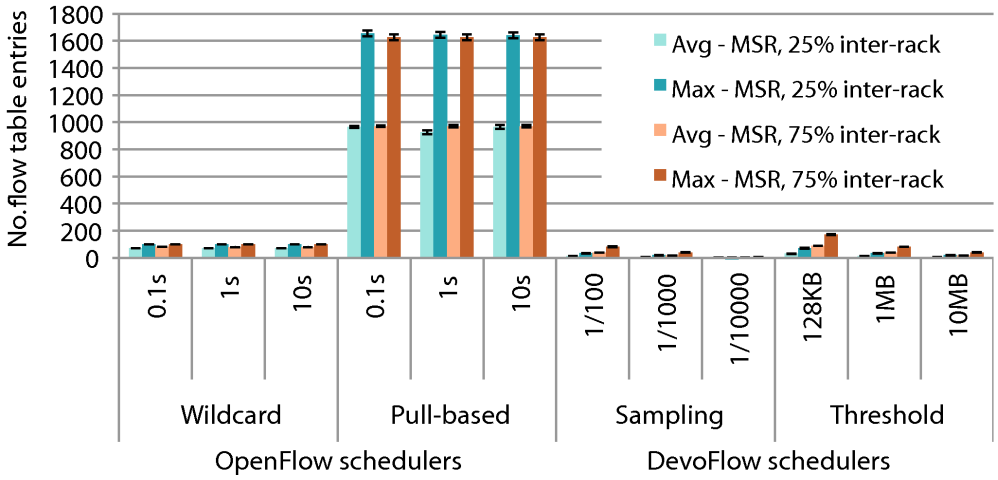


Figure 6.5: The average and maximum number of flow table entries at an access switch for the schedulers using the MSR workload.

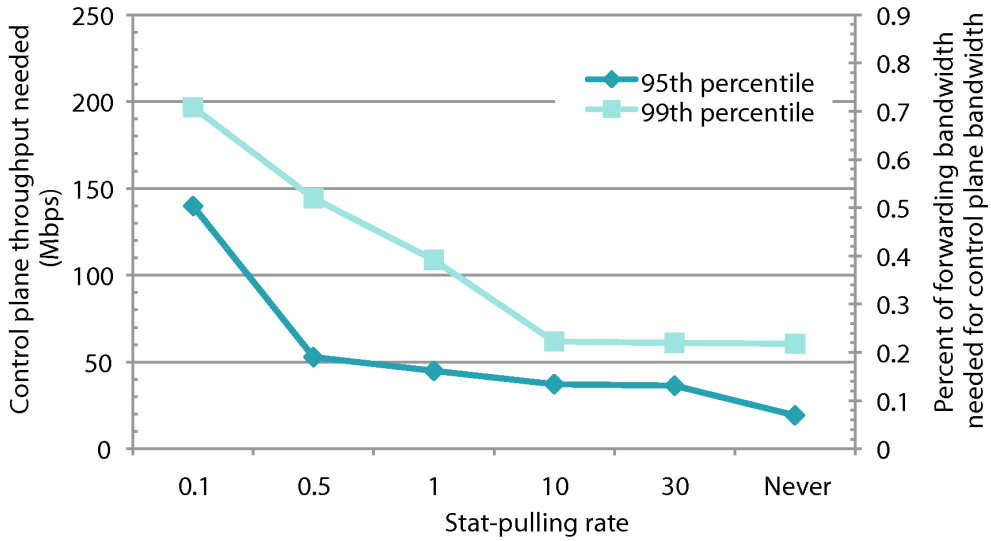


Figure 6.6: The control-plane bandwidth needed to pull statistics at various rates so that flow setup latency is less than 2ms in the 95th and 99th percentiles. Error bars are too small to be seen.

Chapter 7

Conclusions

Conclusions

Large-scale datacenters are now an integral part of the Internet. They host services that billions of people depend on in their daily lives, power the world's financial markets, and store our collective human history. This new "computer" is different than previous computer architectures. Because it uses distributed nodes for computation and storage, the network is needed to share state and data between these nodes.

In this dissertation, we showed that the network topology can make a significant impact on the cost of a datacenter. We found theory and algorithms to help operators design cost-effective, heterogeneous topologies. We proposed the heterogeneous Clos topology and found an algorithm to compute agility on arbitrary topologies. We designed, implemented, and evaluated two datacenter design frameworks: LEGUP and REWIRE. By generalizing the definition of a Clos network, LEGUP can reduce the cost of DCN upgrades and expansions by a factor of 2. And by using unstructured topologies, REWIRE is able to design DCNs with only 10% of the cost of previous solutions in some scenarios. Because capital expenditure is the bulk of the cost of a DCN, these tools make it possible to significantly reduce the cost of operating a datacenter network.

