Traffic Locality as an Opportunity in the Data Center

Thèse N° 9604

Présentée le 5 juillet 2019

à la Faculté informatique et communications
Laboratoire d'architecture des réseaux
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Jonas FIETZ

Acceptée sur proposition du jury

Prof. A. Ailamaki, présidente du jury
Prof. A. Argyraki, Prof. E. Bugnion, directeurs de thèse
Dr C. Minkenberg, rapporteur
Prof. A. Singla, rapporteur
Prof. J. Larus, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Acknowledgements

First, I would like to thank my advisors, Katerina Argyraki and Edouard Bugnion. They provided the means to work on exciting research topics in a very productive work environment. As a team, the two of them enabled me attack problems from a multitude of angles, and their different backgrounds made them a great combination. They were also always supportive and on the look-out for their students.

Katerina has taught me a great deal on how to approach problems with her uncanny ability to bring order to chaos. With this, she has often helped me to do bring structure to my thoughts and ideas, and to communicate my solutions more clearly. Her diverse research interests across a vast set of areas is highly impressive, and allowed her to provide the big picture view, and to pivot and reorient when necessary. At the same time, she is also a great teacher, who has made a lasting impression in me.

Ed's fundamental knowledge in the lower layers of computer architecture as well as his background in the commercial world enabled him to provide insights that only few advisors are able to give. He knows when to ask the critical questions, challenge me to provide numbers to back up speculative solutions, or when to take a step back to get a different perspective on a project.

I would also like to thank the members of my thesis committee, Prof. Anastasia Ailamaki, Prof. James Larus, Dr. Cyriel Minkenberg, and Prof. Ankit Singla. They graciously agreed to take the time to lend me their expertise, to serve in my committee, and to provide valuable feedback.

EPFL and particularly the systems group have been a source of great inspiration, research ideas and collaboration as well as new friendships. Being part of two labs (NAL & DCSL) and frequently having joint meetings with VLSC meant that I got to know many talented people, both professionally and privately. I would like to thank Adrien, Marios, Mia, Dmitrii, Sam, Stanko, George, David, Stuart, Sahand, Luis, Georgia, Arseniy, Pavlos, Dimitri, Mihai and Ovidiu for the many times they lent me an ear and helped out. We shared many other unforgettable moments throughout the years. Much of the same is true for the other PhD students with whom I started on this journey: Amer, Giel, Matej, Victor, Adrian, Catherine, Julia, and Manos, as well as many other friends made along the way.

Additional thanks goes out to my friends in Germany, Switzerland and elsewhere in the world. Knowing that there is a world outside of academia has kept me sane at times. In particular, I would like to thank Tobias for taking the time to read my whole thesis (and the papers that came before) as well as providing valuable feedback for it. Isabel was also a great help in polishing the thesis as well as motivating me in times of need.

Of course, I would like to thank my parents, who have supported me during my many years of studies. It was to no small degree their influence that has brought me to this point in my life, and enabled me to succeed.

Last but certainly not least, I would like to thank Judith. Since the beginning of this journey, she has stood by me, provided support and motivation, helped me keep some work/life balance, and was a frequent copy-editor, all the while suffering from my irregular working hours and tendency to underestimate efforts. This thesis would not have been possible without her.

Lausanne, 20th June 2019 Jonas Fietz

Abstract

In large scale data centers, network infrastructure is becoming a major cost component; as a result, operators are trying to reduce expenses, and in particular lower the amount of hardware needed to achieve their performance goals (or to improve the performance achieved for a given amount of hardware).

In this thesis, we explore the use of locality in the data center to meet these demands. In particular, we leverage locality in two different projects: (1) VNToR, which moves network virtualization from the server to the top-of-rack (ToR) switch, thereby reducing the server hardware needed to achieve a certain performance, and (2) Criss-Cross, which makes the network topology reconfigurable, thereby reducing the network hardware needed to switch typical data center workloads with a given level of performance.

VNToR exploits the locality of traffic flows as well as their power-law distribution in the design of a virtual flow table, which extends the hardware flow table of off-the-shelf top-of-rack switches. VNToR uses this virtual flow table in a hybrid data plane which merges both hardware as well software components. This way it can (1) store tens or even hundreds of thousands of flows, (2) adapt to traffic-pattern changes, typically in less than one millisecond, and (3) uses only commodity switching hardware with a minimal amount of data path memory (4) without compromising latency or throughput.

Criss-Cross is a hierarchical, reconfigurable topology for large-scale data centers. The locality in rack-level flows allows Criss-Cross to adjust its topology to the current traffic patterns. We show that Criss-Cross preserves many of the advantages of Clos topologies: (1) it maintains their hierarchy, (2) the simple routing algorithms, (3) their regular layout of connections for simple physical deployability, and (4) the compatibility to existing management approaches. We demonstrate that for a group-based communication pattern, Criss-Cross improves the average flow completion time by 5.5× and the 99th percentile by 6.3×. For a purely random point-to-point traffic pattern, it improves the flow completion time by 2.2× on average and 3× at the 99th percentile.

Keywords: network, data center, cloud, virtualization, network virtualization, optical circuit switch, routing

Zusammenfassung

In großen Rechenzentren wird die Infrastruktur für das Netzwerk immer mehr zu einem der wesentlichen Kostenfaktoren. Daher versuchen die Betreiber Ausgaben zu senken, indem sie beispielsweise die erforderliche Menge Hardware zur Erreichung eines bestimmten Leistungszielwerts verringern (oder alternativ, indem sie die Leistungsfähigkeit des Netzwerks bei gleich bleibendem Hardware-Budget erhöhen).

Diese Doktorarbeit untersucht, ob man örtliche und zeitliche Nähe (Lokalitätsprinzip) im Netzwerk ausnutzen kann, um Einsparpotentiale auszuschöpfen. Konkret wenden wir das Lokalitätsprinzip in zwei verschiedenen Projekten an: (1) VNToR verschiebt die Netzwerkvirtualisierung vom Server in den Top-of-Rack-Switch (ToR), was die Anzahl der benötigten Server bei gleicher Leistungsfähigkeit reduziert. (2) Criss-Cross ergänzt die Netzwerktopologie um eine rekonfigurierbare Komponente und macht sie damit anpassungsfähig. Dadurch wird weniger Netzwerk-Hardware für die typischen Arbeitslasten im Rechenzentrum benötigt.

VNToR macht sich die Lokalität einzelner 5-Tuple-Flows beim Entwurf einer virtuellen Flow-Tabelle zunutze, die die begrenzten Hardware-Flow-Tabellen von handelsüblichen ToRs erweitert. VNToR verwendet diese virtuelle Flow-Tabelle in einer hybriden Data-Plane, die Hardware- und Software-Komponenten kombiniert. Mit diesem Design kann VNToR (1) zehn- oder sogar hunderttausende Flows speichern und sich (2) an veränderte Muster in der Netzwerkauslastung anpassen – in der Regel innerhalb einer Millisekunde. Zum Einsatz kommen dabei (3) nur handelsübliche Switche mit minimalen Hardwaretabellen und es werden (4) weder Latenzen noch die verfügbare Bandbreite eingeschränkt.

Criss-Cross ist eine hierarchische, rekonfigurierbare Topologie für besonders große Rechenzentren. Criss-Cross nutzt die Lokalitätseigenschaften von Flows auf Rackebene und passt die Topologie an die jeweils aktuellen Netzwerklastmuster an. Dabei können wir zeigen, dass Criss-Cross viele Vorteile der Standard-Clos-Topologien wahrt: (1) den hierarchischen Aufbau, (2) die Nutzung simpler, deterministischer Routing-Algorithmen, (3) ein unverändertes, regelmäßiges Kabellayout für einfache, physische Installation, und (4) die Vereinbarkeit mit bestehenden Managementverfahren. Die Messungen für gruppenbasierte Kommunikation ergeben eine durchschnittliche Laufzeitverkürzung der Flows um den Faktor 5,5 und im 99-ten Perzentil sogar um den Faktor 6,3. Für zufällige Punkt-zu-Punkt-basierte Kommunikationsmuster verbessern sich die Laufzeiten um den Faktor 2,2 bzw. 3.

Stichwörter: Netzwerk, Netzwerkvirtualisierung, Rechenzentrum, Cloud, Routing

Résumé

Dans les larges centres de données, l'infrastructure réseau devient l'un des éléments majeurs en matière de coût. Les opérateurs essayent de réduire ces dépenses, notamment en réduisant la quantité de hardware nécessaire pour satisfaire leurs objectifs en terme de performance (ou, plutôt, pour améliorer la performance en utilisant les ressources déjà disponibles). Dans cette thèse, nous explorons comment le principe de localité peut être exploité afin de remplir ces objectifs. En particulier, nous utilisons le principe de localité dans deux projets : 1) VNTOR, qui qui étend la virtualisation du serveur au top-of-rack switch, réduisant la quantité de hardware nécessaire pour atteindre une performance donnée, et (2) CRISS-CROSS, qui rend la topologie de réseau reconfigurable, réduisant ainsi le hardware nécessaire au changement de workloads pour un certain objectif de performance.

VNTOR s'appuie sur la localité des flows réseaux (traffic flows) ainsi que sur leurs "long-tail behavior" afin d'obtenir une virtual flow table, une extension des hardware flow tables pour les top-of-rack switches. VNTOR utilise cette virtual flow table dans un data plane hybrid qui combine des éléments de hardware et de software. Ainsi, VNTOR peut (1) stocker des dizaines, voir des centaines, de milliers de règles d'accès, (2) s'adapter au changement de pattern du traffic, généralement en moins d'une milliseconde, et (3) nous n'utilisons que du commodity switching hardware avec des ressources mémoires minimal pour le data path (4) sans compromettre ni latence, ni débit. CRISS-CROSS est une topologie hiérarchique reconfigurable pour les grands centres de données. La localité des flows observée au niveau des racks permet à CRISS-CROSS d'adapter sa topologie au pattern de traffic. Nous montrons que cette solution préserve la plupart des avantages des topologies de Clos : (1) CRISS-CROSS maintient ces hiérarchies, (2) les algorithmes de routage simples, (3) les connexions et le câblage existants et (4) la compatibilité avec les approches de gestion du centre de données existantes.

Nous montrons que pour des patterns suivants des communications de groupes, CRISS-CROSS améliore le temps de complétion pour le flow moyen par un facteur de 5.5x, et le 99ième percentile par 6.3x. Pour un traffic complètement aléatoire, ces nombres sont respectivement 2.2x et 3x.

Mots clefs : Réseaux, centre de données, cloud, virtualisation, virtualisation de réseaux, routage

Contents

List of Figures

List of Tables

1 Introduction

Large scale data centers are becoming increasingly common, as more and more businesses migrate IT services into the cloud. These businesses are starting to use cloud services for all aspects of their daily work, from word editing to machine learning. As a result, the number of hyper-scale data centers is expected to double by 2021 compared to the 2016 levels, and 53% of all installed data center servers are expected to be in hyper-scale type data centers by then [26]. With up to 94% of all data center workloads running in cloud data centers, traffic within the data centers is estimated to quadruple in the same timespan [26]. Due to the capital requirements for these large data centers, it is a small number of companies that provide the majority of these cloud services.

The cost of the network makes up for a significant fraction of the cost of each data center, both in capital as well as operating expenses [11, 66, 86]. It is estimated that by 2021, 85% of the traffic in the data center will be east-west traffic [26], *i.e.*, traffic within the data center, and only 15% north-south, *i.e.*, traffic exiting towards the Internet or WAN. Naturally, as the traffic within the data center is growing, so is the importance of the data center network. Besides increases in bandwidth demands, requirements for latencies are also growing. As services become more distributed [38], the target service level objectives (SLOs) for latencies are decreasing. Both of these reasons cause the cost of networks to rise even further.

In the highly competitive cloud computing market, a key factor is maximizing the possible performance while minimizing the amount of hardware needed (and therefore cost). In 2017 alone, over $75 billion were spent on new data centers [136], and estimates show that these levels were probably exceeded by over 50% in 2018 [136]. As such, operators are trying to reduce expenses. For their capital expenses, they are attempting to reduce the amount of hardware needed to achieve their performance goals. Any reduction in the number of

deployed hardware devices also translates into savings in operating expenses, *e.g.*, energy consumption, or management overhead.

There has been a significant amount of work in recent years studying data center traffic under diverse sets of circumstances [12–14,44,47,57,80,89,125]. This work has shown that data center network traffic has inherent temporal and spatial locality, with its exact degree depending on the granularity of flows. This locality opens the door to significant optimizations of data center architecture at all levels.

In this thesis, we explore the use of locality in the data center to reduce the amount of data center hardware needed to achieve a certain level of performance (or to improve the performance achieved for a given amount of hardware). In particular, we leverage locality to:

1. move network virtualization from the server to the top-of-rack (ToR) switch, thereby reducing the server hardware needed to achieve given performance targets,
2. and make network topology reconfigurable, thereby reducing the network hardware needed to switch typical data center workloads with a given level of performance.

The first part of this thesis concerns the virtualization of tenant networks in a cloud environment. Cloud providers virtualize their networks to offer their tenants familiar network abstractions like layer-2 broadcast domains, private IP subnets, and security groups, completely under the tenants' administrative control. These features make it easy for tenants to replicate their physical network organization in the cloud and manage it with the same familiar processes.

Most often, cloud providers implement the virtual network abstractions on the server within the operating system (OS) that hosts the tenant virtual machines (VMs) or containers. The host OS uses virtual switching, for example Open vSwitch [118], to provide connectivity to the local VMs and also opportunistically includes cloud network abstractions: it performs the encapsulation or translation needed to provide the abstractions of a layer-2 network and IP subnet, and it enforces the access rules dictated by security groups.

This approach is flexible and convenient: firstly, it does not require any changes to network devices, which are typically hard to reprogram. Secondly, it allows easy coordination between the deployment of virtual-compute and virtual-network resources as both are deployed through agents running in the same place — the host OS.

However, this approach leads to unnecessary performance overheads. Compute virtualization platforms are designed to avoid host OS involvement as much as possible, and locating

the implementation within the host OS violates this principle. Moreover, this approach is incompatible with bare-metal computing, because to guarantee security, the point of implementation for these network abstractions needs to be at an entity outside the one being governed. Lastly, the network itself becomes vulnerable in the case of a hypervisor breakout. Bugs in hypervisor implementations allow malicious or compromised tenants to escape the confines of their VMs and run commands at the hypervisor level. As the enforcement of network policies is at the hypervisor level, such an escaped attacker could access the network without any protection.

To tackle these problems, we propose to offload the implementation of network abstractions to the top-of-rack switch with VNTOR (Virtual Network at the top-of-rack). This provides a solution to all problems mentioned above: we utilize the locality of traffic to move heavy flows into hardware, while serving the majority of traffic via a software data plane. This allows us to improve performance, reduce overheads at the end-host, and increase security.

The second part of the thesis concerns network topologies and their ability to adapt to the workload. While the research community has suggested a large variety of different topologies [9,24,25,50,51,57,58,65,92,94,100,121,133,134,137,140,141,145], the most common topology in use today is the (folded) Clos topology [1, 61, 125, 131]. Folded Clos topologies provide modularity and a hierarchical structure, which allows for easy calculation of routes, cabling and installation. An example of such a topology can be seen in Figure 2.5.

Given current data center size requirements and radix limitations, these Clos topologies have at least three layers: at the lowest layer, the servers of the data center are combined into racks, each connected to a ToR; ToR switches (and their servers) are grouped into collections called *pods*. These pods are connected to a set of fabric switches that make up the second layer; the fabric switches are then connected to spine switches that make up the top layer.

As described above, operators are trying to limit the cost of the data center and the network. To balance cost with scale, data center networks commonly have to oversubscribe the traffic at the fabric switches. For example, with an oversubscription of 1:4, there is 3× more bandwidth between ToRs and fabric switches than there is between fabric switches and spines [7, 131]. This is the oversubscription Facebook uses in their fabric-based Altoona data centers [7]. The amount of oversubscription is a strategic decision made at deployment time and commits the data center for multiple years, well before the actual networking requirements of the future workloads are known. Oversubscription, however, means that there is less bandwidth between pods than within pods [50, 82], which is problematic for traffic patterns with significant inter-pod traffic, *i.e.*, the general east-west traffic becoming more common in data centers today.

As a consequence, the design challenge for the second part of the thesis is the following: can we eliminate the spine bottlenecks that result from oversubscription while maintaining the modularity and hierarchy of a Clos topology? As a solution, we propose CRISS-CROSS, an alternative topology that utilizes reconfigurability to take advantage of locality in rack-level flows. While the above-mentioned standard topologies are static and symmetric in nature, the traffic itself is dynamic and non-uniform. We therefore propose a topology that preserves the advantages of the Clos topology, but that is reconfigurable to adjust to changing traffic patterns. This is an alternative approach to prior work which has often worked with direct connections as shortcuts to increase bandwidth between active racks or servers, or used non-regular structures to achieve shorter paths [57, 65, 145]. CRISS-CROSS uses optical circuit switches (OCS) to extend Clos topologies and rearrange the links between ToRs and fabric switches dynamically. Links are reconfigured in a way so that large flows bypass the spine connections and instead take shorter routes. Preserving the hierarchy in the topology allows CRISS-CROSS to preserve the simplicity of route calculation of the standard Clos based topologies, their path diversity, as well as the physical deployment layout in the data center.

1.1 Thesis Statement

This thesis demonstrates the feasibility of using the locality of network traffic to improve data-center performance and potentially reduce operating costs. By leveraging the locality of 5-tuple flows, we move the implementation of network virtualization abstractions from the end-hosts to already available hardware at the top of rack; this improves performance and frees up end-host resources. By leveraging the locality of rack-level flows, we continuously adapt the network topology to the workload; this improves performance and keeps the simplicity of the routing schemes implemented in current data centers.

1.2 Thesis Contributions

This thesis makes two main contributions to the area of data center networking. Using the locality of traffic flows, we are able to innovate both at the edge as well as on the topology itself. These innovations provide data center operators with increased performance, and potentially reduced costs.

VNTOR — Implementing virtual networking abstractions at the top-of-rack switch: The locality of traffic flows as well as their long-tailed behavior allows the design of a virtual flow table extending the hardware flow table of off-the-shelf top-of-rack switches. VNTOR uses a

hybrid data plane that consists of both a hardware as well as a software data plane. With this it can:

1. store tens or even hundreds of thousands of access rules,
2. adapt to traffic-pattern changes, typically in less than one millisecond,
3. and uses only commodity switching hardware with a minimal amount of data path memory
4. without compromising latency or throughput.

CRISS-CROSS — A hierarchical, reconfigurable topology for large-scale data centers: The locality in rack-level flows allows CRISS-CROSS to adjusts its topology to the traffic patterns. We show that CRISS-CROSS preserves many of the advantages of Clos topologies:

1. it maintains their hierarchy,
2. the simple routing algorithms,
3. their regular layout of connections for simple physical deployability,
4. and the compatibility to existing management approaches.

Nevertheless, we demonstrate that for group-based communication patterns, CRISS-CROSS improves the average flow completion time by 5.5× and the 99th percentile by 6.3×. For purely random point-to-point traffic pattern, it improves the flow completion time by 2.2× on average and 3× at the 99th percentile.

1.2.1 VNTOR: Rack-Based Network Virtualization

VNTOR is an off-the-shelve switch on which we implement security groups, a specific type of virtual networking abstraction. Security groups are the main mechanism offered to cloud tenants for controlling their communications, enabling them to enforce policies equivalent to those provided by a traditional stateful firewall in a physical network. We picked this particular abstraction for our proof-of-concept, because we consider it the most challenging one to implement at the ToR due to the large number of flows it utilizes. Furthermore, we designed and implemented the abstraction of a *virtual flow table*. A virtual flow table is a construct that fits orders of magnitude more access rules than the ToR's data path memory (the physical flow table). We provide this abstraction through a simple, two-level memory hierarchy, where the

top-of-rack's (fast but small) data path memory acts as a cache for a much larger and slower backing store accessible from the ToR's supervisor engine. To support this caching hierarchy, we designed an algorithm for typical switch ASIC data path memory. This algorithm takes the reduced access rates for the data path memory into account, as both reading the utilization information as well as updating and changing rules is several orders of magnitude slower than reading cache entries. As a consequence, this algorithm fundamentally relies on the temporal locality of the cached rules, which in our case means the locality of the traffic matched by each rule.

Combining all of the above, we are able to show that by using the locality of network flows, it is feasible to centralize the implementation of virtual networking at the top-of-rack switch. We demonstrate that this increases performance for flows, especially for those flows that reside in the data-path memory of the switch. We also show that resource demands on end-hosts are significantly lower. By removing all network virtualization functionality from the end-host, we also reduce the susceptibility of the network to hypervisor breakouts. Because VNToR has very limited hardware requirements, cloud providers can implement this solution today with a firmware update to their top-of-rack switches. This allows them to offer bare-metal support and faster communication with SR-IOV without compromising security or other specialized hardware.

1.2.2 CRISS-CROSS: A Hierarchical, Reconfigurable Data Center Topology

CRISS-CROSS extends Clos topologies with a layer of optical circuit switches to make them reconfigurable. It uses the locality of rack-level flows to adjust the topology to the traffic patterns. At the same time, CRISS-CROSS preserves the advantages of a hierarchical topology, namely the simple routing algorithms, the operator experience with deployment, as well as compatibility with existing management techniques. In CRISS-CROSS we make the primary assumption that the pod — the unit of higher available bandwidth in a Clos topology — is not a natural or effective abstraction to capture data center network patterns. In combination with oversubscription, this causes bottlenecks at the spine links. CRISS-CROSS uses optical circuit switches to rearrange the links between ToRs and fabric switches to break up the strict pod assignment of racks. This allows it to create temporary groups of active racks whose communication can stay at the aggregation level and does not need to use the spines.

To evaluate CRISS-CROSS, we implemented an event-based flow-level simulator designed for flexible topologies. Using this simulator, we evaluate CRISS-CROSS for shortest path routing and a *hierarchical* routing. For both routing algorithms, we evaluate CRISS-CROSS for two

traffic patterns: one with randomly distributed source and destinations, and another with a group-based communication pattern.

1.3 Thesis Roadmap

This thesis is organized as follows:

- *Chapter 2* explains the background knowledge for this thesis. It starts out with an overview over cloud computing and its requirements. This is followed by an introduction into data-center networking, including a brief overview over classical network topologies as used in data centers, and an abstract model of a switch. It also covers the concept of centralized control and software-defined networking in the data-center context and the resulting simplification in management.
  The second part of the chapter is about locality. We first provide the definition in its historic form for memory, and then extend it for networking via the concept of flows.

- *Chapter 3* discusses the work done for VNTOR. It first introduces the reader to different abstractions in cloud data centers with many tenants, focusing on the security group abstraction. Then, it compares the different existing systems for implementing these abstractions, which we use to motivate the design of VNTOR.
  It continues with a description of VNTOR and its underlying system design. Afterwards, it goes into more detail on the implementation of our prototype. The chapter closes with the evaluation of VNTOR for various types of traffic and metrics, before placing it in context with related work.

- *Chapter 4* presents CRISS-CROSS. It provides the background necessary to understand the design of CRISS-CROSS. Afterwards, it describes the CRISS-CROSS topology, developing the topology from non-reconfigurable Clos topologies. Once the topology has been defined, the chapter continues with a description of candidate selection algorithms, the link change procedure, and two different routing algorithms. Then, it presents the evaluation of CRISS-CROSS using our flow-level simulator. Finally, CRISS-CROSS is placed in context with the existing work in the field.

- *Chapter 5* concludes the thesis, in which we discuss some future directions in networking as well as potential research ideas for the future of the presented projects.

1.3.1 Bibliographic Notes

Chapter 3 is based on the paper VNTOR– *Network Virtualization at the Top- of-Rack* [52]. This was joint work published with my colleague Sam Whitlock and George Ioannidis at the ACM Symposium on Cloud Computing (2016). Not included in this thesis is my wo rk on distributed building services [45], my contributions to the security of netw ork functions [117], and my work on alternative RPC protocols for the data center a nd their P4 implementations [85].

2 Background

In this chapter, we provide some of the background for this thesis. We give an overview of data center designs. First, we present the field of cloud computing and its underlying technologies. Then, we extend this with a history of the most common topologies in data centers, starting at tree-based structures all the way to current-day setups, and their management with software-defined networking. Lastly, we close the section with a description of an abstract switch model as main component of data-center networks.

The second part of the background chapter deals with the definition of locality. We introduce the concept of temporal and spatial locality for memory, and then use flows to extend the idea of a working set to networking traffic. We close this chapter with an overview of the existing body of research into traffic patterns in data centers and its findings regarding locality.

2.1 Data Center Designs

This section describes the typical setups of data centers. It begins with an introduction into cloud computing and its influence on the network. It continues with the common topologies currently deployed and some background on their evolution, and explains the operation of using software defined networking. The section closes with an overview over newer topologies suggested by the networking research community in more recent years.

2.1.1 Cloud Computing

Cloud Computing is the concept of rapidly providing resources such as compute storage to a customer on demand without requiring direct management by the tenants themselves. These services are typically run by a third party, the cloud provider. Generally, the term is used to

describe any deployments of data center for a large number of users over the internet, with the data centers themselves distributed across the globe.

Cloud Computing, as it is understood nowadays, started around 2006 with the advent of Amazon Web Services' Elastic Compute Cloud [5]. Since then, most large internet companies have created their own variants, *e.g.*, Microsoft Azure [102], Google Compute Engine [60], IBM Cloud [72] and many others. Fundamentally, all of these clouds provide the same kind of services: they offer customers the ability to utilize virtual machines by the hour. These machines can have varying capabilities depending on the distinct customer needs. As an alternative, and to reduce the reliance on other companies, NASA and Rackspace Hosting released an open-source cloud software called OpenStack [114] for the management of private clouds. OpenStack enables at least larger company's IT departments to provide a similar experience in-house, creating private clouds, but retaining full control over all their aspects.

The business of cloud computing is large, with 2018 revenues projected near $190 billion, and two digit growth rates [55]. Cloud providers compete primarily on price, aiming to provide the highest performance for a given price point. Therefore, cost reduction is a primary driver of innovation in this space. Automation, efficient utilization of resources, flexible allocations, and many other factors are differentiators.

For the consumer, cloud computing provides many advantages, with the main point being the elasticity for demand. There is no capital expenditure beyond the initial system design, and the infrastructure can be scaled up and out nearly arbitrarily depending on business growth and customer demands. It can even be used temporarily to provide additional resources; the advent of machine learning means that nowadays, users can rent specialized hardware temporarily for both training models and even inference in products. In general, all businesses with large, but infrequent computing demands can benefit from cloud deployments.

Virtualization is the core enabling technology behind Cloud Computing [5, 59, 143]. Virtualization in virtual machines (VMs) means exposing abstractions of the hardware interfaces of the underlying hardware via a software layer. This approach has many advantages, of which we describe the most commonly cited ones here [74]. Virtualization allows renting out VMs to sized to the customers' needs, and colocating many of these VMs on the same hardware, *i.e.*, *sharing* of resources that would otherwise be too big. Colocating multiple VMs on the same machine provides efficiency gains. At the same time, virtualization provides *isolation* between the different tenants, *i.e.*, they can not interfere or monitor other tenants' actions. On the opposite side of the spectrum is *aggregation* in case the demands of a customer exceed the size of a resource. A typical example here is storage. Sharing resources in a large data center also allows customers to quickly *scale* their resources if demand arises. Lastly, virtual

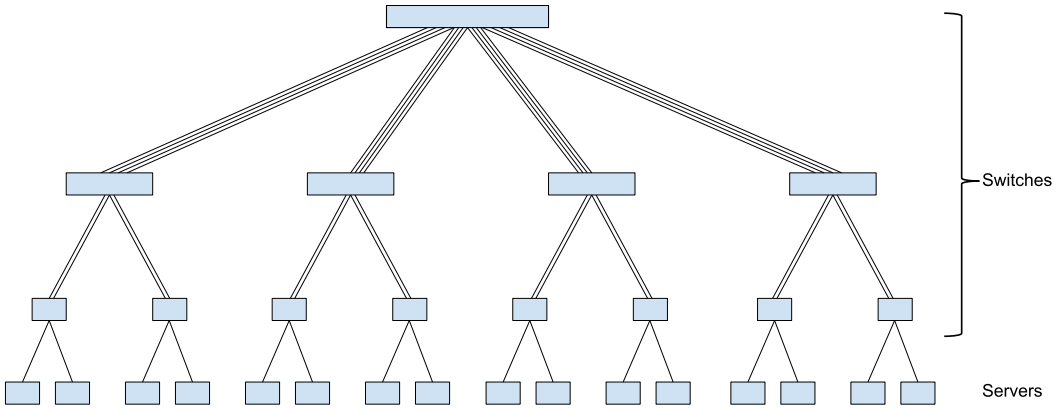


Figure 2.1 – Fat Tree - Doubling link capacity at every level

machines are *easily managed* by automatic processes that can deploy, move, and destroy VMs as they are completely software-based. Thanks to virtualization support in modern hardware, the performance differences of virtual hardware over bare metal are very low [21].