To achieve these results, REWIRE relies on *unstructured* networks, rather than the topology-constrained networks most existing datacenter use. REWIRE demonstrates that arbitrary topologies can boost DCN performance while reducing network equipment expenditure. Traditionally DCN design has restricted the topology to only a few classes of topologies, because it is difficult to operate an arbitrary topology in a high-performance environment. These difficulties have been mitigated by recent work [27, 88, 103], so it may be time to move away from highly regular DCN topologies because of the performance benefits arbitrary topologies offer.

We also showed that dynamic load balancing can improve the performance of a DCN. Even though a Clos topology is well-suited to randomized load balancing, we found that the aggregate throughput of some workloads can be increased 40% by dynamically scheduling flows. However, dynamic load balancing is challenging in DCNs, because of their scale and workloads. Therefore, it is important to identify and manage elephant flows to keep the load balancing problem tractable. The previous approaches for elephant flow detection are based on monitoring the behavior of flows in the network and hence incur long detection times, high switch resource usage, or high control bandwidth and processing overhead. In contrast, we proposed and evaluated a low-overhead end-host-based mechanism for elephant flow detection. Our datacenter traffic management system Ma-

hout is based on this idea, and as a result, it incurs an order of magnitude lower controller overhead than other approaches. Finally, we demonstrated that DevoFlow can also reduce the overheads of DCN traffic engineering by at least an order of magnitude without adversely impacting performance. Unlike Mahout, the DevoFlow approach does not need to modify end-hosts, because it adds mechanisms to switches to lower the overheads of centralize flow management.

While we have addressed several challenges in this dissertation, several challenges remain open. Broadly, we believe that the most important DCN challenges are to: (1) Design and implement an agile, scalable, flexible, resilient, and manageable network for future datacenters. (2) Accurately model datacenter capex and opex. Cabling and management seem to be the big unknowns here. (3) Understand the requirements of and design inter-datacenter networks.

More specifically, the following are steps that could be taken to extend the results of this dissertation.

Datacenter design

- Fully understand datacenter design and expansion. Develop a holistic view of datacenter growth. This should incorporate the cost of servers, electricity, network, and storage. Ideally, the model should incorporate trends in device prices.
- Introduce an accurate cost model for links. For example, our cost model did not take into account "bundling" of links, where a group of links is bundled together, reducing the cost of installing any single link in the bundle.
- Design an optimization framework to design any Clos network configuration. Such a framework would extend Mudigonda et al.'s framework that finds 3-level, homogeneous Clos topologies [89]. More generally, we believe it would be interesting to incorporate output topology constraints into REWIRE's optimization algorithm. This would allow operators to constrain REWIRE's output to a family of topologies (e.g., Clos, fat-tree , BCube or HyperX).
- Improve the definition of network flexibility used by LEGUP. The current definition is expensive to compute and it is unclear if our metric accurately captures flexibility.
- Improve the definition of network reliability used by LEGUP and REWIRE. Ideally, this metric should capture the agility of the network after a failure.
- Determine the gap between optimal and LEGUP's results as a network is incrementally expanded. Further, the gap between LEGUP and the optimal network constructed with the same

budget is unknown.

- Find a connection between the expansion properties (such as its spectral gap) of a network and its bisection bandwidth or agility.
- Extend REWIRE's simulated annealing algorithm so that it can design topologies from a class of topologies (for example, any BCube topology).
- Improve REWIRE so that is is scalable and less computationally expensive. One idea to speed computation is to implement dynamic all-pairs shortest-path (APSP) algorithms (such as [49, 40], which speed up APSP computations performed on a graph with dynamic edge weights.

DCN management

- Even on a regular topology such as a Clos network, debugging network problems is extremely difficult. Ideally, a solution to this challenge is not dependent on topology.
- Determine the performance gains from dynamic flow scheduling with real DCN workloads. With Mahout and DevoFlow, we reverse-engineered a workload from published measurements.
- Build other applications on top of DevoFlow. For example, it may be possible to perform, scalable quality of service (QoS), multicast, routing-as-a-service [26], network virtualization [109], and energy-aware routing [64] with DevoFlow.

These are just a few of the open challenges that remain to fully understand datacenter networks. Datacenter-scale computing is still in its infancy, and much work remains to make this new computer easier and less expensive to build, operate, and manage.