More recent developments go towards different levels of deployment, from software-as-a-service, over platform-as-a-service to the more classical infrastructure-as-a-service, or even serverless computing and its cousin function-as-a-service. These allow for an even better fitting of resources to client demands, as they provide different abstractions beyond the physical hardware abstraction of VMs.

All these solutions however have in common that the tenant has no control over the location of the application in the data center. However, the same is true for the data center operator. Scheduling software knows little about the demands of the applications themselves to optimize their placement in terms of network connectivity. In fact, even if such information was available, other considerations such as an avoidance of shared-fate might make network connectivity secondary.

Most commonly, companies work with public clouds for their applications. However, it is also possible to use private or even hybrid clouds. In this context, public cloud means that the cloud is shared by many diverse tenants and is run by (one of) the above-mentioned companies. On the other hand, private clouds are run by the IT departments of the companies themselves, most commonly by large companies because of the required economies of scale. The advantage of private clouds is that data is retained completely under the owner's control, which can be necessary either for judicial or for competitive reasons. The last alternative are hybrid clouds, where the IT departments utilize both private and public clouds and split their needs between them to provide the best from both worlds, *i.e.*, the near-infinite scaling of public clouds as well as the guarantees for privacy in the private ones.

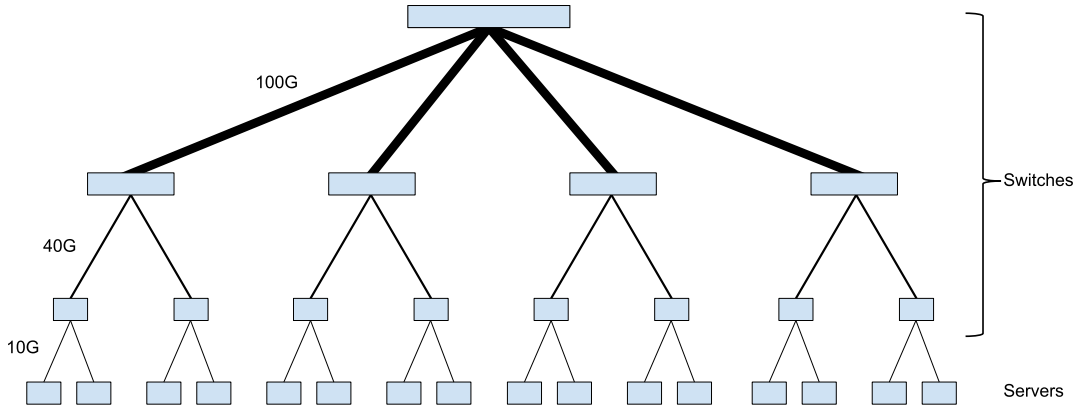


Figure 2.2 – Fat Tree

2.1.2 Common Topologies

Modern data centers employ a small set of common topologies in production. The research community has proposed many alternatives (see § 2.1.3), however, none have made it into real-world data centers to the best of the author's knowledge. Historically, fat trees[1] [91] were the dominating topology. In fact, even today they are one of the most commonly used topologies in the HPC community [40]. This topology is organized as a tree of switches (see Figure 2.1). In Leiserson's original definition, the fat tree was defined as a complete binary tree where the number of links at a level was twice the number of links at the level below (*i.e.*, further from the root). With this definition, there is guaranteed full bisection bandwidth between all nodes. The shown tree is a generalization of this concept beyond binary trees. Given that in the real world there are radix limitations for switches as well as cabling costs, variants of fat trees exist in which multiple links are replaced with a single link of higher bandwidth, without necessarily guaranteeing the same amount of bandwidth. An example for this is shown in Figure 2.2. In this example, the root connects to the second layer with 100Gbit/s, the second to the third layer with 40Gbit/s, and the third to the fourth layer with 10Gbit/s and so on.

The disadvantage of fat trees is their limited scalability. As every additional layer needs an associated increase in link speed or radix increase, there is both a technological limit to the maximum speed and a disproportionate growth in cost as faster link speeds and higher radix switches are substantially more expensive.

A first answer to these challenges was the introduction of multi-rooted (fat) trees (see Figure 2.3). In these, the single core router was replicated, *i.e.*, there were multiple core routers. This setup permits the upward bandwidth of the second layer switches to be split between multiple links. The second layer routers are commonly called *aggregation* switches, as they

[1] As is explained further on in this section, we utilize the original definition of a fat tree by Leiserson [91]

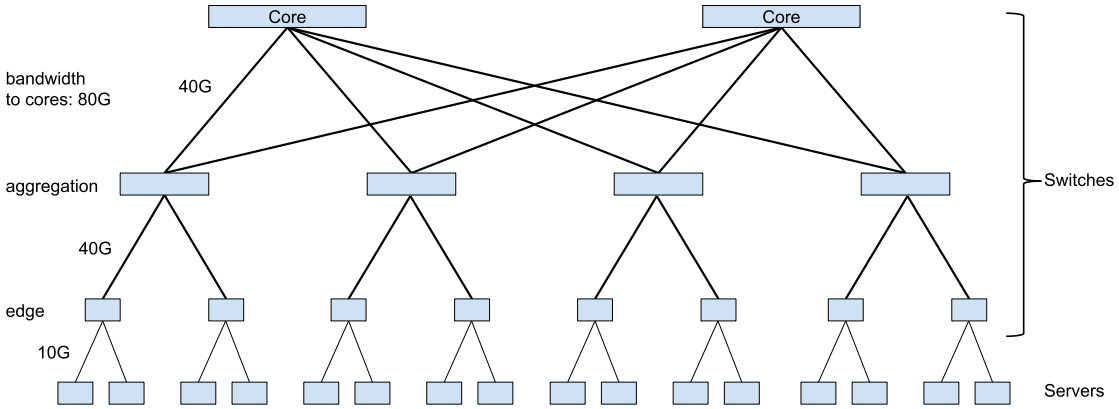


Figure 2.3 – Multi-Rooted Fat Tree

aggregate multiple third layer switches, from now on called *edge* layer . However, these topologies still require the use of high-bandwidth links with their associated high costs for the non-commodity switching technology.

Additionally, although multi-rooted fat trees solve the issue of high numbers of links or high link speeds at the higher levels, they still suffer from a lack of resilience. If an aggregation switch fails in this topology, all edge switches connected to this aggregation switch would lose connectivity to the network as well. Similarly, the failure of an uplink of an edge switch would mean a total loss of connectivity.

The next step in network design was therefore to introduce a certain degree of redundancy for the edge to aggregation links. Examples of this are the classical Cisco cluster architecture [8], Google's traditional network [131] or Facebook's old four-post network [7, 49]. Figure 2.4 shows an example of such a topology for a "four-post" design used by both Google and Facebook. In this setup, each edge switch is connected to four different cluster ("aggregation") switches — the four posts. Furthermore, the four cluster switches are connected in a protection ring to increase resiliency in the face of link failures. Together, this set of cluster switches and edge switches make up one cluster. The cluster switches in one such cluster are in turn connected to four core switches each (again, the four posts). These core switches also have a protection ring connection to preserve connectivity during failures.

The main disadvantage with this setup is that it requires switches with very high radix to connect all racks, with the cluster size defined by the radix of the cluster switches. Additionally, these large radix switches were also very specialized and therefore expensive, ran proprietary software making them difficult to debug. Lastly, even single switch failures would reduce the capacity of the network by 75% either within the cluster for the cluster switches, or between clusters for the core switches.

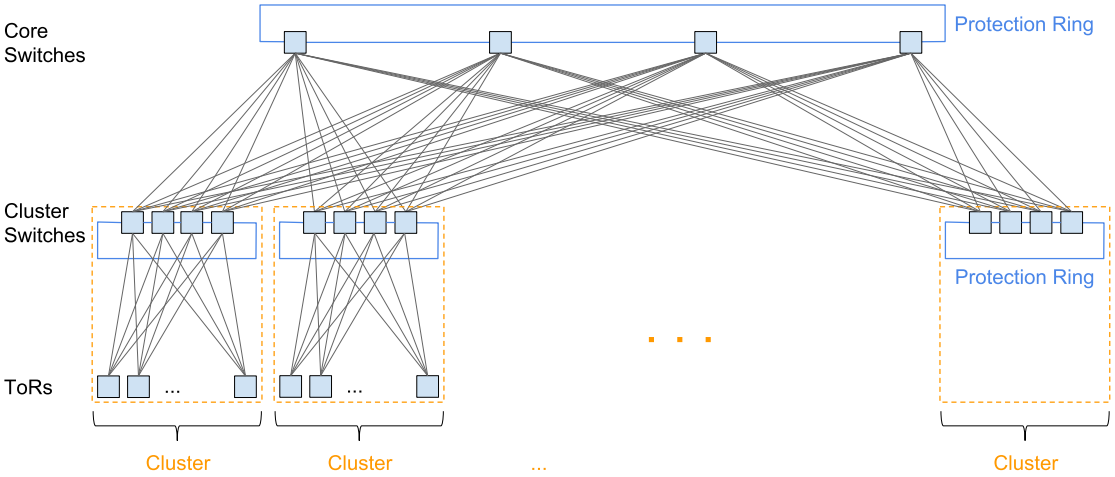


Figure 2.4 – The cluster / 4-post design

In 2008, Al-Fares et.al. [1] published an alternative topology that used only commodity switches. An example of its most general form is shown in Figure 2.5. Incidentally, the authors also named their topology fat tree. However, it is a variant of a folded Clos topology, and we will therefore use this terminology to avoid confusion. Clos topologies are a relatively old concept first developed for telephone networks [29]. They were used to route calls from an ingress stage through a middle stage to an output stage. The term folded refers to the fact that in the data center, the ingress and egress stage are the same switches. Since then, variations of folded Clos topologies have become the de-facto standard in large data centers by Google [131], Facebook [125], Microsoft [132], or Amazon [122].

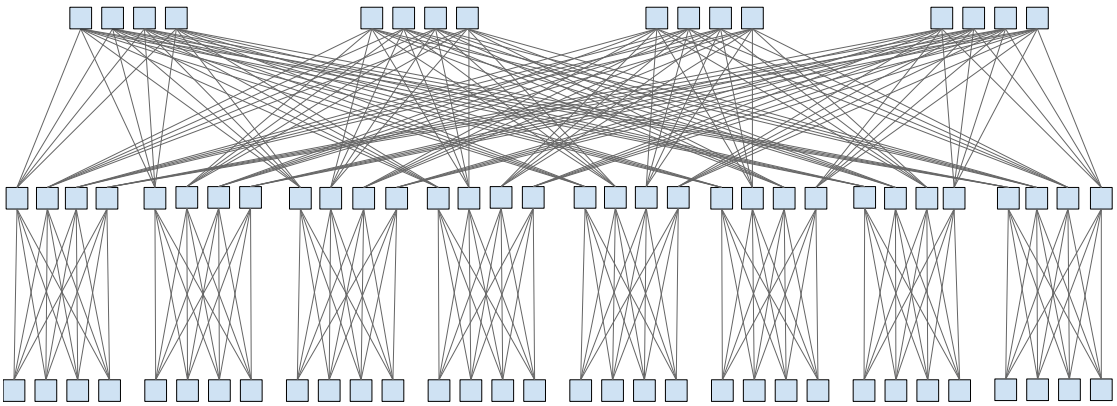


Figure 2.5 – Standard Folded-Clos Topology

Clos topologies tackle the main issues that stand in the way of lowering costs. The use of commodity switching equipment means low capital expenditures. The regular structure of the topology permits simple, standardized cabling, easy understandability, and easy routing.

The high number of links also provides a large degree of redundancy, where the loss of a single link has very limited impact on the bisection bandwidth. The multi-layered structure also allows for oversubscription to control cost (we discuss the matter of oversubscription and its trade-offs further in Chapter 4). Lastly, Clos topologies can also be built with lower-radix switches, removing the need for highly specialized hardware.

One potential downside of Clos topologies is the increased number of networking elements leading to more routing complexity. This disadvantage is mitigated through a centralized approach for network management. The regular structure produces simple rules to create routing entries. The nomenclature for the different layers stays consistent with the one for fat trees, from spines at the cores that connect the *fabric*[2] switches, which in turn connect the edge switches. Again, further intricacies of the topology are discussed in Chapter 4.

2.1.3 Alternative Topologies

Many other alternative topologies have been developed; we describe them in relationship to CRISS-CROSS in §4.7. Some projects utilize partial reconfiguration by adding additional technology to the network in the form of mirrors [9, 57, 145] or wireless links [65]. Others add additional optical circuit switches either for parallel optical networks [51, 121] or for (partial) reconfiguration [24, 100, 141]. A last category of alternative topologies aims to completely change to topology of the network, *e.g.*, by including the servers in routing [63, 64] or using networks inspired by random graphs [133, 137]. All these topologies have various optimization criteria, with a general aim to increase performance and decrease costs. Nonetheless, data center operators tend to be conservative in the deployment of new designs. As an example of this conservatism, Singh et.al. [131] even describe that for the test of a new network topology, they deployed the old one in parallel to prevent any issues. The fail-safe network could then easily take back control with the press of a "big red button". As a consequence of this conservatism, these new topologies have had a hard time of getting adopted so far.

2.1.4 Centralized Control

As described in §2.1.1, modern data centers are run in a largely automated fashion. For cloud data centers, this is synonymous with a centralized management approach. Opposite to early data centers, cloud data centers are run by a single entity, and the networks are very homogeneous. This individual entity therefore has universal control over all components.

[2] In the literature, these are also often called aggregation switches. In this thesis, we follow the naming convention by Facebook and refer to them as fabric switches.

Single ownership and homogeneity mean that a centralized network control plane becomes feasible.

The concept to control the network centrally is called software defined networking (SDN) [98]. It positions itself as an alternative to the classical distributed algorithms for routing and path discovery. SDN enables the operator to get complete visibility of the network. This visibility allows easier reasoning over the state of the network as well as its configuration as compared to the distributed algorithms.

The network is separated into a control and a data plane. The control plane is the intelligence of the network that decides on the routing process. The data plane is only responsible for forwarding of traffic, following the configuration from the control plane. The novelty of SDN is that the control plane is separated from the data plane and centralized in a controller.

This clear separation of concerns allows switch vendors to sell white-box switches providing the data plane component. These switches consist of a generic switch ASIC and a control CPU that can then be custom-programmed by the network operators. As these white-box switches replace specialized hardware, they can be priced very competitively.

Initially, the control plane was completely centralized, with a single controller having full visibility over the network. Later designs introduced several new ideas. Hierarchical controllers [71, 75] make state more manageable as it gets aggregated into larger units, *e.g.*, by combining all links between data centers into one virtual link. Distributed implementations [88] make the network more resilient in the face of failures and improve scalability if the work can be partitioned between multiple controllers.

The first implementation of SDNs used OpenFlow [98], a protocol to program the forwarding tables in switches with match-action rules with a vendor-agnostic protocol. This allows the centralized controller to program the flow tables of all switches according to its policy. The protocol itself specifies a match data structure that gets applied to each incoming packet, and associated actions that the switches should execute if the packet matches the specification. Since this first proposal, SDN has truly penetrated the market. Google was the first major company to publish their work with SDNs in B4 [75], an SDN based solution that uses the global knowledge of network state for better network control. Since then, many more have followed (*e.g.*, Facebook's routing platform Open/R [69], data center fabric [7]).

Controllers have three ways of installing routes in the network, either proactive, reactive or a combination thereof. Proactive means that the rules for a packet are installed before an associated packet ever arrives at the switch. This way, there is no delay, and the packet can be delivered directly to its destination. In reactive routing, unknown flows are redirected to the

controller. The controller then decides what should happen to the packet. Possible outcomes include the installation of rule entries in arbitrary sets of switches, and the forwarding or dropping of the packet.

One particular implementation that has gained popularity is OpenStack [114]. While OpenStack is a whole suite of software to manage (private) clouds, their networking component Neutron [112] in particular uses SDN for network setups and control. Tenants can control all aspects of their (virtual) networks, from IP addresses to logical topology, network functions such as intrusion detection systems, firewalls and many more.

2.2 Switch Architecture

This section will give a brief overview over the components of a normal data center switch. Figure 2.6 shows an abstract model of such a switch[3]. This is not meant as a full introduction into the architecture of switches, but more to give the unfamiliar reader an idea of their structure. In this simplified model, switches consist of three main components: the switching ASIC[4], a CPU for the supervisor, and the actual ports. We follow in this description the one by Seifert & Edwards [127].

The ports are the connectors of the switch to the rest of the network. They are composed of the *physical layer receiver* (PHY) and the *Media Access Control* (MAC) [127]. The PHY converts the electrical or optical signal from the transport medium (*i.e.*, copper or fiber, respectively) to a bit stream on the receiving side, and vice-versa on the transmit. On the receiving side, the MAC does the necessary checks for validity and errors. It also keeps most of the port-specific statistics for the switch. On the transmit side, the MAC generates the needed checksums. Network operators often choose to deploy a mixture of copper and fiber based networks. The reason for this is that the actual PHY for fiber costs significantly more than for copper, especially at higher speeds, but the range of copper is very limited at high link speeds (depending on the technology used, between 3m and 30m). As a result, the operators use copper wherever lengths permit its use, and fiber for the rest.

The ASIC takes care of all the packet forwarding in the switch and makes up the data plane described in §2.1.4. In our model, the switch is made up of a set of ingress pipeline steps that define where a packet should go, the fabric that does the actual connection of input to output ports, an egress pipeline for post-processing the packets, and an output buffer which stores packets until there is a free sending slot on the output port.

[3] We purposefully skip the details of VLANs, priorities, flow control, auto-negotiation etc. and instead refer the interested reader to [127]

[4] Application Specific Integrated Circuit

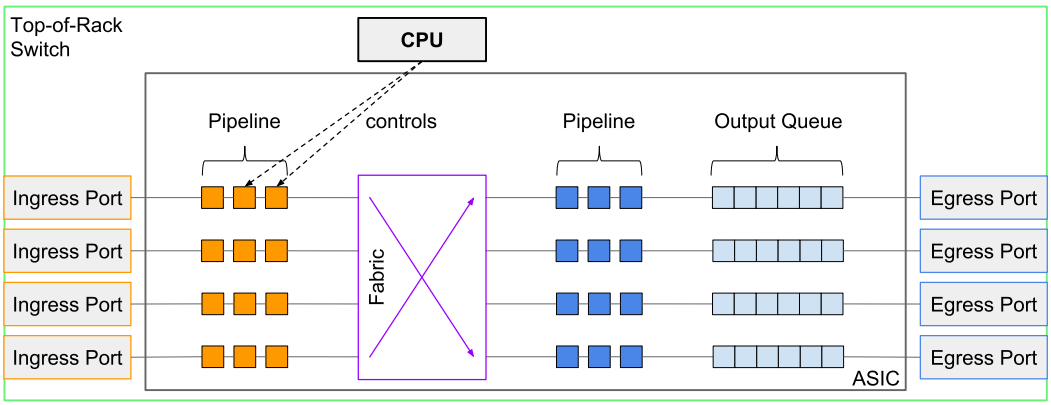


Figure 2.6 – An abstract example of a switch

The ingress pipeline contains the logic for flow control[5], several filters such as access control and VLANs, and the lookup engine. The lookup engine takes the incoming packets and parses out a set of classifiers. Then, the lookup engine uses a set of tables, *e.g.*, routing tables or MAC addresse in combination with the classifiers to determine the correct handling of each packet. It may forward the packet to a specific output port, to a set of output ports (multicast), an internal processor port, or it may simply drop the packet. The lookup engine implementation itself is highly dependent on the specific switch model, but most use a combination of SRAM, content addressable memory (CAM) and ternary content addressable memory (TCAM). After the handling of a packet has been determined, it is passed to the switch fabric. The fabric uses the output port information for a packet to forward it to the specific egress pipeline for that port. The egress pipeline may apply different filters (*e.g.*, VLAN memberships, dropping packets for inactive ports) or transform packets (*e.g.*, time-to-live modifications or adding VLAN headers). After passing the egress pipelines, the packets are scheduled into the output queue (*e.g.*, with Deficit Weighted Round Robin [130]). This queue is responsible for many of the performance aspects of the network . Because multiple input ports can forward packets to the same output port, the output queue can grow in size causing queuing delays. The release of packets from the head of the queues to the wire again uses some form of flow control (*e.g.*, the response to PAUSE frames or an implementation of 802.1Qbb [73]).

The last component of a switch is the supervisor engine which runs on a separate CPU. Classically, these were running vendor specific OSs such as Cisco IOS [28] or JunOS [78]. With the advent of generic white box switches, Linux based operating systems such as Cumulus Linux [34], Microsoft Azure Cloud Switch [104], and Facebook Open Switching System [48] have become more common. This allows very flexible deployment of software on switches.

[5] *e.g.*, PAUSE frames

The supervisor engine has two main tasks. It monitors the switch state (and potentially communicates it to a central controller), and it configures the forwarding state of the switch, *i.e.*, the data plane by modifying the tables used by the lookup engine mentioned above. The CPU derives the forwarding state either via decentralized protocols such as IS-IS, GBP, or OSPF, or via programming from a centralized control plane (see §2.1.4).

2.3 The Locality Principle

The principle of locality, or *locality of reference* [41,42], describes the relationship of a sequence of memory access locations. Every program has a set of memory locations that it uses during its operation. As programs typically do one thing at a time, they often utilize the same parts of memory again and again within a certain interval, or utilize a lot of memory in the same region. These types of accesses are referred to as temporal or spatial locality, respectively.

The primary driver for the definition and discovery of locality was virtual memory and shared computing. To allow operating systems to run multiple programs efficiently but concurrently, they needed to share the available memory. As this was a critical resource, early operating system designers introduced the concept of virtual memory, essentially providing an abstraction of an unlimited memory available to each program, where each program also has its own view of memory. Memory not currently used was unloaded to disk, or paged out.

During an interval of time, only a subset of memory is used by any one program. The subset accessed by the program in the interval prior to a specific point t_{now} in time is defined as the *working set* of the program at point t_{now}. Once a page is accessed, it becomes part of the working set for the specific interval

Assuming the working set of a program is located in fast memory and not paged out, the performance of that program is very similar to the same program, but for which all pages reside in fast memory. When paged-out memory is accessed, it is loaded back into main memory, while another part is paged out.

Temporal and spatial locality are defined through the concept of a *reference string* [126], *i.e.*, the stream of virtual addresses generated by an application.

- *temporal locality*: the reference string contains several closely spaced references to the same address. An example of this might be a loop that sums up values and stores the result.

- *spatial locality*: the reference string contains several closely spaced references to adjacent addresses. A prime example of this is a loop that iterates over an array, *i.e.*, neighbouring memory.

In modern systems, locality is used at all layers. The prime example for this are cache hierarchies in all components of modern computers (and even warehouse computer systems with cache servers). If a program exhibits temporal locality, then caches often already contain the data that the system needs because it was accessed not too long prior to the current access. Spatial locality can be used to make these cache hierarchies even more efficient. As memory buses are often organized in fixed sizes, the memory subsystems often load full cache lines instead of single object sized parts of the memory. Programs that access lots of memory within a single location, *i.e.*, exhibiting spatial locality, profit from this behavior as even previously unaccessed parts of memory are already preloaded. This preloading can also be extended by using specific prefetchers that preload related memory.

In the context of networking, locality has to be defined slightly differently. Here, it is not data that is frequently read or written, but rather a pair of source and destination that is active over a certain time interval. We therefore use the concept of a *flow* [19, 47] to define locality. In this thesis, we define a flow as a set of packets that share a common source and destination.

The granularity of source and destination – or end points – for a flow can vary depending on the specific task. The most common definition of a flow is the 5-tuple of source and destination IP, source and destination port, and the transport protocol. The 5-tuple is a generalization of the definition of a TCP connection which is defined by those same 5 fields.

Beyond the 5-tuple, there exist less granular definitions of flows that aggregate end points [47]. The first aggregation is for end-hosts: removing the ports from the definition of a flow means grouping all packets with the same source and destination IP addresses into the same flow and treating all of them the same way. At the level of topologies, an even coarser definition can be used, where it is only the source and destination racks that define a flow.

After introducing the concept of a flow, we can now use it to define temporal locality in networks by redefining the working set as a subset of flows. Studies on flows have shown that the majority of traffic is carried by a relatively small subset of flows [47, 57, 89, 115, 125]. In many applications such as traffic engineering, it is sufficient to only manipulate the flows in this subset to steer the majority of traffic as wanted. This is similar to virtual memory where significant performance gains can be made by keeping the working set in fast memory. We therefore need to define certain criteria that discriminate between heavy and non-heavy flows, where the heavy flows then form the working set.

Given this intuition, we arrive at the following informal definition: *The working set for flows is the set of flows that need to be treated differently than the default behavior in order to achieve a certain performance target.* Most commonly, the subset of flows within the working set is the set that carries the majority of traffic while being relatively low in numbers. The actual quantitative definition of performance and the subset of flows depends on the application [115].

In the literature, heavy flows are typically called *elephant flows*, with the rest classified as *mice flows* [115] (or less commonly ants [47]). Elephant flows are characterized by a sufficient size as well as a sufficient duration. However, the concrete values for these two restrictions as well as their derivation depend heavily on the use case [107].

Significant work has been done on recognizing elephant flows. Ideally, an algorithm would report the set of flows exceeding a certain threshold and their respective size at any moment in time [47]. The proposed techniques range from simple sampling based strategies [107] such as Cisco NetFlow [27] or Juniper JFlow [77], sample and hold as well as multistage filters [47], end-host based methods [35] to programmable switch based approaches [120], even allowing network wide heavy-hitter detection [68].

Beyond the definition of temporal locality, Roy et.al. [125] define spatial locality as the amount of traffic that stays rack-local, cluster-local, or Facebook-internal.

There are various reasons for these skewed distributions. Examples include:

- Cloud computing tenants that primarily communicate within their own network. The virtual machines of tenants are typically only allowed to communicate with other machines of this tenant or with the outside. Hence, their communication is limited to a small number of end hosts.

- Large jobs for example in map-reduce calculations communicate only within their worker sets. Specifically for map-reduce, there are phases of very active communication in the shuffle phase, and periods of coordination traffic only in the map phase [39].

- Certain data centers are structured in a way that similar services are colocated in the same racks [125]. Therefore, communication is between subsets of racks, *e.g.*, the web server rack that communicates with the cache rack and the database racks.

We utilize the definition of heavy flows in VNTOR. By keeping a subset of particularly heavy flows in the hardware tables, most traffic is handled by the hardware data plane so that the software data plane only has to deal with a small fraction of total traffic. This means that

overall, the performance of this hybrid data plane is similar to that of a hardware switch with sufficient capacity to contain flow entries for all flows.

For CRISS-CROSS, we aggregate the flows at a rack level, as this is the unit of reconfiguration. The topology is then optimized so that the set of heavy flows exhibits minimal spine utilization, *i.e.*, it optimizes the topology so that the heavy flows can stay on the aggregation switches.

2.3.1 Traffic Patterns

In this section, we will give an overview over the literature characterizing locality and network traffic. The interested reader is invited to look at the source material for more details beyond the short characterization here.

Traditionally, much of the traffic in a data center has been north-south – the server talks to a client somewhere on the internet – rather than east-west – the servers talk to each other within the data center. This change is due to a change in workloads: pre-cloud, most clients were located on the internet and communicated with a small number of servers per request. Post-cloud, many functions are implemented within large clusters of computers that mostly talk to each other, with 100s or even 1000s of servers involved per request [11]. This change in application design lead to a shift in the observed flows in data centers.

Benson et.al. [12] studied three different types of data centers, a university, enterprise and cloud data center (~2009/2010). In all types of data centers, they observed less than 10000 active 5-tuple flows during all 1 second intervals. They also observe that 80% of flows are short in all types of data centers, carrying less than 10KB, with the rest making up the bulk of traffic. In a second paper, Benson et.al. [14] observed 19 different data centers and investigated the loss rates across different link types and noticed that nearly 4% of core links were suffering from packet loss.

At the university of Calgary [44], the top 1% of flows were responsible for 73% of all traffic, with the top 0.1% responsible for 46%. Other sources report even more skewed traffic with 0.02% of flows responsible for 60% of all traffic [107].

Early work in the field of cloud data centers mostly looked at more specialized variants. Kandula et.al. [80] examined data center traffic with a focus on a data mining data center. This data center exhibited strongly grouped traffic. The probability of exchanging no traffic between two servers is 89% for all pairs, and 99.5% for pairs in different racks. Interestingly, the arrival rate for rack flows in this research was a relatively modest 2000 flows per second at a rack aggregation level. 90% of traffic was carried by flows that lasted at least 10s.

Ghobadi et.al. [57] looked at another set of more modern Microsoft data centers running a mix of workloads, including index builders, map reduce workloads, database as well as storage systems. In total, four different data centers were investigated, with 100-2500 racks. In these, 46-99% of racks exchanged no traffic, while less than 0.03% of rack pairs accounted for 80% of all traffic. A second aspect that the authors evaluated was the elephant outdegree. The elephant outdegree is defined here as the number of destinations with which the origin switch communicates with elephant flows. During a single day, 20% of racks have an outdegree of over 190, 10% of 70, and the rest less than 10. On a 5-minute basis, about 10% of racks have more than 50 elephant flow destinations, with a maximum of 200.

Roy et.al. [125] report similar numbers as the above papers but for various Facebook data centers (or rather for clusters). They describe traffic as being either rack-local or destined to 1-10% of hosts in a cluster, depending on the clusters main function. Using a definition of heavy-hitter as being the heaviest flows that cumulatively are responsible for 50% of all traffic, they note the following group communication pattern (within a 5ms interval): for webservers, each host communicates with 5-8 other racks. For cache followers, this number goes up to 30 (with extremes of 50 at the tail).

For non-data center workloads, Varghese et.al. [47] showed that about 2% of 5-tuple flows are responsible for 50% of traffic both for a WAN access point and between two ISPs. On the internet, 90% of traffic is caused by a small subset of elephant flows [89].

3 VNToR: Network Virtualization at the Top-of-Rack Switch

This chapter presents our work on VNTOR. This work was largely presented at the ACM Symposium on Cloud Computing 2016 (SoCC'16) in our paper "VNToR: Network Virtualization at the Top-of-Rack Switch" [52], but has been augmented with additional information and details.

3.1 Introduction

Cloud providers are starting to virtualize their networks and expose to their tenants familiar network abstractions like a layer-2 broadcast domain, an IP subnet, or a security group. These abstractions make it easy for tenants to replicate their physical network organization in the cloud and manage it with the same familiar processes.

Cloud providers typically implement network abstractions on the server, within the operating system (OS) that hosts the tenant virtual machines (VMs) or containers. The host OS implements virtual switching, for example Open vSwitch [118], to provide connectivity to the local VMs and also opportunistically implements cloud network abstractions: it performs the encapsulation or translation needed to provide the abstractions of a layer-2 network and IP subnet, and it enforces the access rules dictated by the security groups. This approach is flexible and convenient: first, it does not require any changes to network devices, which are typically hard to reprogram; second, it allows easy coordination between the deployment of virtual-compute and virtual-network resources, because both are deployed through agents running in the same place—the host OS. An example of such a deployment is shown in Figure 3.1a.

Despite its benefits, this approach has significant performance and security problems:

(a) Incompatibility with bare-metal support[1]: For security reasons, network abstractions must be implemented at an entity outside the one being governed. For instance, if a security group specifies that tenant X must not send any traffic to tenant Y, it does not make sense to let tenant X itself enforce this rule. Hence, when tenants are given access to servers, network abstractions must be implemented somewhere *outside* the server.

(b) Unnecessary performance overhead: Compute virtualization platforms are designed to avoid host-OS involvement as much as possible, and implementing network abstractions within the host OS violates this mentality. Bypassing the host OS with single-root I/O virtualization (SR-IOV) [116] and offloading the implementation of network abstractions elsewhere has tremendous potential for performance improvements [43, 109].

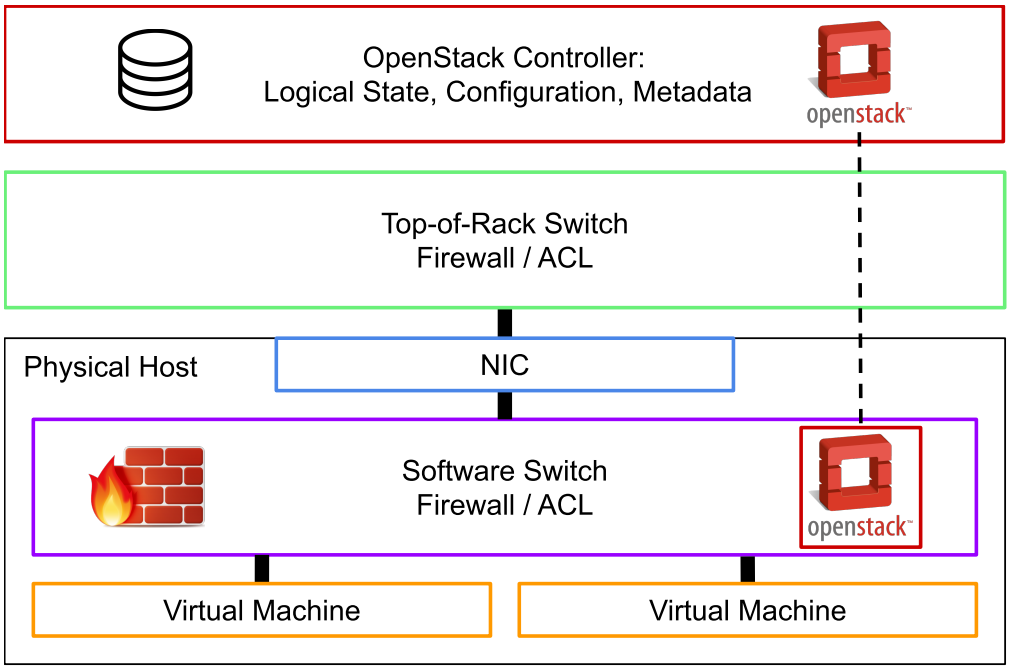
(c) Susceptibility to hypervisor breakouts: Bugs in hypervisor implementations allow malicious/compromised tenants to escape the confines of their VMs and run commands at the hypervisor level. Multiple vulnerabilities have been discovered in recent years (including vmftp [30], CloudBurst [31], KVM Virtunoid [32], and XEN Sysret [33]) that would allow such a tenant full access to the provider network.

To solve these problems, we propose to offload the implementation of network abstractions to the top-of-rack switch (ToR); to show that this is feasible and beneficial, we present VNToR, a ToR that takes over the implementation of the security-group abstraction. Figure 3.1b shows a subset of the possible configurations with VNToR.

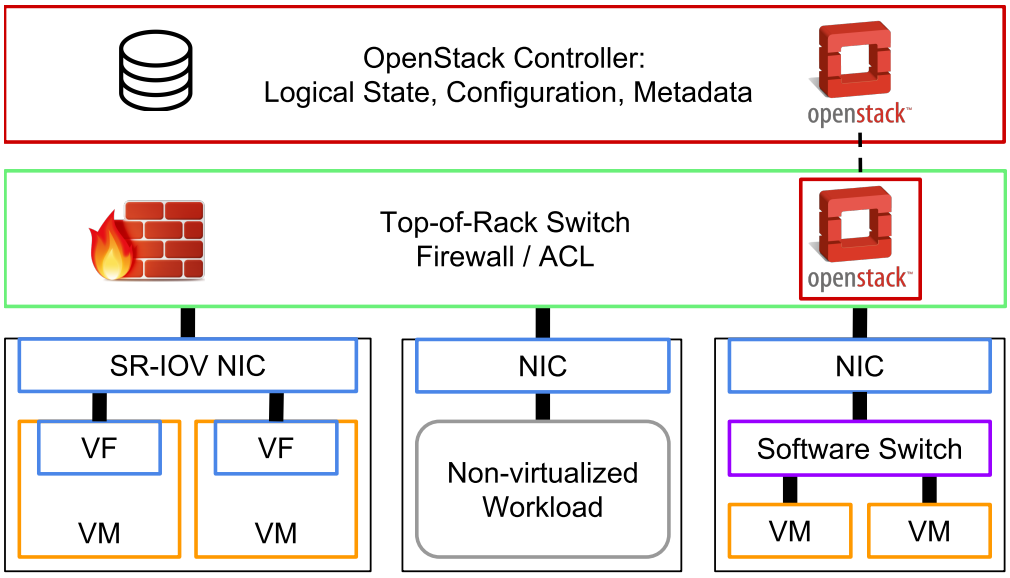
Security groups are the main mechanism offered to cloud tenants for controlling their communications, and they enable tenants to enforce policies that in a physical network would be enforced by traditional stateful firewalls. We picked this particular abstraction for our proof-of-concept, because we consider it the most challenging one to implement at the ToR (we comment on this in § 3.3).

Others have taken similar approaches: Fastrak implements certain network abstractions at the ToR, but only for a few flows selected by the guest OS [109]. In contrast, VNToR does it for all traffic and without any changes to the guest OS, which leads to very different technical challenges. As we were completing our work, Amazon announced they are also offloading the implementation of security groups, but to the network interface card (NIC) as opposed to the ToR [138]; their implementation involves a proprietary NIC without a publicly available technical specification. Microsoft has gone a similar route with their FPGA-based Azure SmartNIC [3]. In contrast, our approach does not require proprietary hardware, but only software changes at the ToR.

[1] Where the cloud provider offers tenants access to physical machines.



(a) Hypervisor-centric (baseline)



(b) VNToR

Figure 3.1 – Two approaches to cloud virtual networking using OpenStack

The main technical challenge we face is that ToRs are often low-end switches without the resources required for a straightforward implementation of network abstractions. More specifically, a typical ToR lacks the amount of data-path memory needed to store all the access

rules needed to implement security groups (*i.e.*, the set of access rules dictated by security groups that contain VMs or containers running in servers connected to the ToR). This is a fundamental limitation related to the ASIC manufacturing process, which is unlikely to disappear in the near future [105].

To address this challenge, our ToR exports the abstraction of a *virtual flow table*, which fits orders of magnitude more access rules than the ToR's data-path memory (the physical flow table). We provide this abstraction through a simple, two-level memory hierarchy, where the ToR's (fast but small) data-path memory acts as a cache for a much larger and slower backing store accessible from the ToR's supervisor engine.

At a high level, the idea of data-path memory as cache has been explored before, in the context of Software Defined Networking (SDN) [83, 142, 144], but the substance of our work is different. In SDN, what makes caching challenging is the presence of overlapping rules, because naïve caching of such rules interferes with forwarding semantics. Hence, related SDN work focuses on the correctness of the caching algorithm in the presence of overlapping rules—an interesting algorithmic problem, which is independent of underlying-hardware details. In contrast, we set out to eliminate the communication overhead due to host-OS involvement, and we must be careful not to move this overhead to the ToR. Unlike SDN work, our ToR has a concrete, hard baseline to match: that of a ToR that does not implement any network abstractions. Hence, we focus on the performance of the caching system, which is very much dependent on hardware details like the polling and update rate of the switch's data-path memory. We discuss this difference in § 3.3.

In summary, our contribution is the design (§3.4), implementation (§3.5), and evaluation (§3.6) of VNTOR, a system for state-of-the-art clouds that implements security groups at the ToR. This requires a ToR that:

- can store tens of thousands of access rules;
- adapt to traffic-pattern changes, typically in less than one ms;
- while using commodity switching hardware with a minimal amount of data-path memory;
- and without compromising latency or throughput.

Cloud providers can implement this solution today with a firmware update to their ToRs, which would enable them to offer bare-metal support and faster communication with SR-IOV without compromising security. We have implemented a prototype on top of a 64 × 10Gbps

Broadcom Trident+ switching ASIC [16] and a low-power XLP supervisor engine [17], and integrated our prototype in OpenStack Neutron [112].

3.2 Background

In this section, we describe the security-group abstraction as exposed by current clouds (§3.2.1); OpenStack and how it implements security groups (§3.2.2); and the SR-IOV hypervisor bypass technology (§3.2.3); we then compare existing security-group implementations (§3.2.4).

3.2.1 Security-Group Abstraction

A security group consists of tenant entities and ingress/egress rules. The entities can be VMs, containers, or physical machines in the case of bare-metal support, and the rules specify what traffic the group members can send or receive. A rule consists of a protocol (typically TCP, UDP, or ICMP), a destination port or port range, and a remote location that is either an IP subnet or another security group. Any traffic between security groups not explicitly allowed by an egress *and* ingress rule is denied.

For example, the following rules allow TCP connections from group A to group B at port 8080:

Group	**Type**	**Protocol**	**Port**	**Remote loc.**
A	egress	TCP	8080	B
B	ingress	TCP	8080	A

The first rule is an egress rule associated with group A, which allows all entities (VMs etc.) from A to initiate TCP connections to any entity from B at port 8080. The second rule is an ingress rule associated with group B, which allows all entities from B to accept TCP connections at port 8080 from any entity from A. If these are the only rules associated with groups A and B, the entities of these groups cannot send or receive any other traffic.

3.2.2 OpenStack and OVS Enforcement

OpenStack is the standard open-source cloud-management system, and Neutron is its network-management module. It consists of one centralized controller and multiple agents, the latter running on all the devices implementing network abstractions (in current clouds, these are only the servers). The controller maintains all state related to network abstractions and distributes it to the relevant agents, which translate it into configuration and install it on the local device.

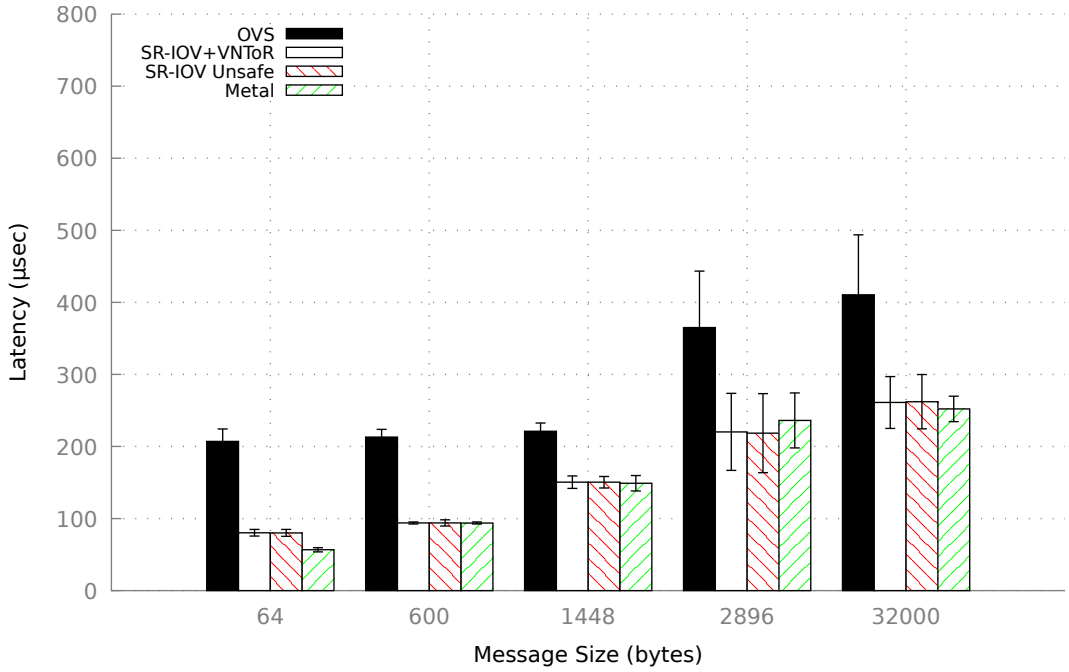


Figure 3.2 – netperf request/response benchmark

Neutron currently enforces the security-group rules at an Open vSwitch (OVS) running as part of the hypervisor, and commercial solutions such as VMware NSX [87] use a similar approach. When a tenant adds a VM to a security group, the Neutron controller sends the group's rules to the Neutron agent running on the VM's hosting server. In response, the agent converts the group rules into standard firewall rules and installs them in the local OVS switch. Firewall rules are installed in an iptables rule chain, with connection tracking enabled, associated with the OVS switch.

3.2.3 SR-IOV and Security Groups

In traditional compute-virtualization platforms, network I/O (and I/O in general) always involves the hypervisor [10, 20, 84]; while this approach maximizes flexibility and portability, it comes at a performance cost, especially as cloud workloads become more network intensive.

SR-IOV was invented precisely to remove this involvement [116]: it enables VMs to interact directly and securely with NIC drivers, through the help of an I/O memory management unit (IOMMU). SR-IOV can improve communication latency dramatically. Figure 3.2 shows average communication latency between two VMs running on different servers connected to the same ToR, when both the server CPUs and the network are underutilized (full experimental setup

	Test	Expected	Amazon EC2		OpenStack		
			PV I/O	SR-IOV	PV I/O	SR-IOV unsafe	SR-IOV VNTOR
TCP	Echo test, ingress and egress rules	accept	√	√	√	√	√
	Echo test, ingress or egress rule only	deny	√	√	√	accept	√
	Echo test, no rules	deny	√	√	√	accept	√
	Reverse-path	deny	√	√	√	accept	√
	Invalid, ingress and egress rules	deny	accept	accept	√	accept	√
	Invalid, ingress or egress rule only	deny	√	√	√	accept	√
	Invalid, no rules	deny	√	√	√	accept	√
UDP	Echo test, ingress and egress rules	accept	√	√	√	√	√
	Echo test, ingress or egress rule only	deny	√	√	√	accept	√
	Echo test, no rules	deny	√	√	√	accept	√
	Reverse-path	deny	√	√	√	accept	√
	Enforcement point:		OVS or NIC	NIC	OVS	None	VNTOR

Table 3.1 – Security-group semantics for intra-tenant traffic as exposed by Amazon and OpenStack. The behavior of each system was tested for echo-tests, invalid packets, as well as reverse-path packets after an initial packet. For each of these, certain rules were either installed or not. (√ means that the expected action was observed).

in §3.6.2). For small packets, average latency is 2.5× smaller with SR-IOV, almost matching bare-metal performance, while latency standard deviation is 4× smaller.

SR-IOV has severe implications for security groups: When VMs bypass the hypervisor and talk directly to the NIC, the hypervisor cannot enforce security-group rules. As a result, OpenStack supports SR-IOV (since "Juno" [113]) only at the cost of a fundamental restriction: security groups are entirely disabled (§3.2.4). Amazon supports SR-IOV under the name Enhanced Networking [6] as part of its Virtual Private Cloud product, but implements security groups on a proprietary NIC that is not available outside their own cloud [138].

3.2.4 Validation of the Status Quo

For the record, we tested whether OpenStack and Amazon expose the security-group abstraction as described in § 3.2.1 and whether SR-IOV affects isolation. In each of the two environments, we created two security groups, *A* and *B*, added a VM to each group, and tried to send traffic between them, while varying the group rules. In some tests we added both the ingress and egress rule from Section 3.2.1, in others one of the two, and in others none. In the "echo" tests, *A* establishes a TCP connection with *B* at port 8080. In "reverse-path," *B* sends a packet with source port 8080 to an ephemeral destination port on *A*, without establishing a connection first. In "invalid," *A* sends packets that are neither part of an established connection nor establish a new connection (they are not SYN packets).

Table 3.1 shows the results, which validate our expectations: Both OpenStack and EC2 expose the security-group abstraction correctly in the base case where I/O is emulated through a paravirtual device ("PV IO" columns); in this case, the hypervisor is not bypassed and rule enforcement and connection tracking can be implemented at the hypervisor. The only minor difference is that Amazon does not verify the existence of SYN flags on the initial packet of a TCP connection. However, in the case where the VMs use SR-IOV, OpenStack accepts their addition to the security groups, but then fails to enforce the rules ("SR-IOV-unsafe" column). Amazon does expose the abstraction correctly, even with SR-IOV, but at the cost of custom hardware.

For completeness, the rightmost column in Table 3.1 shows the results for our solution; we include it under "OpenStack" because we have integrated it in that platform.

To conclude: Existing approaches to security groups have fundamental problems: First, none of them are compatible with bare-metal support. Second, in the OVS-based approach, a tenant may escape the confines of their VM, gain unauthorized access to the hypervisor, and bypass

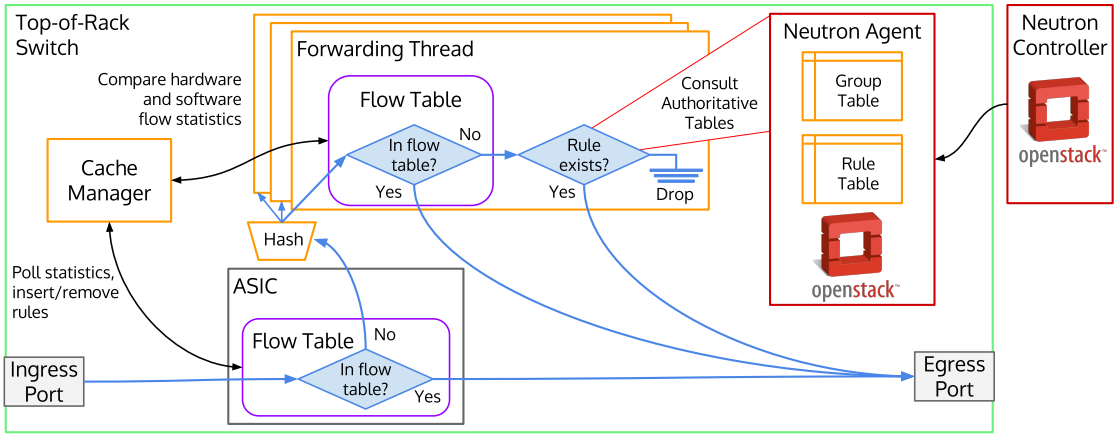


Figure 3.3 – This figure shows the VNToR architecture

security-group rules. Third, to reap the performance benefits of SR-IOV, cloud operators must either disable security groups or deploy custom hardware on their servers.

3.3 Our Proposal: Move to the ToR

We set out to solve these problems by moving the implementation of security groups to the ToR. On top of being physically outside the server—hence compatible with bare-metal support—the ToR has the following advantages as an enforcement point: First, it is the traffic source's entry point into the cloud network, hence it can stop unauthorized traffic from affecting legitimate communications. Second, commodity ToRs are already equipped to enforce access rules (albeit a small number of them) within the switching ASIC at line speed; so, in a sense, we are making access-rule enforcement cheaper by moving it from the server —where it consumes general-purpose processing cycles that could be allocated to other functionality— to the ToR, where it can run on the existing hardware without any performance penalty. We consider our approach aligned with the SDN philosophy of pushing complexity to the network edge: even though we do add functionality to a network device, we do it at the first hop from the server without affecting the network core.

The key challenge is the limited rule capacity of commodity switching ASICs, which also arises in the SDN universe. To address it, SDN proposals use switch datapath memory as a cache for a backing store located at another switch [144] or a processor close to the switch [83, 142]. However, that work focuses on caching algorithms that maintain forwarding semantics in the presence of overlapping rules[2]. In our context, support for overlapping rules is not crucial, as

[2] For example, caching "deny * from 1.1.1.0/24" while not caching "allow * from 1.1.1.1/32" incorrectly denies traffic from 1.1.1.1.

Data struct.	Stored in	Type	Field	Written or updated by	Read or polled by
hwTbl	ASIC	Flow table	*matchAction*	cache manager	hw pipeline
			counter	hw pipeline	cache manager
swTbl	SupE	Hash map	*matchAction*	sw forwarder	sw forwarder
			counter, *weight*, $weight_h$	sw forwarder cache manager	sw forwarder cache manager
			hwEntry	cache manager	sw forwarder
minHeap	SupE	Min-heap, sorted by *ptr* → *weight*	*ptr* (points to *swTbl* entry)	cache manager	cache manager
maxHeap	SupE	Max-heap, sorted by *ptr* → *weight*	*ptr* (points to *swTbl* entry)	sw forwarder	cache manager

Table 3.2 – VNTOR system state

security groups do not currently require it—though it is, of course, a useful feature and can be provided by combining SDN proposals with ours.

What *is* crucial, in our context, is performance, and existing caching systems are not fast enough (whether the rules are overlapping or not). One reason we are moving functionality from the hypervisor to the ToR is to eliminate hypervisor overhead; it does not make sense to cancel out this improvement by slowing down the ToR. To meet this goal, we design a caching system tailored to the properties of state-of-the-art datapath memory, for example, we interleave memory polls and updates to strike a balance between the freshness of the poll results and the delay of the memory updates. Such systems issues have been outside the focus of the related SDN work.

We consider security groups the most challenging network abstraction to implement at the ToR, because it requires both a large number of accept/deny rules (as many as the tenant entity pairs that are allowed to communicate by their security groups) and connection tracking. However, network virtualization involves more than security groups, most importantly layer-2 network overlays; to support these, VNTOR would need to implement virtual extensible LAN (VXLAN) [97], which encapsulates/decapsulates all tenant traffic that belongs to an overlay and traverses the network core. Many current-generation commodity switches already have VXLAN hardware support built-in, hence the only piece we are currently missing is the VXLAN control protocol; we need to either add it to our ToR software stack, or integrate our software stack into an existing one that implements it.

3.4 System Design

In this section, we describe the typical ToR platform that served as our starting point (§3.4.1), the VNTOR architecture (§3.4.3), its two main components in detail (§3.4.4 and §3.4.5), and its integration in OpenStack (§3.4.6).

3.4.1 Platform

A typical ToR consists of a switching ASIC that implements all the line-rate packet processing and a supervisor engine (SupE) that runs all the software needed to manage the ASIC. The ASIC has a (physical) flow table that can fit *pSize* entries, each one storing a rule (a traffic pattern and associated action) and a byte counter. A poll operation on an entry returns the entry's current counter, and an update operation replaces the entry's rule with a new one. The SupE has access to DRAM that can fit $vSize \gg pSize$ entries.

One limitation of current ToR platforms is that a poll or update operation on the ASIC flow table is significantly slower than packet inter-arrival on a multi-Gbps link: the latency of a poll, denoted by t_{poll}, is typically several µs, while the latency of an update, denoted by t_{up}, is tens or even hundreds of µs. We observed such latency numbers in our own prototype, even though we used a state-of-the-art ASIC and controlled the ASIC flow table through the vendor's own software development kit (SDK). One likely explanation is that these operations were traditionally not considered time-critical and were therefore never optimized. Future ToR platforms, with different ASIC hardware/software control mechanisms, should improve these operations, but are unlikely to close the gap between poll/update latency and packet inter-arrival times (as they are unlikely to significantly increase flow-table capacity). We designed VNTOR such that (a) it achieves its goals given the poll/update latency of current ToR platforms and (b) it will be able to leverage the lower poll/update latency offered by future platforms to scale the workload churn that it can handle.

3.4.2 Local vs. Remote Control Plane

The OpenFlow API [98] provides the mechanisms to configure match+action rules of a switch, evaluate the frequency by which certain rules are applied, and even inspect and forward redirected traffic. A Neutron agent could therefore implement cloud virtual networking policies, *e.g.*, security groups, using the API without having to modify the switch. Related research has focused on the scalability of software-defined networks in the context of the limited capacity of the forwarding tables of OpenFlow switches, suggesting changes to the API [36] or ways to partition tables across switches [144].