References

[1] HP ProCurve 5400 zl switch series. http://h17007.www1.hp.com/us/en/products/switches/HP_E5400_zl_Switch_Series/index.aspx.

[2] IEEE Std. 802.1Q-2005, Virtual Bridged Local Area Networks.

[3] OpenFlow Switch Specification, Version 1.0.0. http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf.

[4] sFlow. http://www.sflow.org/.

[5] The OpenFlow Switch Consortium. http://www.openflowswitch.org/.

[6] *The Hadoop Distributed File System*. IEEE, 2010.

[7] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, 2009.

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

[9] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *NSDI*, 2010.

[10] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

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

[12] `http://aws.amazon.com/ec2/`.

[13] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using mantri. In *OSDI*, 2010.

[14] Martin F. Arlitt and Carey L. Williamson. Web server workload characterization: the search for invariants. In *SIGMETRICS*, 1996.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven H, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[16] V. E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffics*. Academic Press, 1965.

[17] T. Benson, Aditya Akella, and David Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[18] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. In *Proceedings of the 1st ACM workshop on Research on enterprise networking (WREN)*, 2009.

[19] Dimitris Bertsimas and John Tsitsiklis. Simulated annealing. *Statistical Science*, 8(1):10–15, 1993.

[20] R. Braden, D. Clark, and S. Shenker. Integrated service in the internet architecture: an overview. Technical report, IETF, Network Working Group, June 1994.

[21] Aydin Buluç, John R. Gilbert, and Ceren Budak. Solving path problems on the GPU. *Parallel Comput.*, 36:241–253, June 2010.

[22] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *NSDI*, 2005.

[23] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. Maestro: A System for Scalable OpenFlow Control. Tech. Rep. TR10-08, Rice University, 2010.

[24] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *SIGCOMM*, pages 1–12, Aug. 2007.

[25] Chandra Chekuri. Routing and network design with robustness to changing or uncertain traffic demands. *SIGACT News*, 38(3):106–129, September 2007.

[26] Chao-Chih Chen, Lihua Yuan, Albert Greenberg, Chen-Nee Chuah, and Prasant Mohapatra. Routing-as-a-service (RaaS): A framework for tenant-directed route control in data center. In *INFOCOM*, 2011.

[27] Kai Chen, Chuanxiong Guo, Haitao Wu, Jing Yuan, Zhenqian Feng, Yan Chen, Songwu Lu, and Wenfei Wu. Generic and automatic address configuration for data center networks. In *SIGCOMM*, 2010.

[28] Fan R. K. Chung. *Spectral Graph Theory*. 1994.

[29] Cisco. Cisco data center network architecture and solutions overview. White paper, 2006. Available online (19 pages).

[30] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(5):406–424, 1953.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[32] José R. Correa and Michel X. Goemans. Improved bounds on nonblocking 3-stage clos networks. *SIAM J. Comput.*, 37(3):870–894, 2007.

[33] Paolo Costa, Austin Donnelly, Greg O'Shea, and Antony Rowstron. CamCube: A key-based data center. Technical report, Microsoft Research, MSR TR-2010-74, 2010.

[34] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM*, 2011.

[35] Andrew R. Curtis, Tommy Carpenter, Mustafa Elsheikh, Alejandro López-Ortiz, and S. Keshav. Rewire: An optimization-based framework for data center network design. In *INFOCOM*, 2012.

[36] Andrew R. Curtis, S. Keshav, and Alejandro López-Ortiz. LEGUP: Using heterogeneity to reduce the cost of data center network upgrades. In *CoNEXT*, 2010.

[37] Andrew R. Curtis and Alejandro López-Ortiz. Capacity provisioning a Valiant load-balanced network. In *INFOCOM*, 2009.

[38] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: scaling flow management for high-performance networks. In *SIGCOMM*, 2011.

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

[40] Camil Demetrescu and Giuseppe F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, November 2004.

[41] Digital Reality Trust. what is driving the US market? White paper, 2011. Available online at `http://knowledge.digitalrealtytrust.com/`.

[42] Luca Donetti, Franco Neri, and Miguel A. Munoz. Optimal network topologies: Expanders, cages, ramanujan graphs, entangled networks and all that. *Journal of Statistical Mechanics: Theory and Experiment*, 8, 2006.

[43] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.

[44] Leah Epstein and Asaf Levin. An APTAS for generalized cost variable-sized bin packing. *SIAM J. Comput.*, 38(1):411–428, 2008.