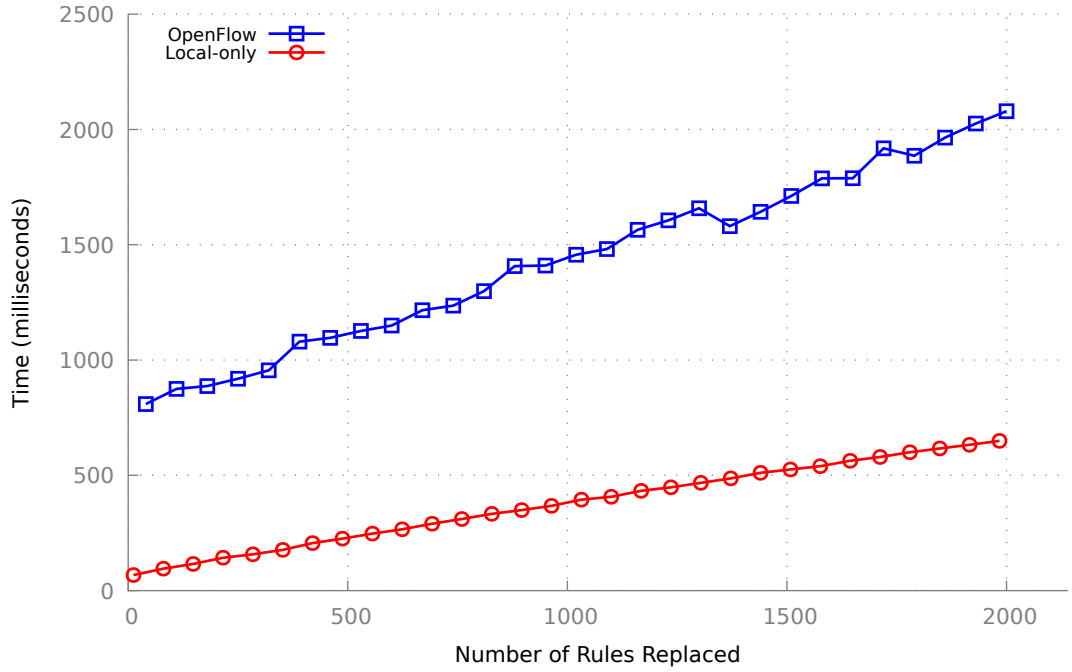


Figure 3.4 – Combined time to poll all TCAM rules and replace n of them with an OpenFlow approach compared to a local client running on the supervisor

Figure 3.4 shows the measured cost of two basic micro-operations that determine the cost of polling and updating flow table rules from a Broadcom Trident+ switch with a low-power supervisor, and is representative of high-volume commercial switches currently deployed in data centers. The micro-benchmark identifies the combined upper bound when polling and updating the ASIC. Figure 3.4 measures the performance an OpenFlow approach in which a remote client repeatedly polls the hardware flow counters and then replaces n rules. In the *local-only* case, the client runs directly on the supervisor, thereby eliminating the serialization and networking overheads of transferring the information through the OpenFlow secure channel.

This benchmark also shows that a centralized approach for cache control causes a high overhead. This overhead is partially due to the controller itself, but also due to the overheads of serializing information on the OpenFlow secure channel and operating the TCP/IP stack from the supervisor. A single polling operation takes $t_r = 40\mu s$ and a single update takes $t_{up} = 287\mu s$. This microbenchmark used batched requests for OpenFlow to minimize the effect of the network. Still, the OpenFlow solution takes $t_r = 389\mu s$, and $t_{up} = 635\mu s$. We see that polling operations are an order of magnitude slower even when amortized in a batch over all TCAM entries, and that rule updates are 2.2× more expensive.

3.4.3 Architecture

Figure 3.3 shows the VNTOR components: apart from the ASIC, there are parallel software forwarders, a cache manager, and a Neutron agent, all running on the SupE. The ASIC performs its lookups in its flow table (from now on "hardware table"). The software forwarders perform their lookups in a hash map of $vSize \gg pSize$ entries that is stored in DRAM (from now on "software table").

The authoritative security-group rules are stored in the Neutron agent; when a new flow is observed, this state is consulted and translated into accept/deny rules that are written in the software table.

The hardware table in the ASIC acts as a cache for the software table in the forwarding threads, and the two of them together implement the virtual flow table abstraction. We refer to rules stored in the hardware table as *cached rules*.

The cache manager continuously polls the hardware table, computes a weight for each rule in that table that reflects the rule's recent popularity, and replaces the lightest cached rules with heavier non-cached rules.

The ASIC handles all traffic that can be served from the hardware table and passes the rest to the software forwarders. More specifically: A packet enters the switch through a port and gets forwarded to the ASIC. If the lookup in the hardware table returns an action other than "forward to the SupE," the packet stays in the ASIC until it is dropped or forwarded. Otherwise, the packet is tagged with its ingress port number and passed to one of the software forwarders, which processes the packet, starting from a lookup. If the software lookup indicates that the packet should be forwarded, the packet is tagged with its egress port number and passed back to the ASIC, which removes the tag and forwards the packet through the designated egress port.

Table 3.2 summarizes system state: *hwTbl* is the hardware table and *swTbl* is the software table. Each element of *swTbl* consists of five fields: *matchAction* (traffic specification and associated action), *counter* (byte counter), *weight*, $weight_h$ (a past value of *weight* used for exponential smoothing), and *hwEntry* (set only if the rule is cached and states the corresponding *hwTbl* entry). Moreover, there are two kinds of heaps that store pointers to *swTbl* entries: First, the cache manager maintains in *minHeap* pointers to recently polled cached rules, sorted by weight, with the lightest rule at the head. Second, each software forwarder maintains in *maxHeap* pointers to non-cached rules, sorted by weight, with the heaviest rule at the head. For brevity, when referring to heap operations, instead of saying that "we insert/remove a node that points to rule R," we say that "we insert/remove rule R."

Algorithm 1 Cache-Management Algorithm

```
1:  while True do
2:    for each rule R cached in hwTbl entry i do
3:      r ← swTbl's entry that stores R's copy
4:      r.counter ← hwTbl.poll(i)
5:      r.weight ← rate · (1 − h) + r.weight_h · h
6:      minHeap.del(oldest element)
7:      minHeap.add(pointer to r)
8:      r_c ← swTbl entry at minHeap.head()
9:      r_c̄ ← swTbl entry at maxHeap.head()
10:     if r_c.weight < r_c̄.weight then
11:       hwTbl.update(r_c.hwEntry,
                       r_c̄.matchAction)
12:       r_c̄.hwEntry ← r_c.hwEntry
13:       r_c.hwEntry ← None
14:       minHeap.del(pointer to r_c)
15:     end if
16:   end for
17: end while
```

3.4.4 Cache Management

Two facts shaped the design of the cache manager:

(a) In an idealized system, Least Recently Used (LRU) is the best replacement policy. Before building our prototype, we simulated an idealized system, where operating on the data structures (including polling/updating *hwTbl*) takes zero time. We gave real data center traces as input to this system and compared the performance of the most intuitive replacement policies. In particular, we experimented with several variants of "least heavily used" (LHU), where each rule has a weight reflecting how many bytes matched the rule recently; this weight may correspond to a sliding time window, or it may be updated at fixed time intervals. LRU outperformed all LHU alternatives, and in retrospect the intuition is obvious: real traffic is bursty, and LRU ensures all traffic bursts (but the first packet) are served from the cache.

(b) In a real system, polling and updating *hwTbl* takes a long time, with two implications for caching. First, we cannot implement LRU, because we cannot update *hwTbl* on every miss: we can perform up to $\frac{1}{t_{up}}$ (a few thousand) updates per second, whereas a switch with multiple 10Gbps ports may observe packet bursts way above this (at the limit, 19.5M minimum-sized packets per second per port). Second, it does not make sense to poll the entire *hwTbl* and then update it: doing so takes hundreds of ms—a very long time in the context of multi-Gbps line rates; by that point, the entries in *hwTbl* would be irrelevant to the currently observed traffic.

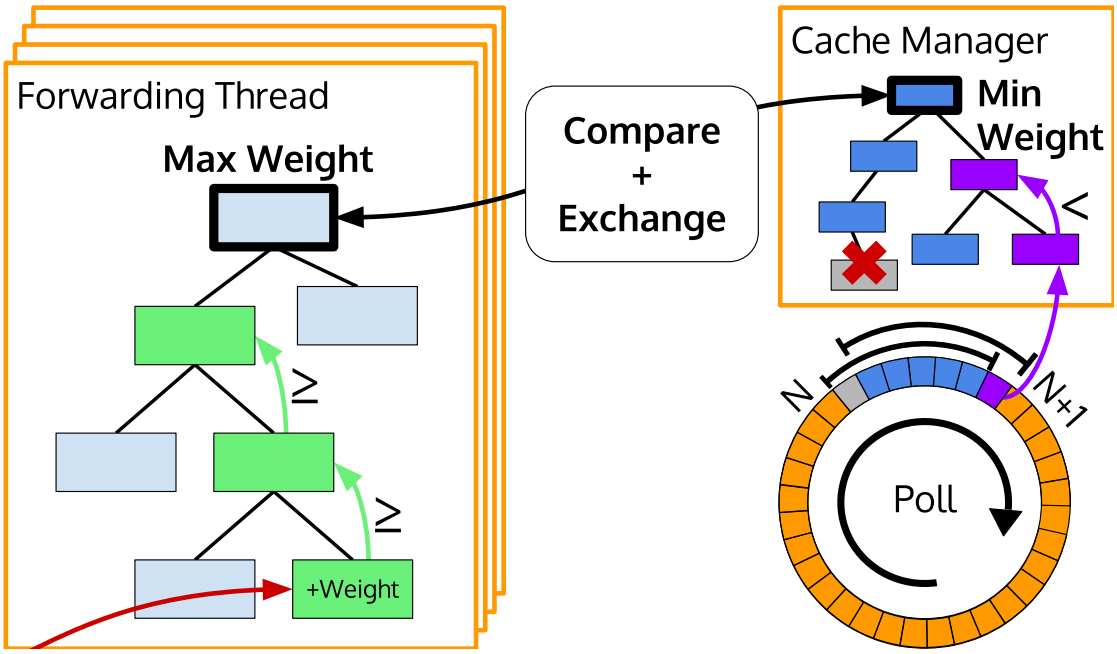


Figure 3.5 – The heap architecture and interactions

Our cache-management algorithm (Algorithm 1, illustrated in Figure 3.5) tries to approximate LRU as well as hardware limitations allow and to make timely updates based on fresh statistics. More specifically: The cache manager continuously polls *hwTbl* (lines 1,2). After polling cached rule R, it first finds and updates R's copy in *swTbl* (lines 3–5). Next, it updates *minHeap* (lines 6,7). Finally, it compares the lightest cached rule (line 8) to the heaviest non-cached rule (line 9); if the latter is heavier (line 10), then the two rules are "swapped," *i.e.*, one is evicted from *hwTbl* and the other inserted in its place (line 11). In case of a swap, the cache manager also updates *swTbl* and *minHeap* accordingly (lines 12–14). A minor point: For simplicity, line 8 references a single *maxHeap*, whereas in reality there are as many as the software forwarders, and the cache manager picks the one with the heaviest rule at the head.

A rule weight reflects the rule's current popularity, smoothed with standard exponential averaging: In line 5 of algorithm 1, *rate* is the traffic rate currently matching the rule (computed from *counter*), while $h \in [0,1)$ is a history factor that determines how much the past matters. History ($weight_h$) is updated (set to *weight*) at fixed time intervals of duration T_h.

We define the *freshness* of cached rule R at a given point in time as follows: consider R's copy in *swTbl* and its *counter* field; R's freshness is the amount of time from the moment *counter*'s value was read from *hwTbl*. Freshness captures the relevance to current traffic of a rule's statistics stored in *swTbl*. The best (smallest) possible freshness is t_{poll}, which holds as soon as the cache manager updates *counter* (line 3).

The size of *minHeap* involves the following trade-off: the smaller it is, the fresher but also the fewer the cached rules considered for eviction. For instance, if $||minHeap|| = 1$, the only cached rule considered for eviction is the one that was just polled; on the positive side, this rule is as fresh as possible; on the negative side, it is unlikely that this happens to be the lightest cached rule, so replacing it is a suboptimal decision. At the other end of the spectrum, if $||minHeap|| = pSize$, the cache manager considers all the cached rules for eviction, some of which were polled hundreds of ms earlier.

Regarding complexity, the most expensive operations performed by the cache manager are the insertions and deletions on *minHeap*, which are $O(\log||minHeap||)$.

3.4.5 Software Forwarding

On top of processing packets experiencing a miss in the hardware pipeline, each software forwarder helps the cache manager by lazily updating some of the cache-related state. More specifically, after receiving a new packet, a software forwarder: (a) Looks the packet up in *swTbl*; if there is no matching entry, it checks the Neutron agent's authoritative state, translates it into accept/deny rule(s), and inserts them in *swTbl*. (b) Identifies the best matching rule R in *swTbl* and updates the *counter* and *weight* fields. (c) If R is not in *maxHeap*, it is inserted. If R is in *maxHeap* and its weight increases, *maxHeap* is rebalanced. (d) Checks whether the rule at the head of *maxHeap* has been cached by the cache manager and, if so, removes it from *maxHeap*. The interaction between the cache manager and the software forwarders is illustrated in Figure 3.5.

The lazy update of the *weight* field may put the data structures in a non-intuitive state. First, *maxHeap* is rebalanced only when a *weight* field increases. Second, the *swTbl* entry that corresponds to a non-cached rule R is updated either when a packet that matches R happens to arrive, or, potentially, when *maxHeap* is rebalanced—hence, if R is idle, its *weight* may become obsolete. As a result of these two points, the *swTbl*'s *weight* fields are not guaranteed to reflect the rules' actual weights, and *maxHeap* is not guaranteed to be a heap.

Nevertheless, laziness does not affect the cache manager's replacement decisions. In the end, the only non-cached rule considered for caching by the cache manager is the rule at the head of *maxHeap*. Hence, what matters is that this rule is the one with the biggest actual weight and the rule's *weight* field is up to date. Both of these properties are guaranteed, because the moment a new packet matches rule R and makes R the heaviest rule, it also causes R's *weight* field to be updated and R to move to the head of *maxHeap*.

Regarding complexity, the most expensive operations performed by a software forwarder are the ones on *maxHeap*, which are $O(\log||maxHeap||)$, where $||maxHeap||$ is equal to the number of non-cached rules (*i.e.*, flows) handled by the forwarder.

3.4.6 OpenStack Integration

Our design is not tied to any particular management system, but we integrated it in OpenStack because it is the open-source standard and is increasingly deployed in industry.

Integrating VNTOR in OpenStack merely requires implementing a Neutron agent that runs on VNTOR's SupE and whose role is to collect, from the Neutron controller, all the relevant security-group state. In particular, our agent subscribes for changes to:

- all the local VMs, *i.e.*, those running in the local rack;
- membership information of the security groups where the local VMs belong;
- membership information of any security group with which a local VM is allowed to communicate;
- all the other relevant network-wide translations defined by Neutron, *e.g.*, encapsulating tunnels.

3.5 Prototype Implementation

In this section, we provide details on our prototype's hardware (§3.5.1) and software (§3.5.2), then describe how we achieve traffic separation (§3.5.3).

3.5.1 Hardware

Our underlying ToR is a 1U reference kit that encloses a Broadcom Trident+ switching ASIC with 64 × 10Gbps ports [16] and an XLP SupE [17]. The ASIC provides full bisection bandwidth between the ports and has a hardware flow table with the properties stated in Table 3.3. The SupE has a Netlogic XLP MIPS processor with 4 cores, 16 hardware threads, and 1024MB of DRAM; it runs Linux and Broadcom's SDK.

One limitation of our reference kit (that did not affect our design) is that the ASIC and the SupE communicate over a PCIe bus that allows maximum throughput 600Mbps or 170Kpps in each direction. This is a limitation of the printed circuit board, not of the integrated circuits: both

Parameter	Value
#entries in *hwTbl* (*pSize*)	2048
Avg. *hwTbl* poll latency (t_p)	40µs
Avg. *hwTbl* update latency (t_{up})	287µs
#entries in *swTbl* (*vSize*)	1.2M
#eviction candidates (||*minHeap*||)	1024
History factor (h)	0.8
Aging interval (T_h)	100ms

Table 3.3 – Prototype properties and configuration

the ASIC and SupE are fully equipped to exchange traffic at 10Gbps, but the SupE's built-in 10GbE NIC is unfortunately not connectable in our reference kit. Hence, our prototype could handle a significantly higher cache miss rate with a proper printed circuit board.

3.5.2 Software

VNToR software is a multi-threaded process with one thread implementing the Neutron agent; one thread implementing the cache manager (§3.4.4); four threads implementing software forwarding (§3.4.5); and one thread receiving traffic from the ASIC and dispatching it to the forwarding threads in a flow-consistent manner.[3] Each software thread is pinned to one SupE hardware thread (so we use only 5 of the 16 hardware threads available). Given the limited ASIC-SupE interconnect of our prototype (§3.5.1), a single forwarding thread could have handled all traffic coming from the ASIC, but we implemented 4 threads in order to validate our design, which calls for parallel software forwarders.

Threads communicate over shared memory, mostly in a lock-free manner. The forwarders and the cache manager communicate without locks: a forwarder indicates which is the heaviest non-cached rule by setting the *maxHeap* top, while the cache manager signals the promotion of a rule to *hwTbl* by setting the *hwEntry* field of the rule's copy in *swTbl*. Due to consistent hashing of flows to forwarders, the forwarders split the *swTbl* in separate parts, eliminating the need for locks. The Neutron agent does lock the authoritative security-group state in order to update it (blocking the forwarders from reading it), but this happens only when security groups for VMs in this rack change, hence it does not affect our prototype performance—and, in any case, we could use standard techniques like shadow tables to eliminate this lock.

The forwarders implement connection tracking following the iptables ESTABLISHED semantics: When a forwarder receives a new packet, it first checks whether a matching entry exists

[3] The dispatch thread emulates receiver-side scaling [103], which is normally provided by the NIC hardware and, hence, would be unnecessary if our SupE's NIC was connectable.

in *swTbl*. If not, and this is a UDP packet or a TCP SYN packet, it reads from the Neutron agent's state the source's and the destination's security groups, say A and B, and their rules. If, according to the rules, the new flow should be allowed, it inserts an exact-match rule into *swTbl* that allows traffic from the source to the destination; in the TCP case, should the corresponding SYN/ACK packet be received during the standard 30 second window, it also inserts an exact-match reverse rule. The exact-match rules are removed from *swTbl* following TCP FIN packets or timeouts.

We set the configuration parameters (Table 3.3) as follows: The size of *swTbl* is determined by the available DRAM. We set $||minHeap||$ such that the freshness of the rules in *minHeap* (§3.4.4), *i.e.*, of the cached rules considered for eviction, is on the same time scale as traffic bursts—about half a second; this ensures that we do not evict a rule that is currently matching an ongoing traffic burst but was polled before the burst started. We set the aging interval T_h to the average time between polls of the same *hwTbl* entry, *i.e.*, history is updated every time the cache manager completes an iteration over *hwTbl*.

3.5.3 Traffic Separation

In the case of VMs running on the same physical machine, a performance enhancement in SR-IOV NICs can potentially circumvent security groups: when an SR-IOV NIC receives a packet coming from a local SR-IOV device (a local VM) that has a destination MAC address associated with another local SR-IOV device, the NIC short-circuits the packet from one device to the other without sending it to VNTOR.

To avoid this, we assign to each SR-IOV virtual function (each VM) a rack-scoped VLAN ID that uniquely identifies all traffic from the VM, as proposed in FasTrak [109]. This prevents traffic between local VMs from being short-circuited (because the destination always belongs to a different VLAN from the source), ensuring that security groups are applied correctly. Neutron manages the namespaces and deploys the VLAN ID assignment to VNTOR and the hypervisors. A VM cannot change its VLAN ID (only the hypervisor can do that), which has the useful side-effect that VMs cannot spoof each other's identities (in a traditional virtualization platform, spoofing is prevented through ingress filtering at the hypervisor, however, since we are bypassing the hypervisor, we need a different approach).

3.6 Evaluation

After describing our experimental setup (§3.6.1), we answer four questions: (§3.6.2) How much does VNTOR improve communication performance given a static workload? (§3.6.3) Does

HW	Dual-socket Intel Xeon E5-2637v2 @ 3.5Ghz, 8 cores, 64GB RAM, Intel 82599 10GE NIC, TurboBoost and intel_pstate disabled to reduce variance, IOMMU enabled for SR-IOV.
SW	Ubuntu 15.10 "Wily" with 4.2.0-25-generic SMP kernel, OVS version 2.4.0, TCP Nagle disabled where applicable to improve performance [109].

Table 3.4 – Server characteristics

VNTOR maintain this advantage as the workload becomes increasingly dynamic? What are its performance limits? (§3.6.4) Does VNTOR's caching algorithm work well with realistic data center traffic?

3.6.1 Experimental Setup

We use a 10GE switch (characteristics in Table 3.3), running either a standard layer-2 stack or our VNTOR software; and 10 servers (characteristics in Table 3.4) connected to the switch. When the servers exchange all-to-all traffic and bottleneck on I/O, the maximum achievable aggregate throughput[4] is close to 187Gbps. vCPUs are always pinned to physical cores to minimize variance across experiments due to scheduling.

To generate traffic, we use netperf [124] and iperf3 [46], respectively, for simple latency and throughput measurements; a benchmark we wrote based on libevent for measuring performance during churn; and tcpreplay for replaying traffic traces. We use OpenStack "Liberty" to manage the infrastructure (deploy VMs and configure security groups).

We compare four setups:

- **OVS**: Standard OpenStack setup. Traffic exchanged between VMs. Security groups enforced at OVS.
- **SR-IOV-unsafe**: Traffic exchanged between SR-IOV-empowered VMs. No security groups.
- **SR-IOV+VNTOR**: Traffic exchanged between SR-IOV-empowered VMs. Security groups enforced at VNTOR.
- **Metal+VNTOR**: Traffic exchanged between native processes. Security groups enforced at VNTOR.

[4] We mean *data* throughput, which is the full bisection bandwidth of 200Gbps minus protocol overhead.

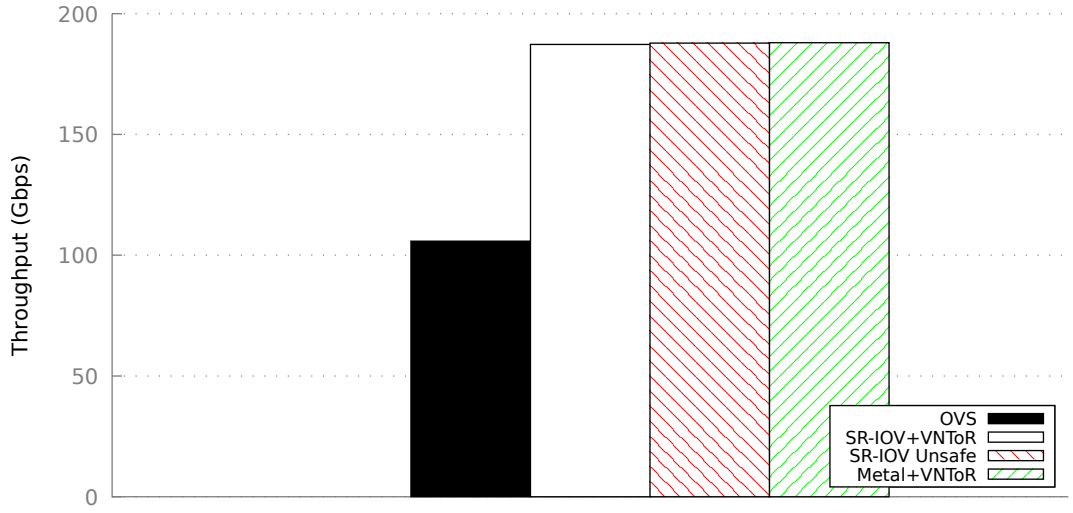


Figure 3.6 – iperf3 streaming benchmark

OVS is the typical cloud setup (§3.2.2) whose performance we aim to improve. In SR-IOV-unsafe, communication is empowered with SR-IOV at the cost of no security-group enforcement (§3.2.3). The last two setups correspond to our system with and without virtualization at the end-point. In all experiments, there are two access rules for each generated TCP connection, like the ones in § 3.2.1.

We want to show that:

- SR-IOV+VNTOR outperforms OVS, *i.e.*, enforcing security groups at the ToR improves performance;
- SR-IOV+VNTOR performs as well as SR-IOV-unsafe, *i.e.*, in our solution, security does not come at the cost of performance;
- SR-IOV+VNTOR performs close to Metal+VNTOR, *i.e.*, in our solution, the cost of virtualization is minimal.

3.6.2 Baseline: Static Workload

First, we test how much VNTOR improves latency and throughput given a static workload, where all the access rules fit in the hardware table.

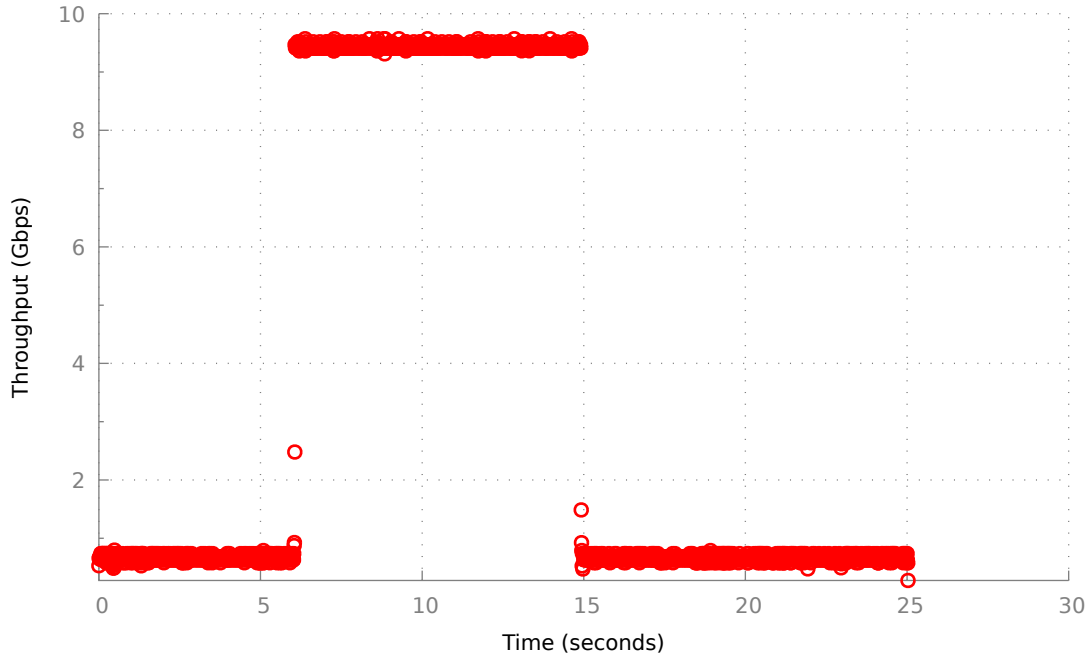


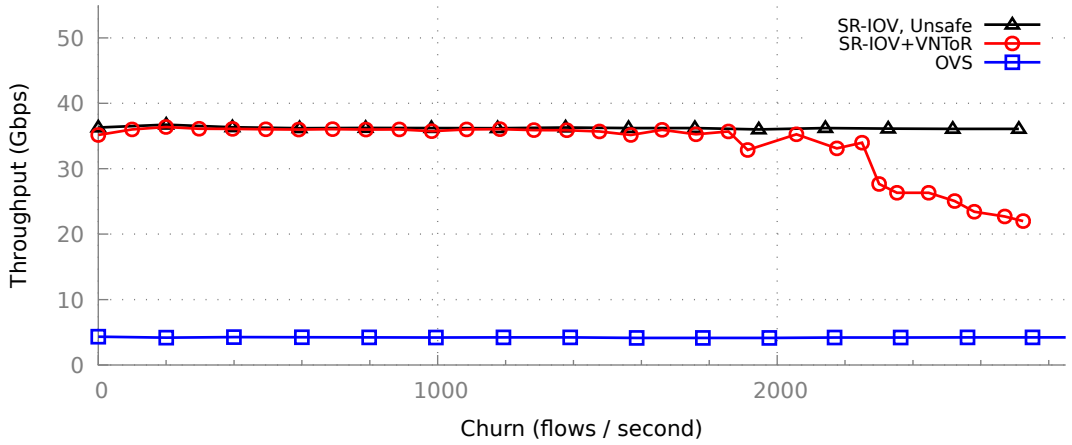
Figure 3.7 – Throughput of a unidirectional TCP flow. VNToR promotes the rule at ~ 6sec and demotes it at ~ 15sec.

We measure with netperf the request/response latency between two processes running on different servers[5], which is typically used to characterize communication performance in data-centers [67, 93, 109]. In these experiments, server CPUs and I/O buses are lightly loaded.

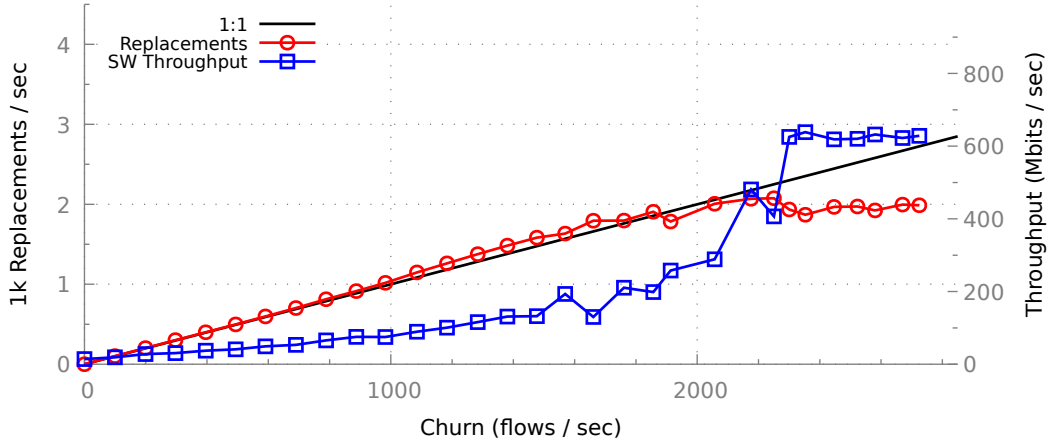
Figure 3.2 shows the results for different message sizes, and we see that SR-IOV+VNToR significantly outperforms OVS. For the smallest message size, SR-IOV+VNToR average latency is 2.5× smaller and latency standard deviation 4× smaller. In this case, the dominant factor is the latency introduced by the networking stack, which is significantly lower in SR-IOV+VNToR due to passthrough (§3.2.3). The advantage decreases for larger message sizes, as transfer time also becomes a significant factor; still, for 32KB messages, SR-IOV+VNToR average latency is 1.6× smaller and latency standard deviation 2.3× smaller.

Next, we measure with iperf3 aggregate throughput when 20 process pairs exchange traffic at the highest possible rate, each pair over 5 parallel TCP connections. In these experiments, when the processes run natively, the servers bottleneck on I/O and they achieve the maximum possible aggregate throughput of 187Gbps—this is precisely why we chose this experiment configuration.

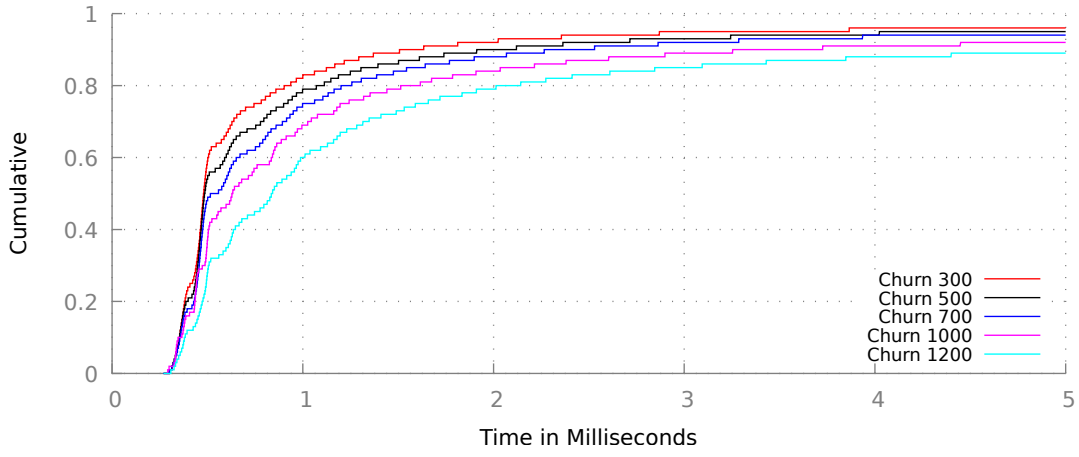
[5] One process sends messages, one at a time, and the other responds. Request/response latency is the amount of time from the moment the first process sends a message until it receives a response.



(a) Aggregate throughput vs. churn



(b) Replacement rate vs. churn



(c) CDF of time to cache

Figure 3.8 – Request/response benchmark with churn

Figure 3.6 shows the results: Metal+VNTOR achieves the maximum possible throughput, while SR-IOV+VNTOR is effectively equal and 1.8× better than OVS.

From both figures we see that SR-IOV+VNTOR performs the same as SR-IOV-unsafe, which confirms that enforcing security groups at the ToR does not introduce any latency (relative to no enforcement). This is not a surprise: in these experiments, the few access rules involved fit and reside in the hardware table, hence are guaranteed to be processed at line speed.

3.6.3 Churn and Breaking Point

Second, we study how VNTOR behaves given a dynamic workload, where the access rules do not fit in the hardware table and the working set keeps shifting.

We start with a microbenchmark to verify that VNTOR behaves as expected when a rule is promoted to the hardware table and when it is demoted back to the software table. We use iperf3 to establish a single TCP connection between two processes running on different servers and stream traffic from one to the other at the highest possible rate. We use tcpdump to capture traffic at the two end-points so that we measure packet drops, retransmissions, reordering, and throughput at 10ms intervals. Instead of letting VNTOR run its caching algorithm, in this particular experiment, we explicitly instrument the cache manager to cache and then evict the relevant rule at particular points in time.

Figure 3.7 shows the results, which are as expected: Throughput is 600Mbps (limited by the PCIe bottleneck between ASIC and SupE inside the switch) when the rule is not cached, and 10Gbps (limited by the line rate) when the rule is cached. The moment the rule is promoted to the hardware table, there is packet reordering (directly proportional to the amount of buffering in the software forwarder and the hardware-table update latency t_{up}), but throughput climbs to line rate within 20ms. Similarly, the moment the rule is demoted to the software table, there are packet drops and retransmissions, yet throughput stabilizes back down to the software level within a few ms without additional penalty.

Next, we create a special benchmark for measuring VNTOR's performance during a dynamic workload. A fixed number of process pairs establish TCP connections between them and exchange requests and responses over each connection. Some of these connections are "elephants," *i.e.*, they exchange n-byte messages as fast as possible, and some are "mice," *i.e.*, they exchange "minimal traffic," which we define as one small message per second. Connections change from elephant to mouse and vice-versa, such that there are always e elephant and m mouse connections. The *churn* is the rate at which elephant flows become

mice and vice versa. We run this experiment with different churn levels, and we observe the request/response latency and throughput achieved by the servers, as well as VNTOR behavior.

We set the benchmark parameters such that the dominant factor in server performance is the latency introduced by the networking stack at the endpoints and packet processing at VNTOR, not by other server or switch bottlenecks. In particular, we ensure that:

- the working set of the rules fits in the hardware table;
- the ASIC-SupE interconnect inside the switch is not saturated;
- the servers are neither CPU- nor I/O-bottleneck-ed.

We present results for $e = 700$ elephant connections, which yields a working set of 1400 rules, *i.e.*, 70% hardware-table occupancy; $m = 50\,000$ mouse connections, which yields a few tens of Mbps of mouse traffic; and $n = 2897$ bytes (two MTUs), the maximum message size for which the servers are not CPU-bottlenecked.[6]

We should clarify that the point of the benchmark is *not* to study the scenario where the working set of elephant flows exceeds the hardware table. VNTOR design does not make sense in that case: a significant amount of traffic misses at the hardware table and is redirected to the software forwarders, saturating the ASIC-SupE interconnect. Instead, the point of the benchmark is to study the scenario where the *total* number of rules far exceeds the hardware-table size, while the working set fits but keeps shifting, *i.e.*, measure VNTOR's ability to quickly identify and cache a highly dynamic working set from among a significantly bigger set.

Figure 3.8a shows aggregate throughput as a function of churn, and we see that SR-IOV+VNTOR clearly outperforms OVS for all churn levels. For low churn levels, this is not surprising as the results presented earlier in Figure 3.2 show that SR-IOV+VNTOR has significantly lower request/response latency, hence the throughput achieved by a set of connections exchanging request/response traffic is bound to be significantly higher. What *is* perhaps surprising is that VNTOR maintains the same throughput (which is on par with SR-IOV-unsafe) until churn reaches 2200 flows/sec (each elephant connection lasts roughly 640ms on average), before degrading gracefully, sufficient for the expected churn at a ToR [80].

Figure 3.8b maps the breaking point of 2200 flows/sec to internal VNTOR behavior: The y-axis is the rate at which cached rules are replaced, and the x-axis is churn. The 1:1 line corresponds to an ideal system that caches a new rule for every new elephant flow (hence keeps all elephant

[6] Servers exchange request/response, not bulk traffic. As messages get larger, copying packets to/from the NIC becomes the dominant factor in request/response latency and saturates the CPU.

	univ1	univ2
Number of flows	556 602	190 064
Duration (sec)	3914	9479
Flows under 1KB	50%	85%
Acceleration	15×	6.5×
Average accelerated rate (Mbps)	378	406

Table 3.5 – Statistics for univ1 and univ2 traces from [14].

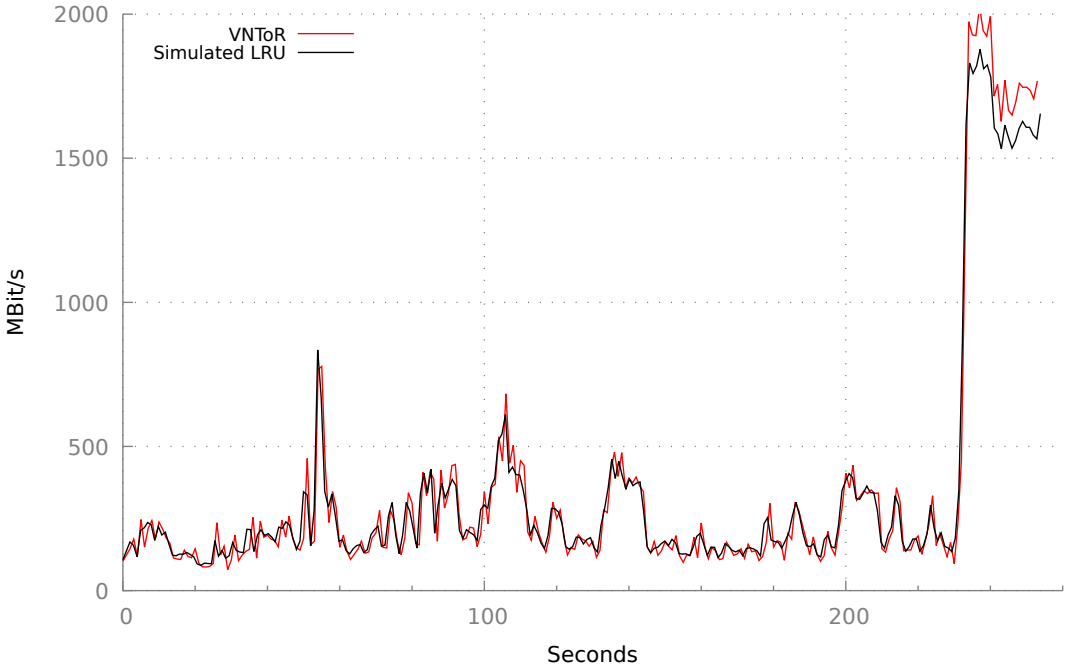
traffic in the hardware pipeline). We see that VNToR follows this line closely until it reaches 2200 replacements/sec and then starts falling behind. The impact of delayed and missed replacements is directly visible on the software-forwarder traffic, which shoots up at that exact moment (because the hardware pipeline cannot handle all elephant traffic any more) and quickly saturates the ASIC-SupE 600Mbps bottleneck.

Finally, Figure 3.8c shows a rule's *time-to-cache* (TTC), measured from the reception of the corresponding flow's first packet until the update call to the hardware table completes. TTC includes the time it takes for the software forwarder to promote the rule to the top of the max-heap, and for the cache manager to consider the rule and cache it. The hardware-table update latency t_{up} = 300µs constitutes a lower bound for TTC. We see that for a churn of 1000 flows/sec, the median TTC is 628µs (about twice the lower bound), the 90th percentile is 3.2ms, and the 95th percentile is 10.4ms.

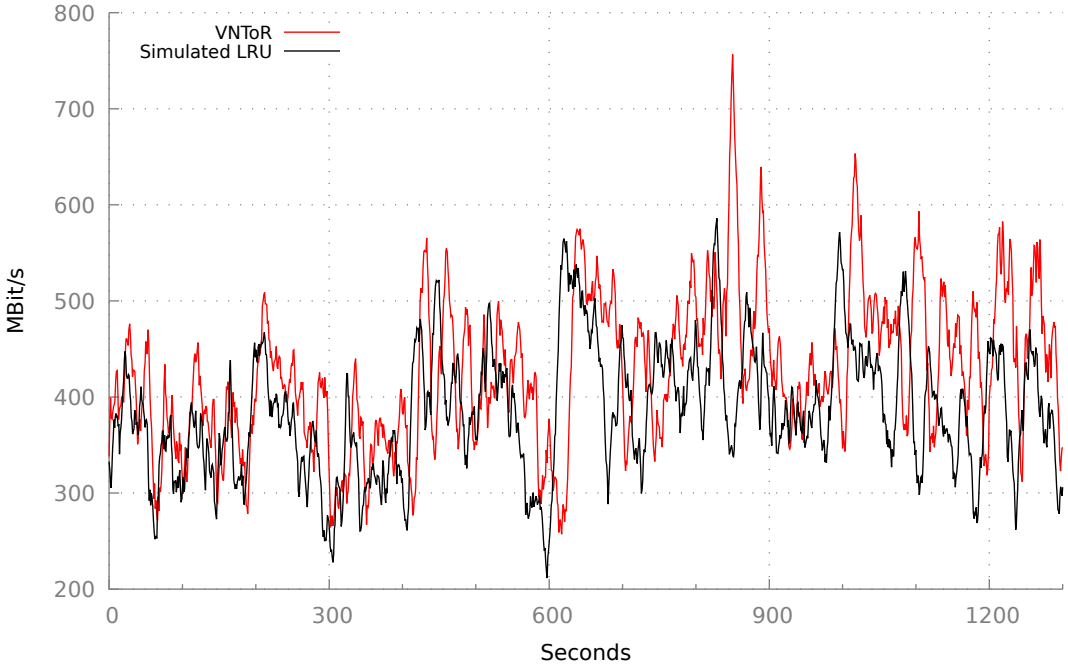
3.6.4 Trace-based Study

Finally, we assess VNToR's caching performance given realistic data-center traffic. We use the only two publicly available data-center traces that we are aware of, captured at a university data-center in 2010 [14]; Table 3.5 states their main characteristics. These traces have many short flows and very high churn, which makes them interesting workloads for caching systems. As they are several years old, we "accelerated" them as much as allowed by our server hardware and pcap-based replay software (by a factor of 15 and 6.5, respectively) while preserving inter-flow arrival times, thereby creating two more traces.

VNToR handled all four traces with zero packet drops. While this is an expected outcome given the low rate of the traces, the interesting question is whether VNToR's caching works well for the traces, *i.e.*, whether it succeeds in serving most of the traffic from the hardware table. To answer, we measured VNToR's cache hit rate and replacement rate, and we compared them to those of an idealized system with an equally-sized cache and an LRU replacement



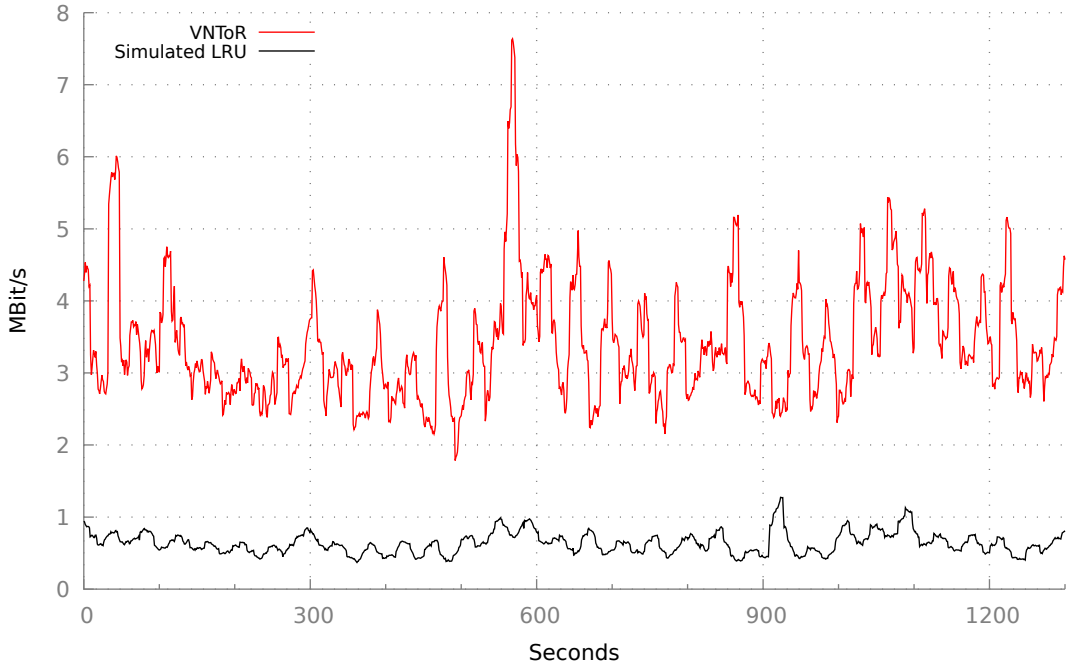
(a) Univ 1



(b) Univ 2

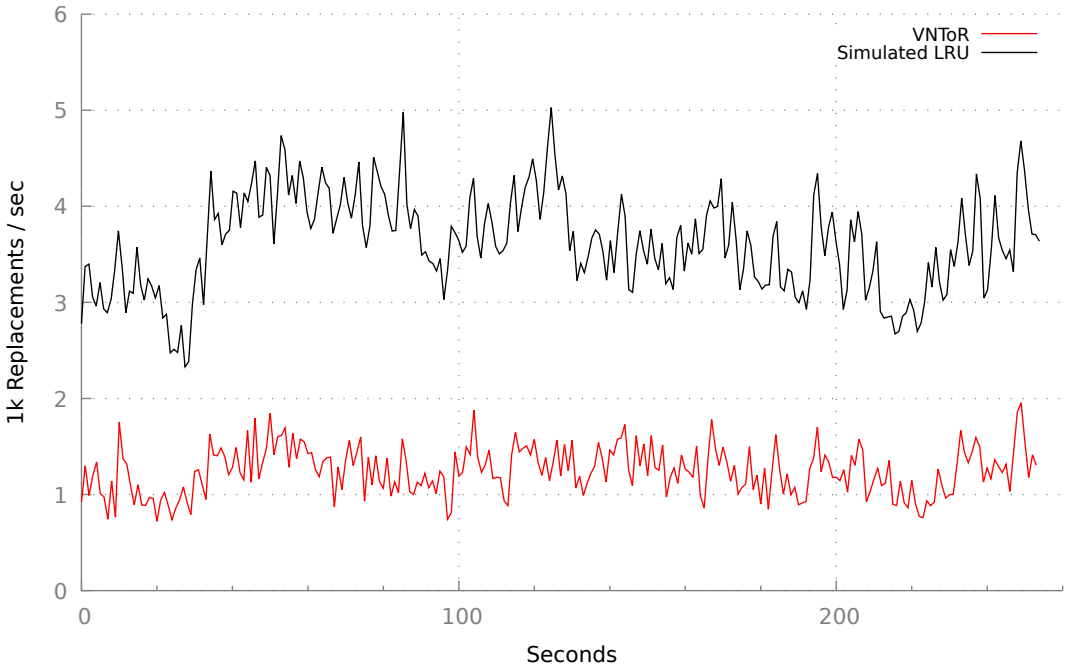
Figure 3.9 – Comparison of VNToR and idealized-LRU cache hit traffic for the accelerated "univ1"and "univ2" traces from [14]

(a) Univ 1

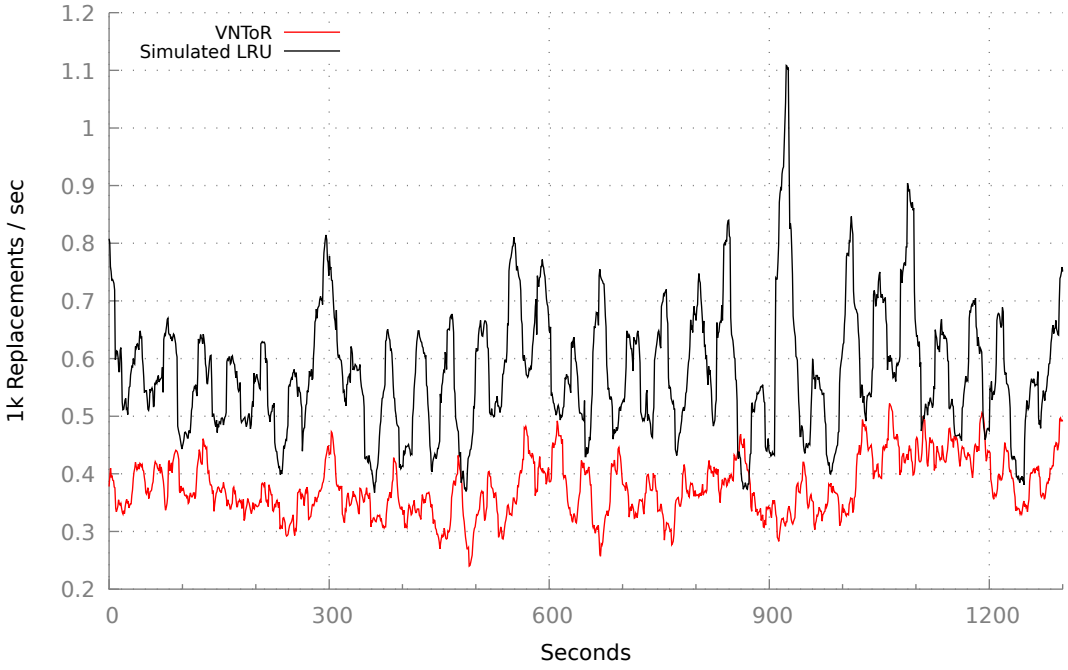


(b) Univ 2

Figure 3.10 – Comparison of VNToR and idealized-LRU cache miss traffic for the accelerated "univ1"and "univ2" traces from [14]



(a) Univ 1



(b) Univ 2

Figure 3.11 – Comparison of VNToR and idealized-LRU replacements per second for the accelerated "univ1"and "univ2" traces from [14]

algorithm. We chose this comparison point because it outperformed all idealized alternatives we simulated (§3.4.4).

For all four traces, traffic has sufficient locality to be served mostly from the hardware table. For the two original traces, both VNTOR and LRU achieve near-perfect hit rate, so we only show results for the two accelerated traces: cache hit traffic (handled by the hardware pipeline) in Figure 3.9, cache miss traffic (handled by a software forwarder) in Figure 3.10, and replacement rate in Figure 3.11. We see that VNTOR's cache miss traffic is up to a few tens of Mbps (Figures 3.10a and 3.10b), only a small fraction of the cache hit traffic (Figures 3.9a and 3.9b). As expected, VNTOR trades off hit rate for a lower replacement rate: VNTOR's miss rate is 4–5 times higher than the idealized LRU's, but its replacement rate is 2–3 times lower (Figures 3.11a and 3.11b). As a side-note, in one of the traces, traffic from elephant flows spikes at the end, but is absorbed by the cache (Figure 3.9a).

3.7 Related Work

FasTrak also moves the implementation of network abstractions to the ToR, but only for a small number of latency-sensitive flows [109]. Hence, that work does not face the challenge of fitting a large number of access rules in the ToR's limited datapath memory.

We have already discussed how we relate to SDN proposals in § 3.3: Several use the switch datapath memory as a cache for a backing store located at another switch, in the case of DIFANE [144], or a processor close to the switch, in the case of CAB [142] and CacheFlow [83]. vCRIB provides the abstraction of a centralized rule repository, while the rules underneath are deployed on both hypervisor and switches [108]. In general, the goal is that the network as a whole exposes the abstraction of a single "Big Switch" [23, 106] implemented in a distributed manner [81]. We share the general challenge but have a different focus from this work: we design and build a caching system that meets a significantly harder performance baseline, and we achieve this by tailoring the solution to the properties of state-of-the-art datapath memory.

The limited computational power of switch supervisor engines is a recurring motivation for offloading functionality to higher-performing, centralized controllers [36, 144]. Indeed, FlowVisor [129] rewrites all rules attempting to process traffic at the SupE to redirect this traffic to the controller. Our evaluation suggests that a modern SupE provides sufficient computational power to maintain the state that is necessary for making fast and intelligent resource-management decisions.

ServerSwitch [95] inserts a switching ASIC as a PCIe card within a server and can offload excess rules to the CPU [96]. In contrast, VNTOR uses a high-volume commercial 10GigE ASIC in a 1RU form factor and does not require an expensive, power-hungry server.

Devoflow [36] raises the level of abstraction in the protocol between the SDN controller and the switches, in part by converting wildcard rules into exact-match rules locally within the switch. Similarly, VNTOR similarly converts OpenStack policies into local match/action rules.

Traffic classification for identifying "heavy" elephant flows is a well-studied problem [18, 89, 115, 139]; VNTOR's approach is pragmatic and emphasizes quick, online decisions that scale to large numbers of flows.

3.8 Conclusion

We presented the design and implementation of VNTOR, a ToR for cloud networks that implements security groups. Unlike the traditional approach of implementing security groups at the server (within the OS that hosts tenant entities), VNTOR enables bare-metal support, avoids the performance overhead that results from involving the host OS, and reduces susceptibility to hypervisor exploits. We presented a VNTOR prototype built on top of a standard ToR and integrated in OpenStack. We showed that our prototype implements security groups correctly, while offering latency and throughput close to those of the underlying switching ASIC; as a result, it significantly outperforms a traditional, state-of-the-art server-based implementation.

4 CRISS-CROSS: Dissolving data-center pods while maintaining the hierarchy of a data-center fabric

4.1 Introduction

Large internet corporations rely on hyperscale data centers to provide their services. These data centers grow ever larger and have become major cost centers [62], with a significant part of the cost being due to their networks. Given the central role of networking, data center network designs have to balance performance with the cost of ownership.

Hyperscale data center networks typically use folded Clos topologies [1, 61, 125, 131], which provide modularity and a hierarchical structure, allowing for easy calculation of routes, cabling and installation. Given current data center size requirements and radix limitations, these Clos topologies have at least three layers: at the lowest layer, the servers of the data center are grouped into racks, each connected to a top-of-rack (ToR) switch; ToR switches (and their servers) are grouped into collections called *pods*, each connected to a set of fabric switches that make up the second layer; the fabric switches are then connected to spine switches that make up the top layer.

To balance cost with scale, data center networks typically have to oversubscribe the traffic at the fabric switches. For example, with an oversubscription of 1:3, there is 3× more bandwidth between ToRs and fabric switches than there is between fabric switches and spines [7, 131]. The amount of oversubscription is a strategic decision that is made at deployment time and commits the data center for multiple years, well before the actual networking requirements of the future workloads are known in practice.

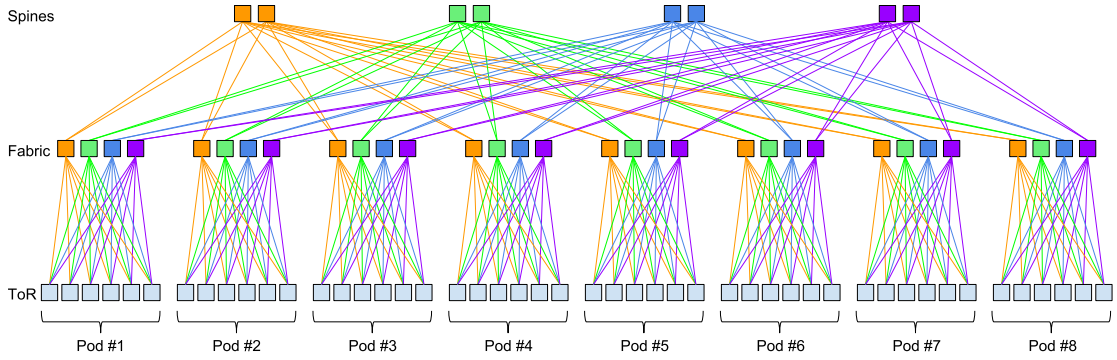


Figure 4.1 – A typical display of a Clos topology where the fabric switches are grouped with their respective pods. The colors signify which spine group a fabric switch belongs to.