[45] Thomas Erlebach and Maurice Rüegg. Optimal bandwidth reservation in hose-model VPNs with multi-path routing. In *IEEE INFOCOM*, 2004.

[46] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, 2002.

[47] 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 *SIGCOMM*, 2010.

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

[49] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *J. Algorithms*, 34(2):251–281, February 2000.

[50] Naveen Garg and Jochen Koenemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In *FOCS*, 1998.

[51] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[52] Phillip B. Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*, 1998.

[53] Lukasz Golab, David DeHaan, Erik D. Demaine, Alejandro Lopez-Ortiz, and J. Ian Munro. Identifying frequent items in sliding windows over on-line packet streams. In *IMC*, 2003.

[54] Navin Goyal, Neil Olver, and F B. Shepherd. The vpn conjecture is true. In *STOC*, 2008.

[55] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kimand P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[56] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. *SIGCOMM CCR*, 35:41–54, Oct. 2005.

[57] Albert G. Greenberg, James R. Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *Computer Communication Review*, 39(1):68–73, 2009.

[58] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, MartĂ-n Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. In *SIGCOMM CCR*, July 2008.

[59] 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 *SIGCOMM*, 2009.

[60] 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 *SIGCOMM*, 2008.

[61] Hadoop MapReduce. http://hadoop.apache.org/mapreduce/.

[62] James R. Hamilton. Data center networks are in my way. Presented at the Standford Clean Slate CTO Summit, 2009.

[63] James R. Hamilton. Cloud computing is driving infrastructure innovation. Presented at Amazon Technology Open House Keynote, 2011.

[64] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: saving energy in data center networks. In *NSDI*, 2010.

[65] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986.

[66] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[67] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992 (Informational), November 2000.

[68] Daniel Horn. *Stream Reduction Operations for GPGPU Applications*, volume 2, chapter 36. 2005.

[69] HP POD. http://h18000.www1.hp.com/products/servers/solutions/datacentersolutions/pod/.

[70] F. K. Hwang and Dana S. Richards. Steiner tree problems. *Networks*, 22(1):55–89, 1992.

[71] IANA DSCP registry. http://www.iana.org/assignments/dscp-registry.

[72] IBM. IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[73] IBM Scalable modular data center. http://www-935.ibm.com/services/us/index.wss/offering/its/a1025610.

[74] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[75] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. on Comput.*, 3(4):299–325, 1974.

[76] S. Kandula, S. Sengupta, A. Greenberg, and P. Patel. The nature of datacenter traffic: Measurements & analysis. In *IMC*, 2009.

[77] Changhoon Kim, Matthew Caesar, and Jennifer Rexford. Floodless in seattle: a scalable ethernet architecture for large enterprises. In *SIGCOMM*, 2008.

[78] John Kim, William J. Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. *SIGARCH Comput. Archit. News*, 35(2), 2007.

[79] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[80] M. Kodialam, T. V. Lakshman, and S. Sengupta. Efficient and robust routing of highly variable traffic. In *Third Workshop on Hot Topics in Networks (HotNets-III)*, 2004.

[81] M. Kodialam, T. V. Lakshman, and S. Sengupta. Maximum throughput routing of traffic in the hose model. In *Infocom*, 2006.

[82] Murali Kodialam, T. V. Lakshman, James B. Orlin, and Sudipta Sengupta. Oblivious routing of highly variable traffic in service overlays and IP backbones. *IEEE/ACM Trans. Netw.*, 17:459–472, April 2009.

[83] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *OSDI*, 2010.

[84] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, 1985.

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

[86] Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, Andrew R. Curtis, and Sujata Banerjee. Devoflow: Cost-effective flow management for high performance enterprise networks. In *HotNets*, 2010.

[87] Tatsuya Mori, Masato Uchida, Ryoichi Kawahara, Jianping Pan, and Shigeki Goto. Identifying elephant flows through periodically sampled packets. In *Proc. IMC*, pages 115–120, Taormina, Oct. 2004.

[88] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, and Jeffrey C. Mogul. SPAIN: COTS data-center ethernet for multipathing over arbitrary topologies. In *NSDI*, 2010.

[89] Jayaram Mudigonda, Praveen Yalagandula, and Jeffrey C. Mogul. Taming the flying cable monster: A topology design and optimization framework for data-center networks. In *USENIX ATC*, 2011.

[90] Radhika N. Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, and Vikram Subram. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[91] H. Nagamochi. *Studies on Multicommodity Flows in Directed Networks*. PhD thesis, Department of Applied Mathematics and Physics, Kyoto University„ 1988.