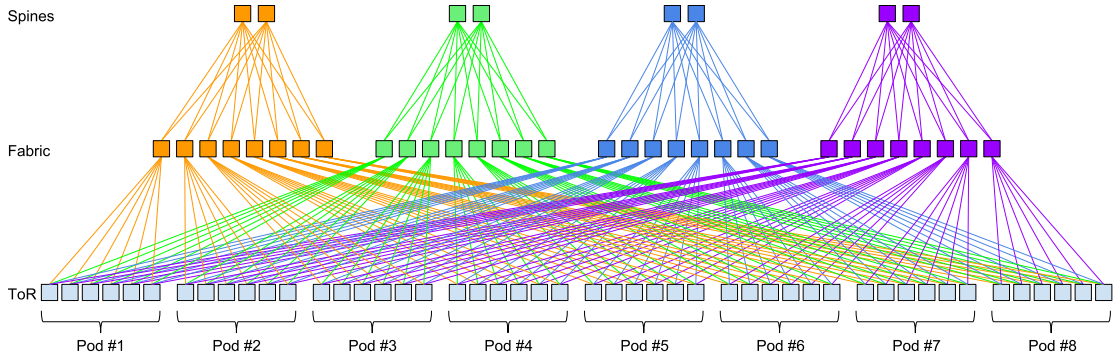


Figure 4.2 – Typically, the fabric switches in Clos topologies are grouped with their respective pods. However, it's also possible to group them with their respective spine switches shown here. Spine switches and their fabric switches form different spine groups, signified by their different colors

Oversubscription, however, means that there is no full bisection bandwidth between the pods [50, 82], which is problematic for traffic patterns with significant inter-pod traffic: such traffic has to cross the spines and share the limited bandwidth between fabric switches and spines, potentially leading to congestion, reduced flow completion times and, therefore, reduced performance.

The natural way to address this problem is to break away from the strictly modular, hierarchical nature of Clos topologies. One option is to replace fabric and spine switches with a random network of small diameter [133, 137]. Another option is to dynamically adapt the network topology to the current traffic pattern, either by directly interconnecting some pairs of ToRs through circuits [50, 134], mirrors [145], or discoballs [57], or by replacing part of the Clos topology with a random network [141]. These are all elegant solutions that improve performance relative to an oversubscribed Clos topology of similar cost; this improvement,

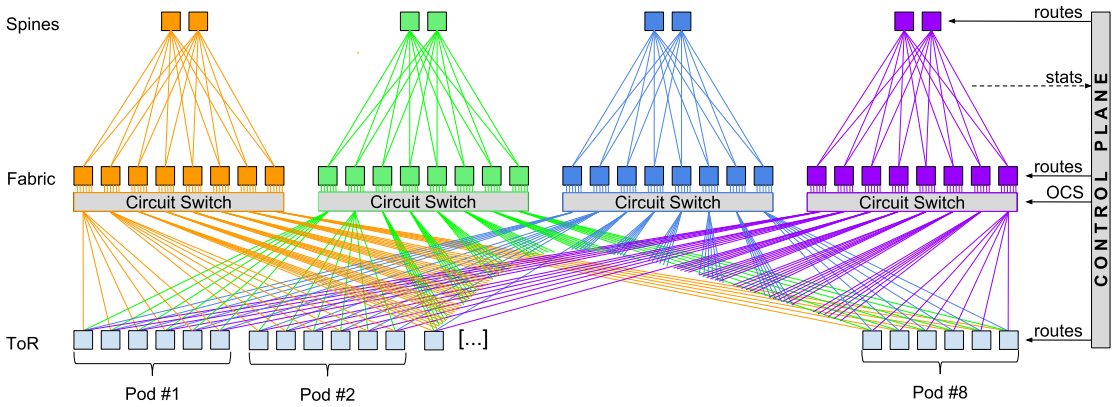


Figure 4.3 – Architecture of CRISS-CROSS, illustrated with 8-radix switches and fabric oversubscription of 1:3, resulting in a deployment using 4 spine groups (each with their OCS module) for 48 racks. When the OCS module is configured in its baseline passthrough state, the topology is a standard folded-Clos topology where the 48 racks are organized in 8 pods.

however, requires giving up, more or less, the simple routing and physical deployment that come with the modular, hierarchical nature of Clos topologies.

In this chapter, we ask the question: can we eliminate the spine bottlenecks that result from oversubscription while maintaining the modularity and hierarchy of a Clos topology? We propose CRISS-CROSS, an approach that adapts the network topology to the current traffic pattern; however, instead of directly interconnecting ToRs like prior work, we use optical circuit switches (OCS) to rearrange the links between ToRs and fabric switches, such that traffic contributing to spine congestion ends up bypassing the spines. Our key observation is that the pod, which is a group of racks of fixed size with a transitive property of 1-hop communication between any two ToR switches, is not a natural or effective abstraction to capture data-center network patterns.

In fact, other authors [57] have argued in a similar fashion. Ghobadi et.al showed that topologies need a high fan-out, *i.e.*, that each rack can connect directly to many other racks. In our case, we need a lower fan-out than ProjecTor, because we utilize 1-hop indirection instead of direct connections, effectively multiplying the number of destinations per uplink. This design decision is supported by another paper by Roy et.al. [125] about Facebook network behavior which also showed group-based communication patterns; we support these patterns as CRISS-CROSS is able to adjust group membership for the 1-hop communication, and in fact even allows partial membership.

To motivate the design of CRISS-CROSS, we first take a closer look at the default Clos topologies. Figure 4.1 shows the classical way of drawing folded Clos topologies. In this case, it is a 1:3

oversubscribed (2 fabric uplinks vs. 6 fabric down links) topology with three layers and a switch radix of 8 for all switches. In total, this topology serves 48 racks, 8 per pod. In this presentation format, the fabric switches are grouped with the ToR switches to which they are connected. One important observation however is that each of the fabric switches in a pod is connected to a different set of spine switches. We call these groups of spine switches *spine groups*. This is signified by the different colors used both for the spine switches and the fabric switches in the pods.

Figure 4.2 shows a different layout of the fabric switches. In this identical topology, we have only changed the drawing location of the fabric switches and sorted them by spine group (*i.e.*, color). In this display format, it becomes more obvious how traffic flows. For each uplink from a ToR, traffic can flow via any of the spines in that uplink's color's spine group. This means that there is one spine group per uplink from a ToR. For traffic flowing from the spine downwards, there is only one unique path to reach each of the destination ToRs.

Figure 4.3 illustrates how we extend the Clos topology with optical circuit switches in CRISS-CROSS. As above, there are four spine groups (shown in the four colors) for the four uplinks per ToR, each with its own OCS module. The OCS module provides a level of indirection between the ToRs and fabric switches. If the OCS switches are all configured in their base configuration, the topology is that of a standard folded Clos with 8 pods, the same as in Figure 4.1 and Figure 4.2. However, it is now possible to connect a ToR to any of the fabric switches by interchanging the links within a spine group, creating partial group membership.

CRISS-CROSS effectively dissolves the notion of pods, and specifically breaks the transitive 1-hop communication property in favor of providing higher bandwidth between selected pairs of racks, while maintaining all other properties of the topology.

CRISS-CROSS uses an asynchronous control plane algorithm that (1) monitors the congestion level of the links between fabric switches and spines, (2) identifies a pair of such links to swap for reducing spine traffic, and (3) updates the routes in the network accordingly without dropping packets while minimizing or even avoiding out-of-order delivery by offloading affected links before executing any changes.

We evaluate CRISS-CROSS on a simulated, 3-layer topology using multiple traffic pattern types. Our results show that CRISS-CROSS improves the average flow completion time of group communication patterns by more than 5.5×, and the 99^{th} percentile flow completion by more than 6.3×. In fact, CRISS-CROSS even improves the flow completion time of a purely random point-to-point traffic pattern by 2.2× on average and 3× at the 99th percentile.

The closest related work is Larry [24], which also proposes a reconfigurable topology that maintains Clos modularity and hierarchy; however, that work does not address the problem of spine oversubscription and is focused on the oversubscription of fabric links.

The rest of this chapter is organized as follows. §4.2 provides the necessary background both for Clos topologies as well as optical circuit switches. §4.3 lists the requirements that a data center topology should fulfill. The topology of CRISS-CROSS is described in §4.4, including the algorithm used to replace links, routing, and connectivity guarantees. Further details of a potential implementation are described in §4.5. The performance of CRISS-CROSS is evaluated in §4.6. We close the chapter with a discussion of related work in §4.7, a brief discussion of future research in §4.8 and conclude in §4.9.

4.2 Background

4.2.1 Clos Topologies

Clos topologies are hierarchically structured network topologies. For the purposes of this paper, we will only discuss three layer, folded-Clos, and use the following naming convention for the three layers of the topology: the *spine* layer, made up from spine switches, the *fabric* layer, made up from fabric switches, and the *edge* layer, made up from top-of-rack (ToR) switches.

Figure 4.3 illustrates a 3-layer folded-Clos when the OCS module is in its baseline (passthrough configuration).

A *pod* is defined as a set of ToRs that are connected to the same fabric switches. This property is transitive within Clos topologies: if ToRs A and B are connected via a fabric switch, and ToRs B and C the same, then A and C are also connected via a fabric switch.

Communication between pods requires two additional hops between fabric switches via a spine switch. The topology is also organized to form transitive groups: a set of fabric switches are identically connected to a set of spine switches to form a *spine group*. Figure 4.3 illustrates the 4 spine groups, using different colors, of a deployment with 1:3 oversubscription.

This representation of a Clos topology is different than most in the literature. Typically, the focus is on the pods, and therefore the ToRs in each pod are shown grouped with their respective fabric switches, and shows the availability of bisection bandwidth in the topology. Our representation makes clear which spines are utilized by each of the fabric switches to provide connectivity between all ToRs, and which spine is used by which uplink.

Number of pods	radix
Size of pod (S_{pod})	R_{fab_down}
Fabric switches per pod	R_{tor_up}
Number of spine groups	R_{tor_up}
Spines per group	R_{fab_up}
Fabric switches per group	radix
N_{spines}	$R_{fab\_up} \times R_{tor\_up}$
N_{fabric}	N(spine groups) × radix
N_{ToR}	radix × R_{fab_down}

Table 4.1 – Sizing formulas for oversubscribed Clos architectures. R represents the number of ports of a switch; N the number of switches.

4.2.2 Oversubscription and Scaling

Non-oversubscribed Clos topologies are rearrangeably non-blocking, *i.e.*, they deliver full bisection bandwidth between all ToRs. However, because of economic pressures and the need to increase the scalability of the network, deployed topologies typically have some level of *oversubscription*, leading to reduced bandwidth between certain parts of the network. Within a 3-layer Clos topology, there can be oversubscription between any two layers, defined as the ratio between uplink and downlink ports.

For example, Facebook uses an oversubscription ratio of 1:4 (instead of the more standard 1:3) at this level [7], motivated both by the need to reduce the cost of the network, and increase the size, measured in the number of racks, of the data-center cluster. They preserve a large number of spare ports to be able to vary this oversubscription ratio, at the cost of wasting ports as long as there is oversubscription.

Oversubscription also typically occurs at the ToR, allowing more servers to be connected to a single ToR. In practice, this oversubscription also oftentimes involves different link speeds.

The result of typical oversubscription is that bandwidth is always non-blocking within a rack, typically plentiful and often non-blocking within a pod, but restricted across pods, based on the fabric oversubscription ratio. Table 4.1 provides the key formulas that define oversubscribed Clos topologies. Table 4.2 uses these formulas to give an overview over all possible configurations when all switches have a radix (*i.e.*, number of ports per switch) of 16 (which we will use for our evaluations). A radix of 16 also implies that the sum of down and up links is equal to 16: $R_{down} + R_{up} = 16$.

We observe from the formulas as well as Table 4.2 that: (1) the size of a pod, measured in the number of ToRs, is defined by the number of downlinks from fabric switches; (2) the number

$R_{fab\_up} : R_{fab\_down}$	R_{tor_up}	N_{tor}	N_{fabric}	N_{spines}	#Links$_{total}$	#Links$_{fab \leftrightarrow ToR}$	BW_{Bisect} / BW_{Rack}
1:15	1	240	16	1	256	240	0.42%
1:15	2	240	32	2	512	480	3.33%
1:15	4	240	64	4	1024	960	3.33%
1:15	8	240	128	8	2048	1920	3.33%
2:14	1	224	16	2	256	224	7.14%
2:14	2	224	32	4	512	448	7.14%
2:14	4	224	64	8	1024	896	7.14%
2:14	8	224	128	16	2048	1792	7.14%
4:12	1	192	16	4	256	192	16.67%
4:12	2	192	32	8	512	384	16.67%
4:12	4	192	64	16	1024	768	16.67%
4:12	8	192	128	32	2048	1536	16.67%
8:8	1	128	16	8	256	128	50.00%
8:8	2	128	32	16	512	256	50.00%
8:8	4	128	64	32	1024	512	50.00%
8:8	8	128	128	64	2048	1024	50.00%

Table 4.2 – Set of possible deployment configurations for radix=16

of fabric switches in a pod in turn is exactly the number of uplinks from each ToR; (3) the number of *spine groups* is equivalent to the number of uplinks from the ToR; (4) the number of spines in each group, in turn, is defined by the number of uplinks from each fabric switch. From this, we can derive the total number of racks in the cluster.

Table 4.2 additionally shows the tradeoff in terms of number of links at the different layers in the network, which impacts the overall bisection bandwidth of the system.

4.2.3 Optical Circuit Switches

Optical circuit switches (OCS) have made recent advances in scalability as well as switching speeds, making them more attractive for the data center. Common circuit switches in the market typically use 3D-MEMS. There is a fundamental tradeoff between switching speeds, insertion loss, and port count in MEMS-based switches. Large port counts reduce the response speed of switches because they require mirrors with larger apertures and tilting ranges [99]. For large port counts, this limits maximal switching speeds to the order of 10-100ms [22, 119] Newer advances, including 2D-based MEMS wavelength-selective switches (WSS) [51] have reduced the switching times to O(10µs) for relatively small port counts. The most recent results show sub 1µs [128] reconfigurations, while increasing port counts to the range of competitiveness ($> 50 \times 50$) with 3D-MEMS based products. 128×128 radix silicon photonic single chip switch fabrics are viewed as feasible [111].

The largest circuit switches with µs scale latencies are available with 2048 ports [101]. For the combination of these latencies and port counts, the authors had to make tradeoffs in

the generality of their reconfigurations, reducing the number of possible configurations, *i.e.*, which input ports can be connected to which output ports.

Figure 4.3 suggests that our design relies on OCS as a level of indirection, and that the OCS modules should scale with the number of fabric switches per spine group and the number of fabric downlinks, *i.e.*, $2 \times \text{radix} \times R_{fab\_down}$.

4.3 Problem statement

Our approach is based on the observation that: (1) existing data-center networks rely on fabric oversubscription of folded Clos networks to reduce costs and increase cluster sizes; (2) data-center traffic patterns, *e.g.*, as presented by Facebook, show temporal locality, in particular when considering rack-to-rack traffic behavior [125]; and (3) the heavy degree of virtualization and automation in large-scale cloud data centers, driven by different concerns such as high-availability, power management, and hardware diversity leads to inherently unpredictable network traffic patterns [12, 14, 80].

The overall performance of the network therefore varies based on the changes of the rack-to-rack network patterns: if the traffic is mostly intra-pod, the spine links will be largely underutilized and oversubscription is a non-issue. Conversely, if the traffic goes mostly between pods, the spine links will become bottlenecks. This is true in particular when the traffic patterns are randomly distributed across rack pairs, or in the presence of group communication patterns between groups of servers that are randomly allocated to racks.

In such topologies, the limiting resource is the set of spine links that carry traffic across pods. This work addresses the following problem statement:

Is it possible to use reconfiguration in an oversubscribed network to increase available bandwidth for flows in order to reduce their flow completion times, while keeping the hierarchy, supporting unmodified IP-based protocols, and commodity-parts-based hardware?

Connectivity and Path Diversity: The solution needs to guarantee connectivity between all racks at all times, both during failures as well as during reconfigurations. For this, the topology needs to provide a certain path diversity.

Scalable Routing: Scalable routing refers to two things: routing tables need to be limited in size, and the calculation of routes as well as change sets needs to have a small algorithmic complexity.

Deployability: The proposed architecture needs to be deployable in a data center with limited additional work compared to current Clos topologies. It should use standard equipment for as many components as possible. Specifically, the cabling complexity and installation of switching hardware should not be a cause of cost increases or deployment times.

Application transparency: Reconfigurations of the network should be totally transparent to applications and to existing, established flows. Ideally, reconfigurations should not cause packet drops or out-of-order delivery.

4.4 Design

Figure 4.3 shows the high-level design of the physical topology and of the control plane of CRISS-CROSS. Starting with a standard folded-Clos topology, a programmable OCS module is inserted for each spine group. We discuss the resulting topology (§4.4.1), followed by the high-level algorithm behind CRISS-CROSS (§4.4.2), the routing implications (§4.4.3), and detail why the algorithm guarantees connectivity (§4.4.4).

CRISS-CROSS satisfies the requirements listed in §4.3. In normal operation, there are always at least as many paths between two racks as there are uplinks from each. Both failures and reconfigurations therefore will leave other paths to preserve connectivity. There is one OCS module per spine group (and therefore per color), guaranteeing that reconfigurability can never create an unconnected graph.

Routing needs at most N_{ToR} entries in the routing tables for each switch; the calculation of routes is possible in a centralized controller with the same complexity as Clos topologies.

The architecture is also deployable physically as fabric switches are already co-located in centralized locations, and the OCS are per fabric group [7]. Beyond the OCS, CRISS-CROSS uses only standard components. It requires minimal adjustments to the control plane for routing, and utilize standard draining techniques [131] to limit impact to existing routes.

4.4.1 CRISS-CROSS Topology

The use of OCS turns CRISS-CROSS into a superset of all folded-Clos topology. In fact, the baseline permutation of the OCS is simply one specific folded-Clos topology. As such, CRISS-CROSS preserves the hierarchical structure, port and link counts as well as most of the benefits of oversubscribed folded-Clos topologies identified in §4.2.

The ability to permute links could allow a topology that dynamically swaps racks between two pods as to optimize intra-pod traffic. Unfortunately, such a swap would require $O(radix)$ links to be changed dynamically, which would require a complex, difficult to analyze transition phase, especially to meet the requirement of continuous network connectivity.

Instead, CRISS-CROSS dissolves the pod structure by undoing the transitive property of pod membership by swapping a single pair of links at a time. Prior work has shown that there is significant locality in flows between racks [125]. CRISS-CROSS aims to capture this temporal locality and apply reconfigurations so that bandwidth between subsets of ToRs is increased.

Figure 4.3 shows that the spine group, and its associated fabric switches constitutes a *reconfiguration cluster*, with one OCS between each reconfiguration cluster and its attached ToRs. This circuit switch allows us to connect any ToR to any fabric switch within the reconfiguration cluster, *i.e.*, connect any of the links of one color to any of the fabric switches of the same color of Figure 4.3.

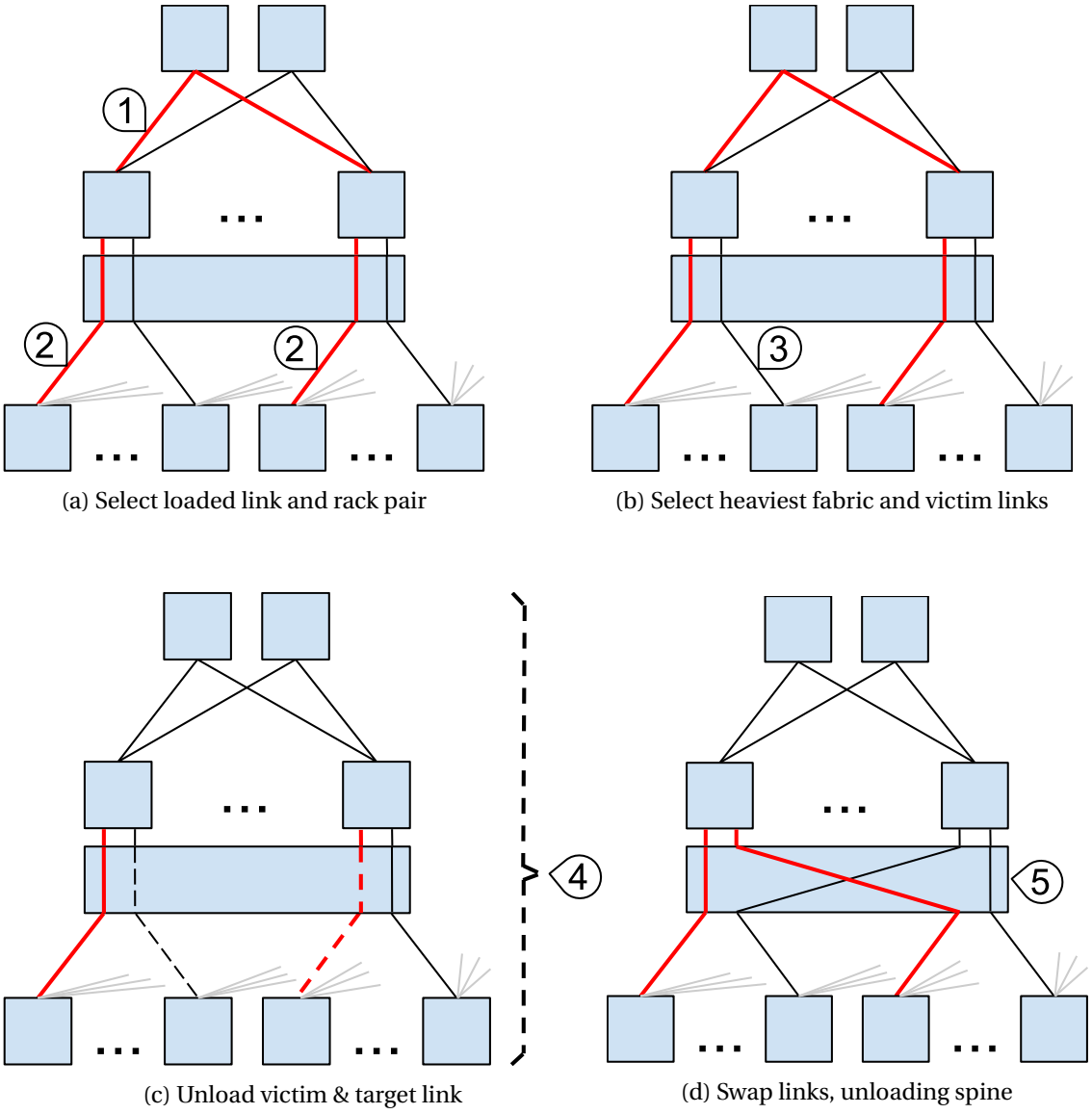
By labelling links with a static color (*i.e.*, with a given reconfiguration cluster), CRISS-CROSS ensures that connectedness is preserved across all permutations: assuming that the baseline folded-Clos provides connectivity between any two ToR, without having to route via a third ToR, that property holds as well for CRISS-CROSS (see §4.4.4).

Table 4.3 shows the required total port count for the OCS module of each reconfiguration cluster, which must support at least all possible bipartite permutations between the ports to the fabric switches and ports to the ToR. The number of links is radix × R_{fab_down}. With contemporary single-ASIC switches typically having 32, 64, or even 128 ports, this leads to port counts in low thousands, with recent developments in OCS technologies moving CRISS-CROSS into the realm of feasibility (see §4.2.3).

CRISS-CROSS requires that the topology consists of at least two reconfiguration clusters to ensure the continuity of traffic during link reconfiguration. In fact, concurrent link reconfigurations are possible as long as at least one color can temporarily ensure that every destination has a route at all times, including when links are down.

$R_{fab\_up} : R_{fab\_down}$ / Radix	16	32	64	128
1:1	256	1024	4096	16384
1:3	384	1536	6144	24576
1:(Radix-1)	480	1984	8064	32512

Table 4.3 – Number of ports required in an optical circuit switch as a function of the fabric oversubscription and the switch radix.



(a) Select loaded link and rack pair

(b) Select heaviest fabric and victim links

(c) Unload victim & target link

(d) Swap links, unloading spine

Figure 4.4 – Algorithm for reconfiguration of links

Physical deployability is also simple: for example, Facebook already organizes their data centers around spine groups [7], with all fabric switches in the same location. We extend on this in §4.4.5.

Routing remains as straightforward as with a standard folded-Clos topology, as the hierarchical nature of the network is preserved. Even shortest path routing as deployed in current data centers is possible, both with standard protocols such as iBGP or OSPF with the associated typical deployment options [90]. The subtleties required to minimize impact to traffic during reconfiguration are described in §4.4.2; the routing itself is described in §4.4.3.

Finally, CRISS-CROSS has a useful fallback property which allows for a conservative deployment model: if desired, the OCS modules can be configured to revert CRISS-CROSS back to a standard, static, folded-Clos topology. As an alternative, the decision to deploy CRISS-CROSS can be done in software, at any point in time after installation.

4.4.2 High-level Algorithm

We now describe the high-level control plane algorithm of CRISS-CROSS. As shown in Figure 4.3, we assume a centralized algorithm has access to statistics, at the granularity of flows, for all spine links. The flow granularity must be of the rack-pair, *i.e.*, identify the source and destination ToR. The algorithm uses that information to (i) update routes to drain links, (ii) bring these links down, (iii) reconfigure the OCS, and (iv) bring the links back up in the new topology and (v) update the routes to use them.

Figure 4.4 shows the key steps of the algorithm, whose main goal is to unload spine links by connecting the most active pairs of racks directly to one fabric switch. The algorithm operates within a reconfiguration cluster and has four main steps illustrated by Figure 4.4a–4.4d:

a: It identifies the hottest spine link of the reconfiguration cluster (1), then the rack-pair that contributes the most to the traffic of that spine. This unambiguously identifies the two target links and their ports on the ToR and the on the fabric switch (2) (see §4.5.1 for details).

b: By definition, the two target links are attached to two different fabric switches of the same reconfiguration cluster. The swap must identify a victim link that is attached to either of the target fabric switches but not connected to either of the target ToRs. We select v as the least-loaded link that matches the requirements (3), see §4.5.2.

c: The algorithm updates the routing tables to unload two links: the victim link v, and the target link that is connected to the other fabric switch than v. Once the traffic is unloaded, the links can be disconnected (4).

d: The two links are swapped via an OCS reconfiguration. Routes are then updated (5), see §4.5.3.

4.4.3 Routing

Routing in CRISS-CROSS is very similar to the shortest path routing in a Clos topology. We propose two alternate routing schemes — shortest path first (SPF) and hierarchical routing.

Theses schemes are implementable because they allow aggregation of switches, and are computable in limited time due to the hierarchy of the topology.

CRISS-CROSS follows the observation from Jupiter Rising [131] that there are three different types of switches in the topology, and each has specific route calculations. For the hierarchical routing scheme, we make the following observations:

- The *ToR* switches are responsible for a specific subnet. Their ports are either down or up ports; any traffic coming in from below is sent to any port upwards, while the downwards traffic is directed to the appropriate server. The choice for which up-port to take can be decided with varying mechanisms, but equal-cost multi path (ECMP) is sufficient for simple load balancing.
- The *fabric* switches have route entries for all subnets of all directly-connected ToR switches. All other traffic is sent to any of the spine ports, with ECMP a valid load-balancing strategy at this level as well.
- Lastly, the *spine* switches. As in standard Clos topologies, the spine identifies the unique route to a destination rack. Unlike a standard Clos, the routes cannot be aggregated at the pod level.

The calculation for the shortest paths needs one additional restriction: For the *ToR* switches, if the source and destination share a fabric switch, then only those uplinks to the shared fabric switches should be added, as all other routes are via the spine and therefore longer. For both routing schemes, all routes have a length of 3 or 5.

We examine two different situations: the state of routes in the stable states between configurations, and the changes in routing during the execution of the reconfiguration algorithm.

Steady state: For the hierarchical case, the OCS reconfiguration does not permanently impact the ToR as (1) the connected subnet does not change and the ToR must only differentiate traffic destined for its own rack; (2) all other subnets remain reachable over all uplinks since the link changes operate exclusively within a spine group. The downlink routes of the two fabric switches involved in the reconfiguration change as they must now direct traffic to a new subnet and no longer direct traffic to another one. All spines of the reconfiguration cluster must update the routes of the two racks involves in the swap, *i.e.*, changing the unique downlink for two fabric switches. This gives a complexity of $2 \times N_{spines}$ updates. The additional rule for the shortest path routing implies that only the ToR routing tables are impacted. The impact is necessarily limited to ToRs on the affected fabric switches. More specifically, it is

limited to the routes of the two swapped ToRs, and the routes with a destination of either of the swapped ToRs. This means that the number of updates is bounded by $2 \times R_{tor\_up} * R_{fab\_down}$.

Transient routing changes: To prevent packet loss, links need to be unloaded before they can be reconfigured. As a result, transient route changes must be made during the reconfiguration. This affects *all* ToR of the network (but only the ToRs) as each ToR can be reached only via one link from the spines in a single spine group (*i.e.*, color). However, this is also the only link through which it can be reached from the fabric switches in that color group, and their spines.