[92] Hiroshi Nagamochi and Toshihide Ibaraki. Max-flow min-cut theorem for the multicommodity flows in certain planar directed networks. *Electronics and Communications in Japan, Part 3*, 72(3):58–71, 1989.

[93] Hiroshi Nagamochi and Toshihide Ibaraki. On max-flow min-cut and integral flow properties for multicommodity flows in directed networks. *Information Processing Letters*, 31:279–285, 1989.

[94] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proc. ANCS*, pages 1–9, 2008.

[95] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: Dynamic Access Control for Enterprise Networks. In *Proc. WREN*, pages 11–18, Aug. 2009.

[96] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), December 1998.

[97] Y. Nourani and B. Andresen. A comparison of simulated annealing cooling strategies. *Journal of Physics A: Mathematical and General*, pages 8373–8385, 1998.

[98] NVIDIA. NVIDIA CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.

[99] Sabine R. Öhring, Maximilian Ibel, Sajal K. Das, and Mohan J. Kumar. On generalized fat trees. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 37–, Washington, DC, USA, 1995. IEEE Computer Society.

[100] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[101] Yuval Peres, Dmitry Sotnikov, Benny Sudakov, and Uri Zwick. All-pairs shortest paths in $o(n^2)$ time with high probability. In *FOCS '10*, 2010.

[102] Lucian Popa, Sylvia Ratnasamy, Gianluca Iannaccone, Arvind Krishnamurthy, and Ion Stoica. A cost comparison of data center network architectures. In *CoNEXT*, 2010.

[103] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *SIGCOMM*, 2011.

[104] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handley. Data center networking with multipath tcp. In *Hotnets*, 2010.

[105] April Rasala and Gordon Wilfong. Strictly non-blocking wdm cross-connects for heterogeneous networks. In *STOC*, 2000.

[106] Matthew Roughan, Subhabrata Sen, Oliver Spatscheck, and Nick Duffield. Class-of-service mapping for QoS: A statistical signature-based approach to IP traffic classification. In *IMC*, pages 135–148, 2004.

[107] M. R. Samatham and D. K. Pradhan. The de bruijn multiprocessor network: A versatile parallel processing and sorting network for vlsi. *IEEE Trans. Comput.*, 38:567–581, April 1989.

[108] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. Presentation at the O'Reilly Velocity Web Performance and Operations Conference, 2009.

[109] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martín Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In *OSDI*, 2010.

[110] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *NSDI*, 2012.

[111] Sun's Modular Datacenter. http://www.sun.com/service/sunmd/.

[112] Adrian S.-W. Tam, Kang Xi, and H. Jonathan Chao. Use of Devolved Controllers in Data Center Networks. In *INFOCOM Workshop on Cloud Computing*, 2011.

[113] Hongsuda Tangmunarunkit, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Network topology generators: degree-based vs. structural. In *SIGCOMM*, 2002.

[114] Arsalan Tavakoli, Martin Casado, Teemu Koponen, and Scott Shenker. Applying NOX to the datacenter. In *HotNets*, 2009.

[115] Amin Tootoonchian and Yashar Ganjali. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proc. INM/WREN*, San Jose, CA, Apr. 2010.

[116] J. Touch and R. Perlman. Transparent Interconnection of Lots of Links (TRILL): Problem and Applicability Statement. RFC 5556 (Informational), May 2009.

[117] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *STOC*, 1981.

[118] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and effective fine-grained tcp retransmissions for datacenter communication. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 303–314, New York, NY, USA, 2009. ACM.

[119] Adam Vierra. Case study: NetRiver rethinks data center infrastructure design. *Focus Magazine*, August 2010. Available online at http://bit.ly/oS4pCu.

[120] VMware. http://www.vmware.com.

[121] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. In *Hot-ICE*, 2011.

[122] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast congestion control for TCP in data center networks. In *CoNEXT*, 2010.

[123] Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, and Yongguang Zhang. MDCube: a high performance network structure for modular data center interconnection. In *CoNEXT*, 2009.

[124] Xen. http://www.xen.org.

[125] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[126] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[127] R. Zhang-Shen and N. McKeown. Designing a predictable internet backbone with Valiant load-balancing. In *Thirteenth International Workshop on Quality of Service (IWQoS '05)*, 2005.

[128] Albert Y. Zomaya and Rick Kazman. Algorithms and theory of computation handbook. chapter Simulated annealing techniques, pages 33–33. Chapman & Hall/CRC, 2010.