Recall that the swap involves two links of the same spine group; the changes must ensure that that specific spine group is temporarily not used to communicate to/from the two affected ToR switches. Specifically, (1) the two involved ToRs must update the ECMP group to temporarily exclude that uplink; and (2) all other ToRs must insert two specific uplink routes for the destination ToR. These specific uplink routes can be load-balanced via an ECMP group that excludes the spine group. All these transient operations can be undone in the final step of the algorithm. This procedure is the same for both routing algorithms. In fact, SPF can avoid any changes if shorter paths already exist. On the other hand, when a single shortest path is disconnected, then all the other longer paths need to be readded.

4.4.4 Connectivity between ToRs

The prior section gives a good intuition on why the graph is always connected, even during a reconfiguration. We will now describe this in more detail: all the fabric switches in a spine group are connected via the spine switches. Every one of the spine switches can forward traffic between any of the fabric switches. Therefore, we have path redundancy already within the spine groups two top layers, as long as there is more than one uplink available per fabric switch, and they do not have exclusive failures.

The algorithm described above guarantees that a ToR is connected to at $N_{spinegroups}$ different fabric switches, each of which can reach all other fabric switches in the same spine group. Reconfiguration is only between links in the same spine group, and so there are available paths between all ToRs via the other $N_{spinegroups} - 1$ spine groups.

Additionally, the fact that the links are only swapped within the spine group means that the fabric switches reachable via this uplink remain the same. Only the path length for two fabric switches changes.

From the above description, it is also clear that CRISS-CROSS satisfies the path diversity requirement. Some additional care needs to be taken when there are large numbers of failures.

As long as there is more than one uplink available and the fabric switches have connectivity to at least one shared spine switch, the reachability between ToRs is guaranteed.

4.4.5 Deployability and Physical Layout

In this section, we will explain how CRISS-CROSS satisfies the requirement for deployability, particularly as it related to the physical layout of the data center. Little is known about the physical deployment of most data centers, but Facebook has published some information in [7]. They organize their data centers around the spine groups, with all fabric switches of the same spine group in the one location, *i.e.*, all fabric switches with the same "color" are in the same place.

As the circuit switches in our design need to be between all the racks and the fabric switches in one spine group (color), this layout is directly adaptable. From a deployability standpoint, we can simply add an OCS at this location. In fact, the only change required would likely be to replace a non-programmable (or statically programmed) optical patch panel with a programmable OCS.

4.5 Implementation

This section describes the concrete implementation for each of the steps in the algorithm in §4.4.2, *i.e.*, the candidate selection algorithm, link changeover, and route reestablishment. The implementation assumes a centralized SDN controller to coordinate these steps in a real data center. We describe both the implementation in our simulation environment as well as potential solutions in a real data center.

4.5.1 Candidate Selection

The candidate selection in the simulation environment tries to emulate the approximate nature of candidate selection in the real world. It first identifies the heaviest used spine link, and then the rack responsible for most of the traffic on this link.

The description in the algorithm 4.4.2 simplifies the criteria to a pair of racks. The actual goal of the candidate selection algorithm is to find the rack / fabric switch pair with the highest amount of traffic going via the spine, *i.e.*, the bottleneck of the topology. The logic here is simple: if a rack communicates a lot via a remote rack, then it communicates a lot with the attached racks and should be moved closer to them. Ideally, the algorithm chooses the pair with the highest number of flows, which we implement in the simulation. For this, each rack

has to keep a count over how many flows are active from other racks via its uplinks (local traffic does not need to be further optimized). The algorithm then choses the rack out of all with the highest number of flows via a single fabric switch.

We have however also tested reducing the overhead of this task to the simpler issue of identifying the busiest pair on a single spine link. We present the results for the first version, but the alternative has competitive results as well. There are two general cases: zero or one links are at capacity, or multiple links are at capacity. In the first case, the link with the highest congestion is chosen as the critical link. For all flows across this link, the one with the highest throughput currently is identified. In our simulation, this is a parameter explicitly tracked. In a data center, this might be achieved via techniques such as flow sampling [107], or techniques described by Harrison *et al.* [68].

Candidate selection can run continuously and in parallel to the normal operation of a network.

4.5.2 Victim Selection

The goal of the victim selection is to minimize the new load put on the spines. Switching a link from one fabric switch to another means that traffic which originally was local over this link now has to go via a spine. On the other hand, all traffic to the ToRs on the new fabric switch becomes local. So, to make perfect decisions, one needs to know exactly the amount of bandwidth used to local and to remote racks, for both the source and the destination. We implement a heuristic that considers all links on both fabric switches involved, identifies which victim ToR has the least local traffic on these links for both, including the candidate tor, and then choses that victim ToR (and its link) for replacement. Another approach would be to ignore the potential downside of a swap and simply choose a good offloading candidate. While we decided to simulate the more optimal algorithm, preliminary testing revealed only minor differences in performance.

4.5.3 Link Change Procedure

Once the algorithm in the prior section has identified the set of links to change, they need to be swapped without impacting running connections. We separate the description of this link change procedure into three phases: preparation and unloading, reconfiguration, and route reestablishment.

For the preparation of the link, we follow the routing changes described in §4.4.3. Unsetting the routes on all involved switches can be done in a simple round trip time, *i.e.*, less than

a millisecond, and the offloading itself will depend on the buffer utilization, *e.g.*, for small buffers, it would also be less than <1ms. The reconfiguration even of slower TCAMs takes less than 1ms [52]; the total time for unloading links is therefore less than 2ms.

Another solution would be to have a completely source-routed network [76, 79, 110, 123]. In that case, switches could just bounce packets back to the sender if a link goes down. This would also allow a ”lazy” update mechanism, where this sender would then automatically discover new links when an old link goes down and is used right after. This would further reduce the number of updates of the routing table, as typically only small subsets of servers communicate with each other [125].

4.5.4 Link-level Detection / Configuration

The latency to bring up a physical link is highly dependent on technology. This is also a non-optimized operation in most switches, and would require some engineering effort for an improvement to re-establish Ethernet links (*e.g.*, via an optimized auto-negotiation) and to notify the control plane. However, this is likely dominated by the reconfiguration of the physical link via the OCS, *i.e.*, vary between 10µs (Helios) and 100ms (largest optical switches commercially available).

Route reestablishment: This is the last step of §4.4.3 when routes need to be re-added. To prevent misrouted traffic, the spines and the two updates to the fabric switches need to happen first. As no traffic for the destination ToRs can reach these layers without new entries at the ToR, this can happen in parallel to the OCS reconfiguration and is therefore likely dominated by its reconfiguration time. Once links have been established, the two destinations routes are added back to all uplinks.

4.6 Evaluation

In this evaluation, we show that CRISS-CROSS reaches its primary goal of introducing reconfigurability and that it reduces the flow completion times and sustainable load. For all experiments, we use the simulation environment described in §4.6.1. In §4.6.2, the reader will find the experimental methodology.

First, we examine the temporal behavior of the number of flows on the spine as well as reconfiguration rates in §4.6.3. After this introduction to the dynamic behavior of CRISS-CROSS, we look at the effect of different reconfiguration rates and the difference in routing algorithms in §4.6.4. Then, we show the increased sustainable load for a given service-level

objective (SLO) in §4.6.5. In §4.6.6, we describe the improvement of flow completion times at different quantiles, and close the evaluation with a sensitivity analysis of CRISS-CROSS in respect to the number of uplinks and oversubscription in §4.6.7.

4.6.1 Simulation Environment

We evaluate CRISS-CROSS with a flow-level simulator written for this purpose. The simulator uses an event driven model of flow arrivals, flow completions, and timed reconfiguration events and is extendable. The assumption behind CRISS-CROSS is that the necessary locality in flows needs rack-level aggregation. Therefore, this is the chosen granularity for flows. Flows in the network observe max-min fairness to simulate the behavior of TCP flows.

We model a centralized SDN controller that creates routes either via a shortest-path algorithm (SPF) or via the hierarchical routing scheme described in §4.4.3. The control plane has global knowledge about all flows and executes link-change decision at zero cost.

4.6.2 Experimental Methodology

The experiments are done on a subset of the topologies listed in Table 4.2, which we specify at the beginning of each section. All experiments use a radix of 16 with a 10 Gbps link speed as the size of the simulation is limited by runtimes. In §4.6.3 and §4.6.4, we use a $4:12$ oversubscription with $R_{tor\_up} = 4$, *i.e.*, four uplinks from the ToR corresponding to four spine groups. This oversubscription ratio is close to the 1:3 oversubscription used in practice. We do not define an oversubscription ratio for the ToR, as we investigate and simulate only rack-level flows.

In §4.6.7, we perform a sensitivity analysis for four topologies defined by an oversubscription of either $2:14$ or $4:12$, and R_{tor_up} of either 4 or 8. Note that $2:14$ is the highest level of oversubscription that still provides path redundancy within a spine group. As a baseline, we also simulate a rearrangeable non-blocking topology, where all uplinks of a ToR are connected to a single non-blocking switch.[1]

We define load l as a fraction of the total bandwidth BW_{tot} of a non-blocking topology with the same number of racks and rack uplinks. Using this definition and assuming a uniform distribution of sources and destinations, we calculate the theoretical maximum load l_{max} at which the spine of a given topology would be fully saturated. To calculate l_{max}, we first look at how much bandwidth is available on the spine. Then, we determine how much

[1] It is important to note that this does not imply that there is never any congestion. Congestion is still possible on the uplinks from each individual rack, and in fact, expected.

traffic can be sustained in total for the given traffic distribution. For the total bandwidth of a non-oversubscribed Clos topology, we have

$$BW_{tot} = N_{ToR} \times R_{tor\_up}$$

The fraction of traffic via the spine is $\frac{R-1}{R}$, as there are R pods in a Clos topology, and only $\frac{1}{R}$ of all traffic is expected to be local for a uniform distribution. The total bandwidth via the spine is

$$BW_{spine} = N_{spines} \times R = R_{fab\_up} \times R_{tor\_up} \times R$$

This means that the maximal theoretical sustainable bandwidth is

$$BW_{max} = BW_{spine} \times \frac{R}{R-1}$$

resulting in a theoretical maximum load of

$$l_{max} = \frac{R \times R_{fab\_up}}{(R-1) \times R_{fab\_down}}$$

The evaluation uses two traffic patterns on CRISS-CROSS. Both traffic patterns utilize a Poisson process for arrival times and flow sizes. We present the results for a flow rate of 3000 flows/sec, as this number presents a reasonable trade-off in regards to simulation overhead and is in line with numbers reported in the literature for real-world arrival rates [80]. We have done exploratory studies on different flow rates, and gotten qualitatively similar results.

Random all-to-all (RATA): The first scenario is an all-to-all traffic pattern with an equal distribution for the source and destination racks. This traffic pattern has no inherent locality, and likely represents traffic in a data center in which network locality is not used for scheduling. This scenario is an example of the traffic patterns described in the literature [12, 14, 57, 80], but without introducing any skew to simulate the worst-case locality. RATA constitutes a challenge for CRISS-CROSS as there is no long-term skew.

Alternating best/worst case (ABW): The second scenario is a combination of two separate sub-patterns. The two sub-patterns are the best and the worst-case traffic patterns for a Clos topology. The best case in a Clos topology is purely pod-local traffic[2]. This means that for the best-case sub-pattern, all communication is within pods. The worst-case traffic for a Clos topology is purely non-pod-local traffic. We make a further requirement for this sub-pattern

[2] In case there is oversubscription at the rack level, purely rack-local traffic would have a higher total available bandwidth. However, our simulation is concerned only with flows at rack level or higher.

by limiting the communication groups to the size of a pod. Hence, there exists a Clos topology for which this worst-case sub-pattern is actually a best case. Within each sub-pattern, the source and destination racks follow a uniform distribution.

We then alternate the best and worst case over time to create a combined pattern. This combined pattern maximizes the number of reconfigurations needed, and therefore stresses the reconfiguration part of the design. For all experiments, we alter between the two sub-patterns every 10 seconds. ABW stresses CRISS-CROSS and forces it to optimize the topology every 10 seconds, while it is the worst-case for a standard Clos topology half of the time.

We compare three different setups in our experiments:

1. No reconfigurations, *i.e.*, a baseline folded Clos topology
2. 10, 30, 70, 100, 300, and 500 reconfigurations/second, corresponding to estimates for target reconfiguration rates, with 10 reconfigurations per second being comparable to single OCS on the market today
3. Unlimited reconfigurations

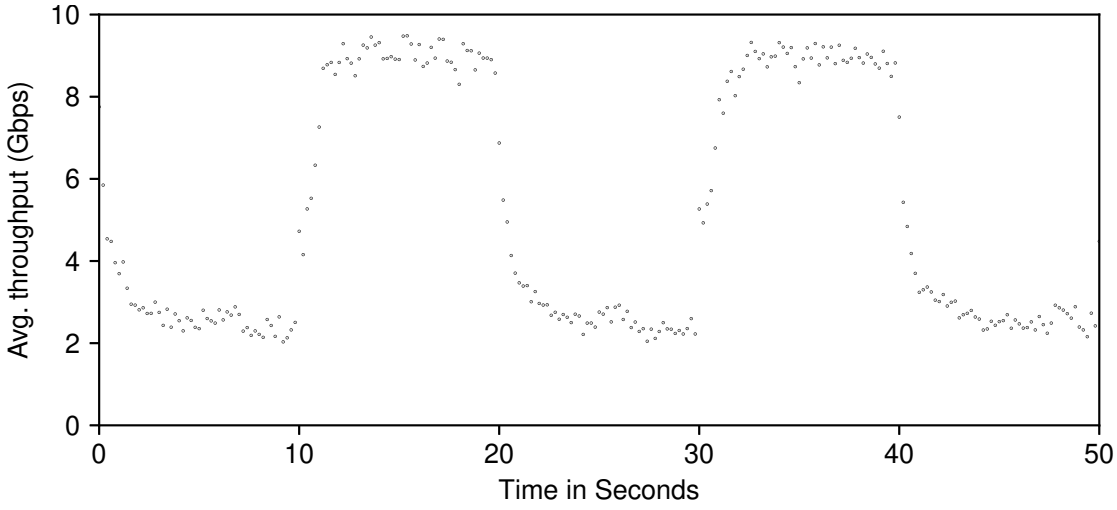
In this evaluation, we only present a small number of topology variants. However, these variants are representative of large groups of settings; a full exploration of the whole state space is computationally prohibitive.

4.6.3 Effect of Reconfiguration

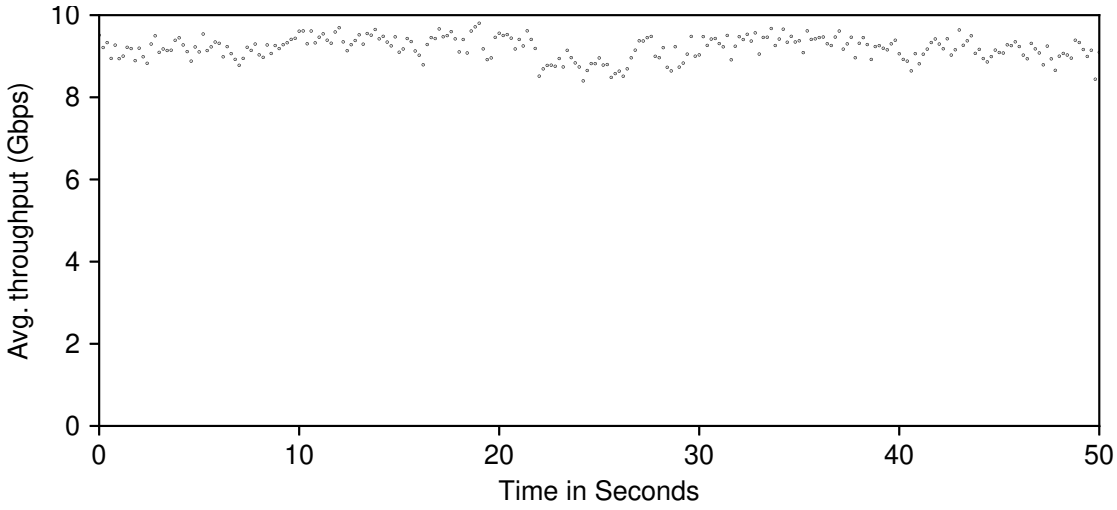
We begin our exploration of CRISS-CROSS behavior with a look at the effects on average flow throughput for the ABW scenario. All simulation runs are on a 1:3 oversubscribed topology with 4 uplinks.

Figure 4.5 shows the average flow throughput across all flows over time. In Figure 4.5a, we show the behavior of throughput without reconfiguration enabled. Figure 4.5b shows the exact same traffic, but with enabled reconfiguration. In both experiments, ABW starts with the worst-case sub-pattern, and it alternates every 10 seconds.

For the non-reconfigurable scenario, the average bandwidth available for flows is only around 2.5-3 Gbps in the worst case, which is in line with expected bandwidth for a 1:3 oversubscribed topology. At 10 seconds, the sub-pattern switches to the best case. Of note here is that not all flows finish immediately, so that there is a short period of changeover where average



(a) Flow throughput without reconfiguration / equivalent Clos topology

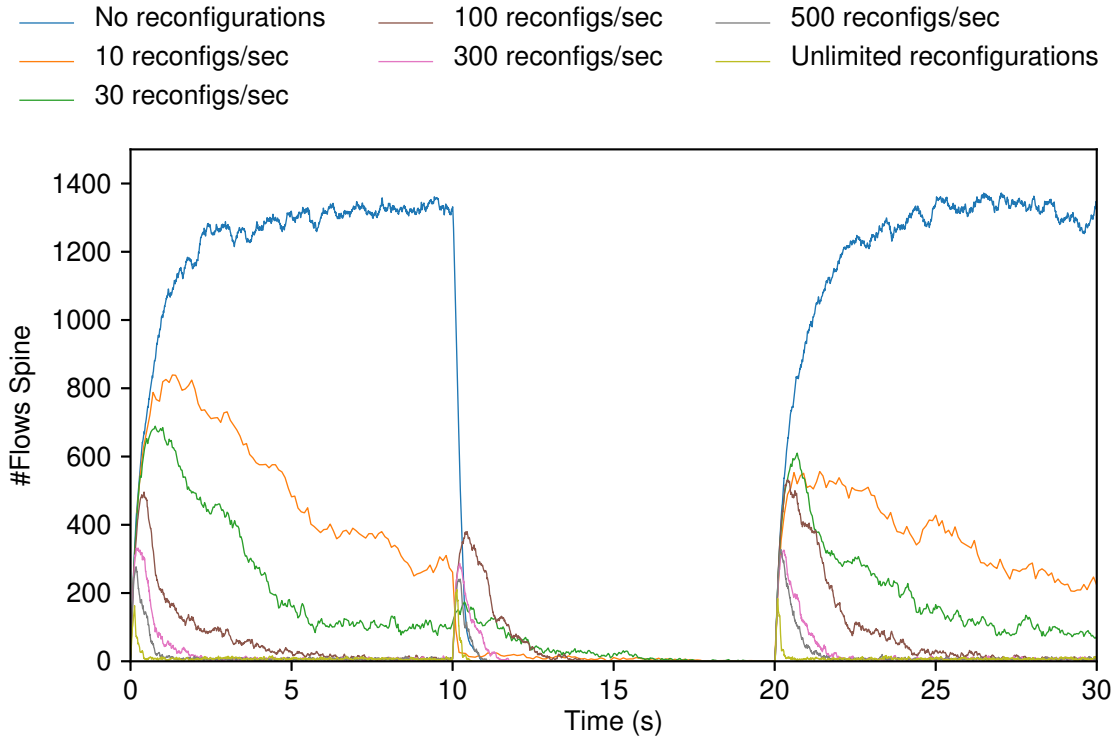


(b) Flow throughput with reconfiguration

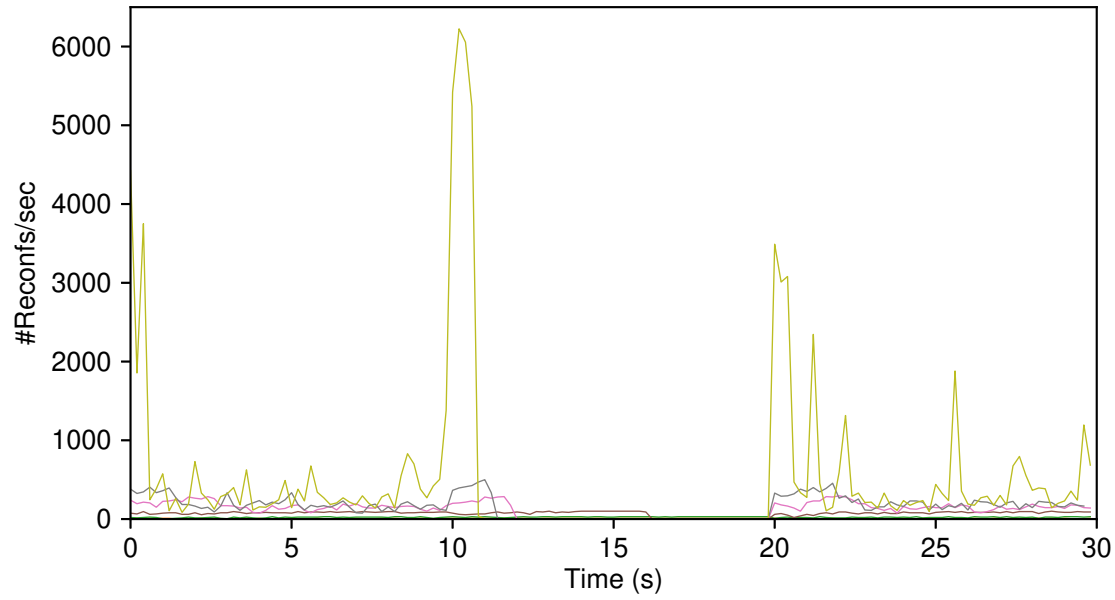
Figure 4.5 – The effect of reconfiguration on average flow throughput over all active flows for the alternating best/worst case traffic scenario. The graph at the top shows the throughput without configurations on the equivalent Clos topology, the bottom the same topology but with enabled reconfiguration.

throughput increases. Once throughput stabilizes, the average is around 9-9.5 Gbps. The average bandwidth oscillates with the state of the ABW pattern.

For the reconfigurable case, although the pattern also changes between its two extremes, the average throughput is about the same as for the best-case pattern in the prior paragraph. This validates the general assumption that CRISS-CROSS can find a good topology for both traffic matrices.

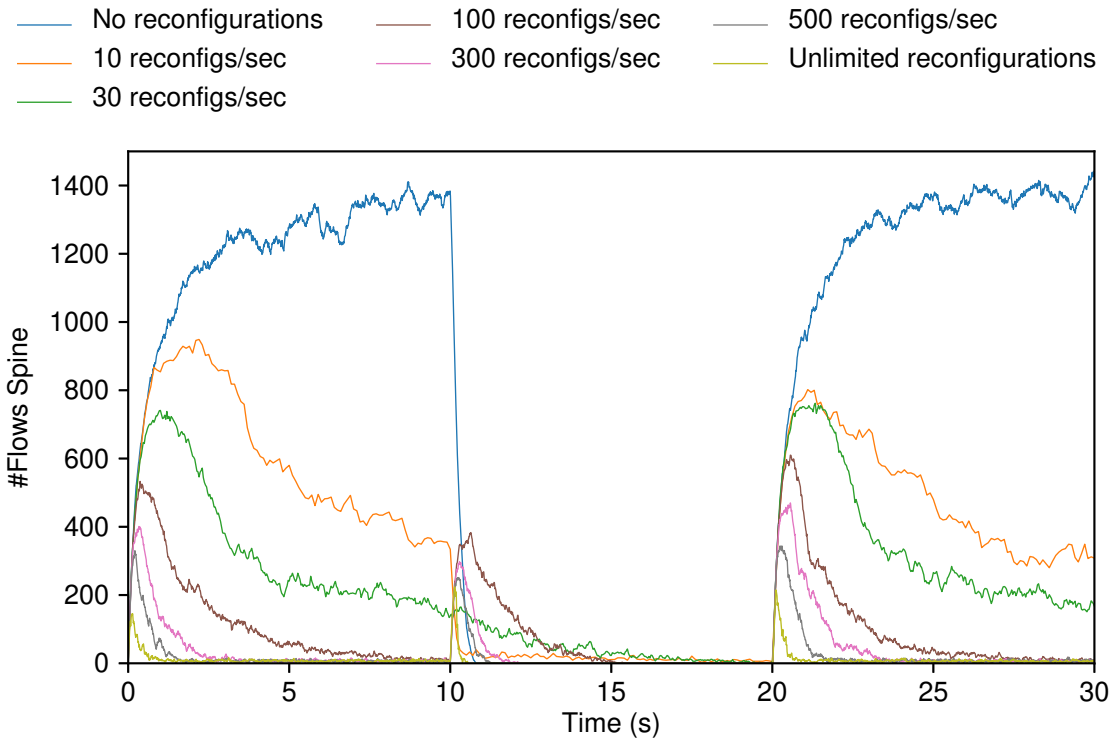


(a) Flow count

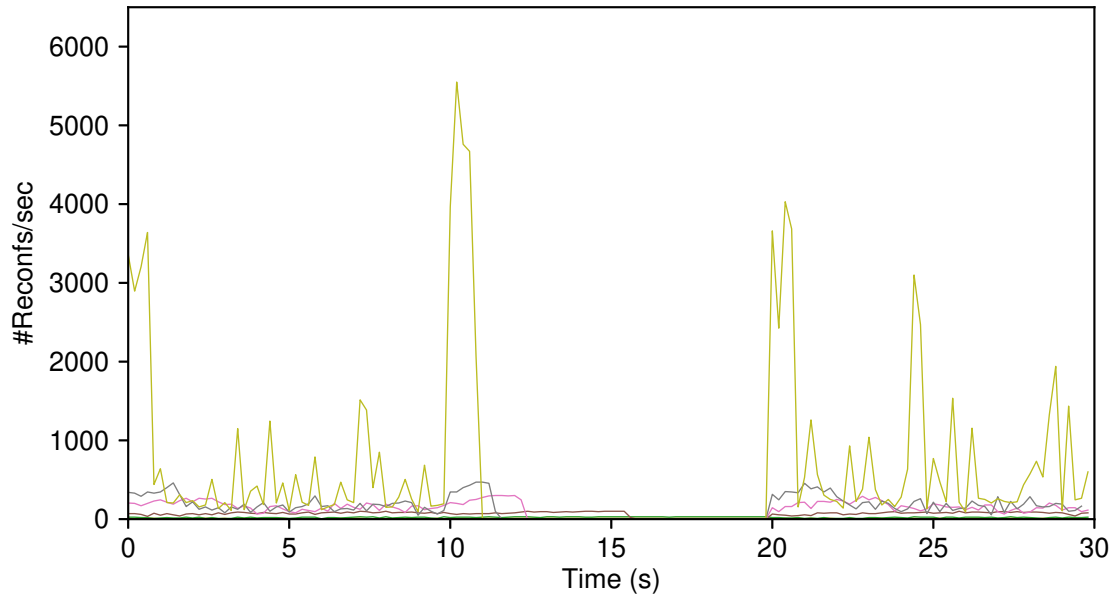


(b) Reconfiguration rate

Figure 4.6 – Temporal behavior for the shortest-path routing algorithm with the alternating best/worst-case traffic scenario, switching every 10 seconds. Here, CRISS-CROSS has a 4 : 12 oversubscription in the fabric, with $R_{tor\_up} = 4$ and 4 spine groups. The offered load is at 20%, close to the maximal sustainable load for the non-reconfigurable topology. The flow arrival rate is 3000 flows/sec.



(a) Flow count



(b) Reconfiguration rate

Figure 4.7 – Temporal behavior for the hierarchical routing algorithm with the alternating best/worst-case traffic scenario, switching every 10 seconds. Here, CRISS-CROSS has a 4 : 12 oversubscription in the fabric, with $R_{tor\_up} = 4$ and 4 spine groups. The offered load is at 20%, close to the maximal sustainable load for the non-reconfigurable topology. The flow arrival rate is 3000 flows/sec.

As a next step, we look at the effect of different reconfiguration rates and the adjustment speed of CRISS-CROSS. We continue to use the ABW scenario on the same topology, but we extend the analysis to include our two routing algorithms. The goal of the reconfiguration algorithm is to reduce the load on the spine. The typical way of measuring load is with the throughput on the affected links. However, we utilize max-min fairness for all flows, and —as there is no oversubscription at the lower layers in the simulation—, the bandwidth is only limited by ECMP collisions on the lower layers and the available bandwidth on the spine itself. This means that offloading flows from the spine only has a limited impact on utilization of the spine. So, while we are able to show a reduction in bandwidth, we find it more instructive to look at the number of flows utilizing the spine. In Figures 4.6 and 4.7, we first look at this number of flows on the spines, and then at the number of reconfigurations actually executed to achieve these numbers.

We utilize a load of 20%, and a flow rate of 3000 flows/second, first with shortest-path routing, and then with the hierarchical approach, in Figure 4.6a and Figure 4.7a respectively. For both, we show the number of flows on the spine over time for several different rates of reconfiguration, including no reconfiguration (*i.e.*, a non-reconfigurable Clos topology), and unlimited reconfigurations (which should provide the best performance).

For SPF, the behavior is as expected. The number of flows stabilizes for the non-reconfigurable topology after an initial startup phase. When the pattern changes over to the best-case, this number drops quickly to zero, before rising to the same level when the pattern switches back to the worst case. For the reconfigurable topologies, we see a short spike at the beginning of each 10s interval while CRISS-CROSS adapts the topology to the current traffic pattern. As the topology adapts to the running traffic pattern, the number of flows on the spine decreases. Of course, this adaption takes longer the fewer reconfigurations per second CRISS-CROSS has to work with. In fact, for reconfiguration rates below 100, it takes longer than the full interval of 10 seconds, while it's around 200ms for unlimited reconfigurations.

Opposite to the non-reconfigurable topology, CRISS-CROSS adapts to the worst-case. This means that when the traffic pattern changes to the "best-case", the traffic now uses the spine. Therefore, we see an initial spike in the number of flows again. However, this spike is not as large as it is for the worst-case sub-pattern, as CRISS-CROSS does not reach an optimal solution for the worst-case sub-pattern within the 10s interval. This is because not all racks are actively communicating in their groups during each interval, and so some of the connections between racks for the "best-case" pattern persist even during the "worst-case" part of the pattern. This also explains why the lower reconfiguration rates have fewer flows on their spines in this case.

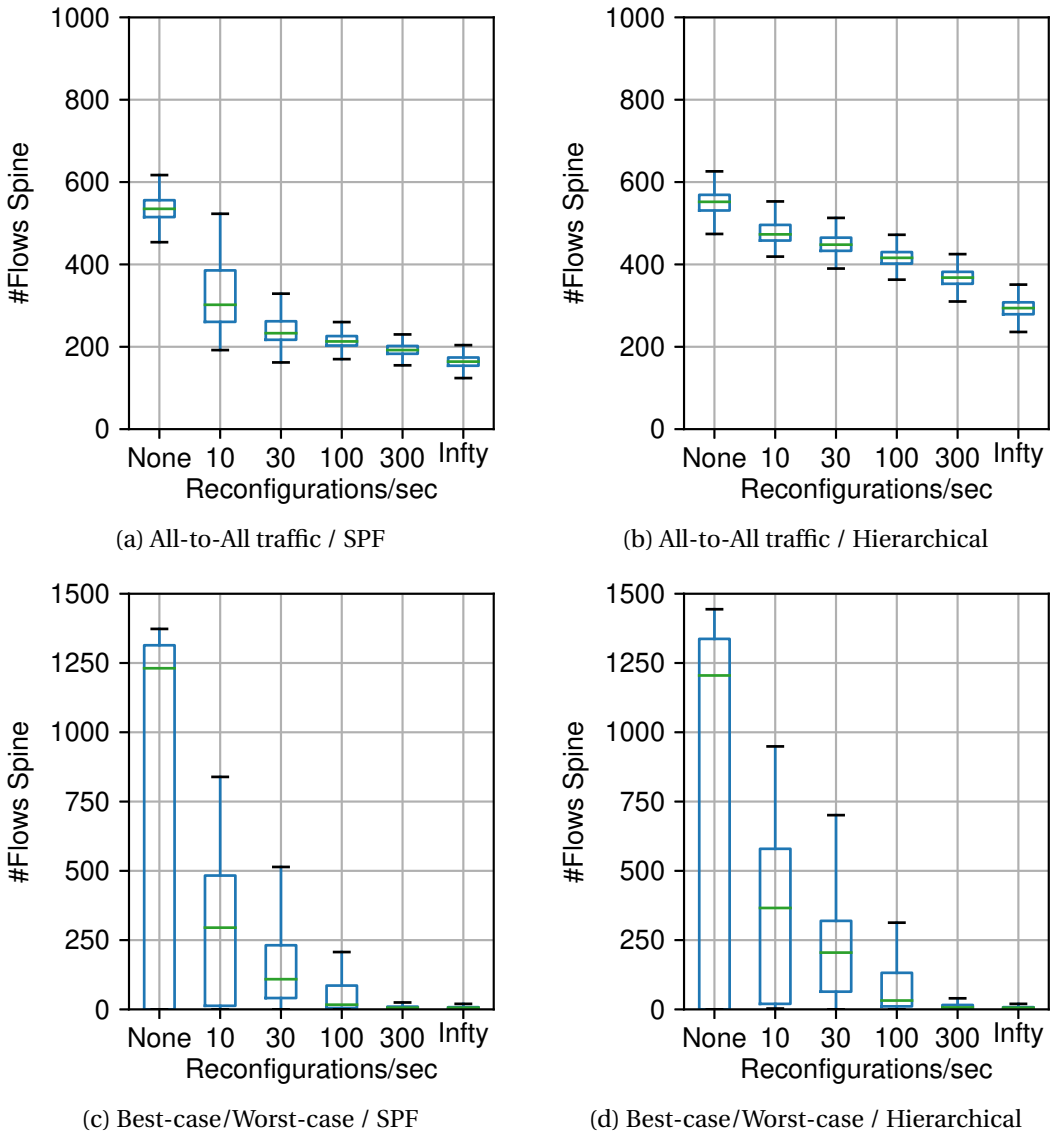
We repeat the same experiment in Figure 4.7a, but this time with hierarchical routing. The results here are very similar to the ones above. However, in the reconfigurable cases, the spikes at the beginning of a new traffic pattern are higher, and it takes longer for CRISS-CROSS to reduce them. This is because for SPF routing, as soon as a pair has one shared fabric switch, then all flows between those pairs bypass the spine. In hierarchical routing, this is only true for 1/4 of the flows.

Figure 4.6b and Figure 4.7b measure the actual reconfiguration count. When switching between two sub-patterns, there is a large spike in the unlimited case. Yet, CRISS-CROSS never stops adjusting the topology for the worst-case sub-pattern. This is for the same reason as the larger spike in the number of flows above: the heuristics cannot determine a truly optimal topology for that sub-pattern. On the other hand, when switching back to the former best-case sub-pattern, it is able to revert most of its changes from before and completely stop all reconfigurations. While this seems like an issue, traffic patterns in practice are never as black and white, and so the optimal topology is a moving target anyway. Hence, any algorithm that moves towards this optimal topology quickly enough can make significant improvements to total network performance. The number of reconfigurations for both routing algorithms is relatively constant.

4.6.4 Spine Utilization

In this section, we evaluate the utilization of the spine for a full run of the experiment. We show the results for both reconfiguration algorithms and both RATA and ABW in Figure 4.8. The box graphs show the quartiles as well as the minimum and maximum values for occupancy of the spine, and a subset of reconfiguration rates that show the trends. The topology is the same as in §4.6.3, *i.e.*, 1:3 oversubscription and four uplinks per ToR.

Figure 4.8a and Figure 4.8b show that there is limited variance in the number of flows utilizing the spine over time for RATA. However, even in this random source and destination pair process, there is sufficient locality to allow reconfiguration to reduce the number of flows between 2.9× for 500 reconfigurations per second, and 3.2× for unlimited amounts of reconfiguration for SPF routing. The results for hierarchical routing are not as good, but CRISS-CROSS still achieves nearly a 50% reduction in the number of flows on the spine with unlimited reconfigurations/second. We can also see that any number of reconfiguration already reduces the median number of flows on the spine significantly, even as low as 10 reconfigurations / second.



(a) All-to-All traffic / SPF

(b) All-to-All traffic / Hierarchical

(c) Best-case/Worst-case / SPF

(d) Best-case/Worst-case / Hierarchical

Figure 4.8 – Overview of spine occupancy in the number of flows for different reconfiguration rates. Here, CRISS-CROSS has an oversubscription of 4:12 and $R_{tor_up} = 4$. The load is 20%. Shown are both SPF (left) and hierarchical routing (right).

Interestingly, for SPF, even relatively low reconfiguration rate of 100 reconfigurations / second achieves similar results as the unlimited case. Additionally, once again, as we saw in §4.6.3, SPF routing leads to lower numbers of flows on the spine than hierarchical routing.

The results in Figure 4.8c and Figure 4.8d for ABW show a greater spread. This is expected, as the two sub-patterns are completely orthogonal. For the non-reconfigurable case, either all, or none of the flows cross the spine. In this case, CRISS-CROSS reduces the average number of

flows on the spine of over 3.5× for 100 reconfigurations/sec, 16× for 500 reconfigurations/sec, and 52× for the unlimited case.

4.6.5 Sustainable Load

In this experiment, we compare the flow completion times for various loads and show that for a given SLO, CRISS-CROSS can sustain a significantly higher load average. For this, we compare the median, the 90th and 99th percentiles for both the hierarchical routing algorithm (left) and SPF routing (right) for the all-to-all traffic pattern in Figure 4.9. Again, we use the same topology with a 1:3 oversubscription ratio and four uplinks. The vertical line is the maximum theoretical load at which all spine links would be saturated 100% of the time for the non-reconfigurable case. Due to the different length of individual flows, we have normalized the flow completion times by dividing it with the completion time in an uncongested case, effectively resulting in a slow-down factor.

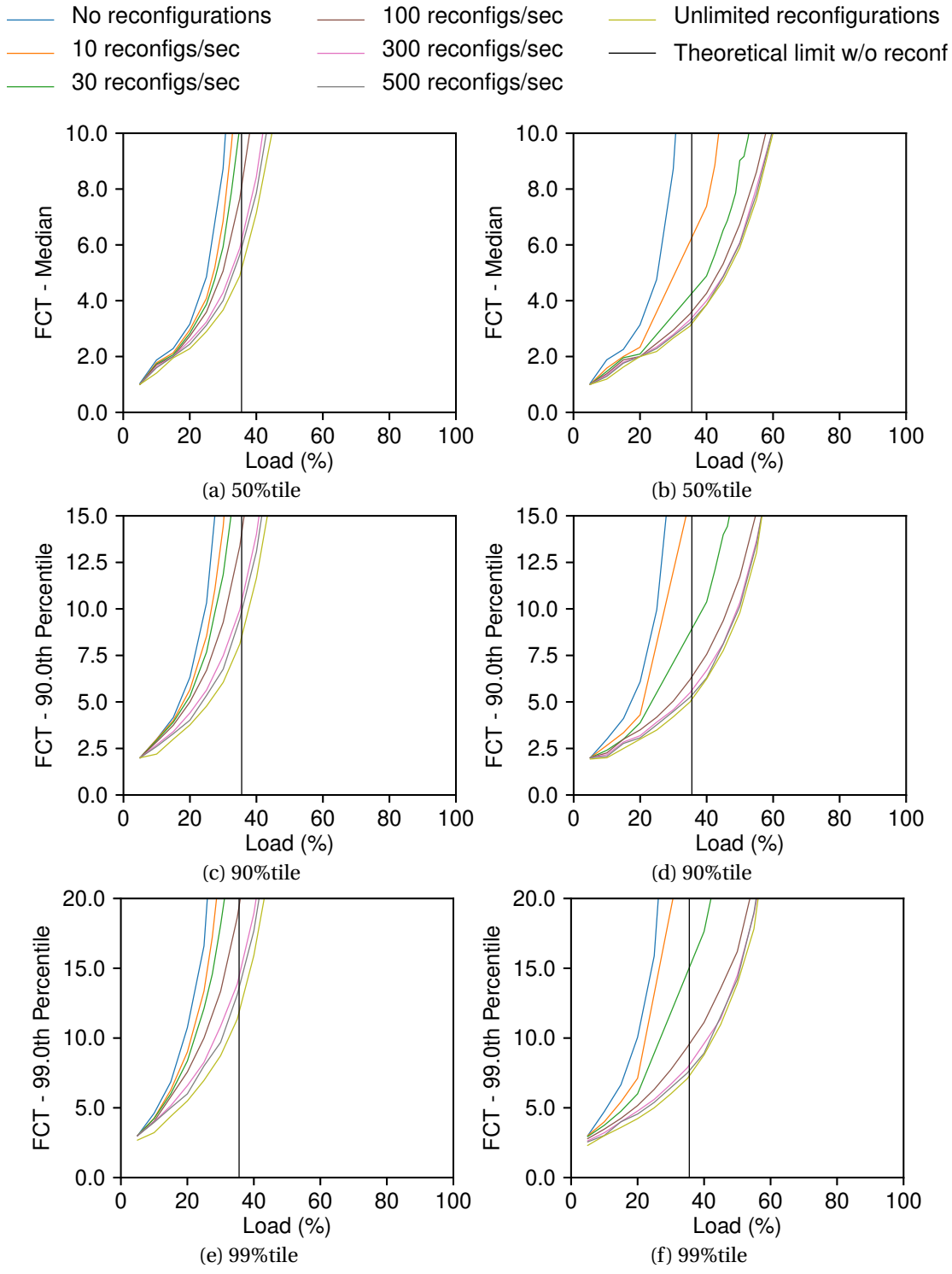
As before, we compare multiple reconfiguration rates. As the load approaches the theoretical maximum, the flow completion time of the non-reconfigurable topology goes to infinity as expected. CRISS-CROSS, for both routing algorithms and all reconfiguration rates, has a significantly higher limit.

For the hierarchical routing, for an SLO of 10× at the 99th percentile, CRISS-CROSS is able to sustain a load approximately 30% higher than the non-reconfigurable version for reconfiguration rates of 300 reconfigurations/second or higher, with only limited additional gain on higher reconfiguration rates.

For the shortest path routing, 100 reconfigurations / second suffice to achieve a similar performance as unlimited reconfigurations. CRISS-CROSS with this reconfiguration rate is able to provide similar flow completion times as the static topology even at $> 2\times$ the offered load. Even low rates of 10-30 reconfigurations / second significantly improve the performance.

4.6.6 Reconfiguration Rate and Flow Completion Times

This section investigates the effect of reconfiguration on the distribution of flow completion times. For this, we present the complementary cumulative distribution function (CCDF) of the normalized flow completion time. The metric reports 1 for uncontended flows and > 1 when links are shared across flows. We choose a target load of 20%, as it is still within the theoretical limit for the topology analyzed in §4.6.5. As a baseline for maximally achievable



(a) 50%tile (b) 50%tile

(c) 90%tile (d) 90%tile

(e) 99%tile (f) 99%tile

Figure 4.9 – Load offered vs flow complettimes, 4:12 oversubscription, 4 uplinks, all-to-all traffic scenario. Hierarchical routing (left), SPF (right)

performance, we also include a non-blocking topology without any oversubscription in our results in addition to the varying reconfiguration rates from before.

Figure 4.10 and Figure 4.11 show the result for the RATA scenario. The figures on the top have a 1:7, the ones on the bottom a 1:3 oversubscription ratio. The figures on the left have only four uplinks, while the figures on the right have eight uplinks per rack. This means that the topology at the bottom left is the same as the one used in §4.6.5.

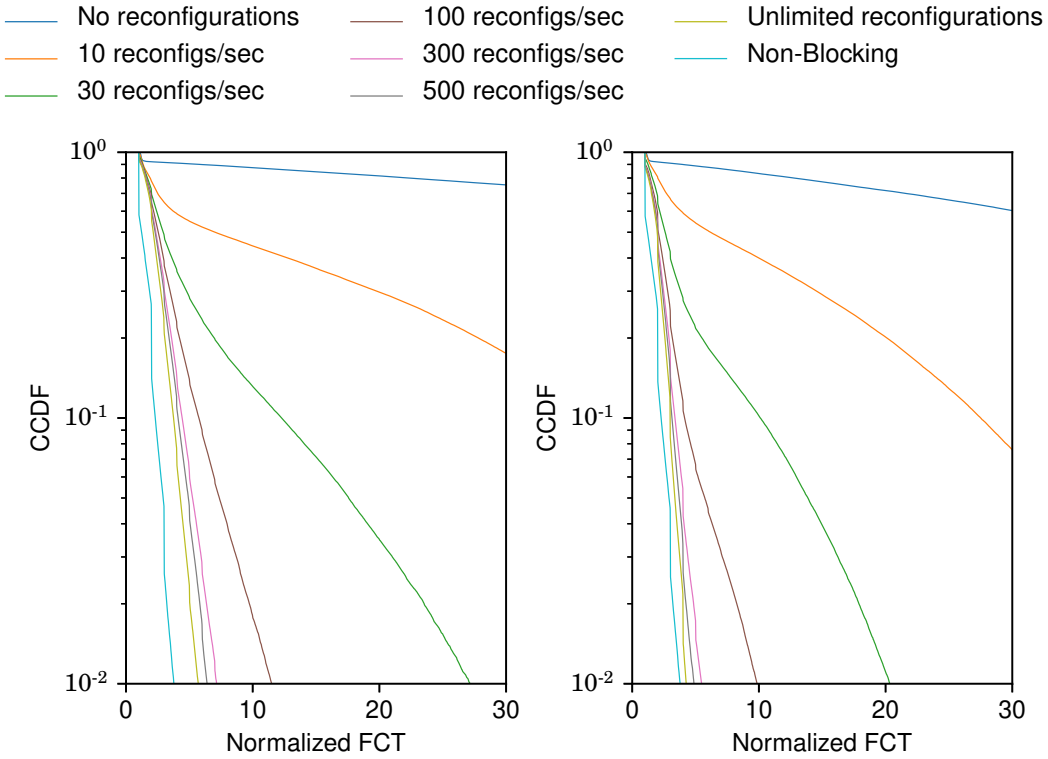
We first look at the shortest-path routing case. In Figure 4.10, we can see that introducing reconfiguration reduces the FCT by over 2× across all percentiles. For the 1:3 oversubscription in Figures 4.10c and 4.10d, there only is a small difference between the reconfiguration rates, and the reconfigurable topology even comes close to the fully non-blocking topology in terms of FCTs. For the higher oversubscription in Figure 4.10a and 4.10b, the rates of 100 reconfigurations/second and below leads to significantly higher FCTs than the faster rates. However, the non-reconfigurable topology is not able to sustain this load at all. Only the pod-local flows (1/16th of all) have acceptable flow completion times, with a significant amount of FCTs approaching infinity, a sign of overload for the topology. This shows that introducing even low rates of reconfiguration can make loads sustainable that were not sustainable beforehand.

For the hierarchical routing in Figure 4.11, the results are a bit more mixed. For the high-oversubscription case (Figures 4.11a and 4.11b), there is a clear difference in the performance of the different reconfiguration rates, with none coming close to the baseline. For the 1:3 over-subscription (Figures 4.11c and 4.11d), the performance is very similar to the one described in the paragraph above, with minimal differences between the reconfiguration rates. The main takeaway from the latter set of graphs is that SPF nearly always outperforms hierarchical routing.

4.6.7 Sensitivity to Topology Size

The goal of this comparison is to show the effect of different CRISS-CROSS oversubscriptions and uplink counts on flow completion times using the four topologies from §4.6.5 (*i.e.*, 1:3 and 1:7 oversubscription, and 4 vs 8 uplinks). For RATA, we use the same graphs (Figure 4.10 and 4.11) as before. However, we extend our analysis on the same topologies to ABW with Figure 4.12 for SPF routing and Figure 4.13 for the hierarchical routing scheme.

We first investigate the performance for RATA in Figures 4.10 and 4.11. For shortest path routing, and at a load of 20%, there is only a small effect of the level of oversubscription for the reconfigurable topologies. The addition of uplinks (and therefore spine groups) only has a minimal effect on the flow completion times. This is because the flow throughput doubles with



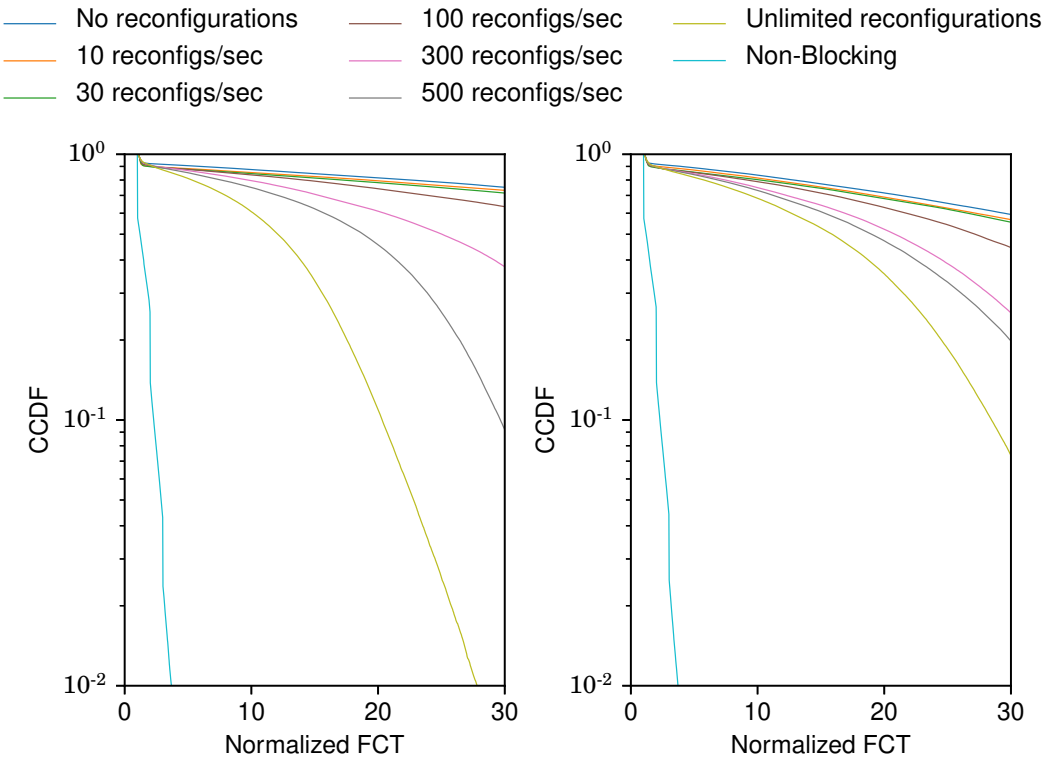
(a) Oversubscription 2:14, $R_{tor\_up} = 4$

(b) Oversubscription 2:14, $R_{tor\_up} = 8$

(c) Oversubscription 4:12, $R_{tor\_up} = 4$

(d) Oversubscription 4:12, $R_{tor\_up} = 8$

Figure 4.10 – Overview over flow completion times for an all-to-all traffic scenario with SPF routing at 20% load



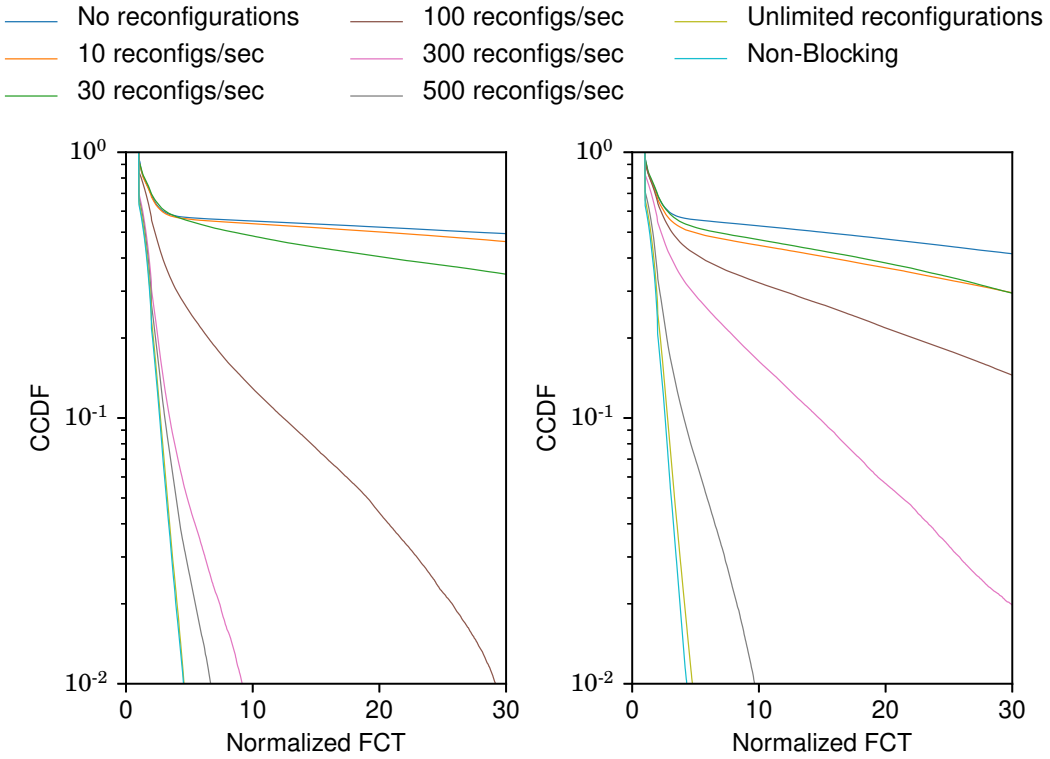
(a) Oversubscription 2:14, $R_{tor\_up} = 4$

(b) Oversubscription 2:14, $R_{tor\_up} = 8$

(c) Oversubscription 4:12, $R_{tor\_up} = 4$

(d) Oversubscription 4:12, $R_{tor\_up} = 8$

Figure 4.11 – Overview over flow completion times for an all-to-all traffic scenario with hierarchical routing at 20% load



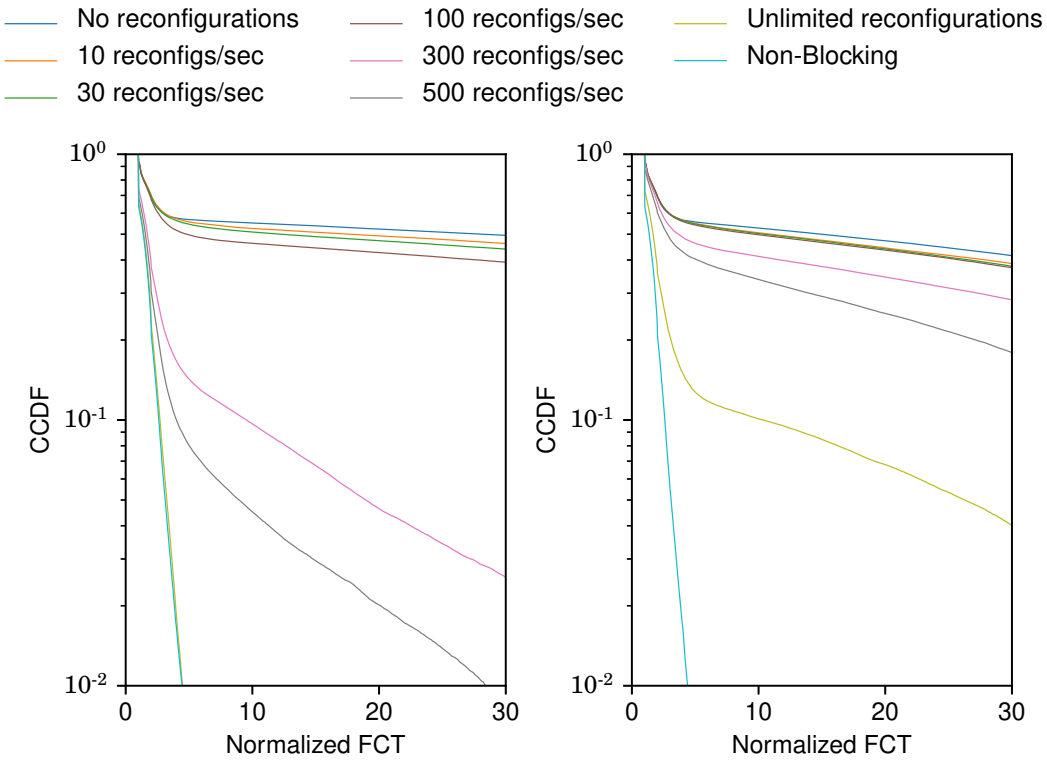
(a) Oversubscription 2:14, $R_{tor\_up} = 4$

(b) Oversubscription 2:14, $R_{tor\_up} = 8$

(c) Oversubscription 4:12, $R_{tor\_up} = 4$

(d) Oversubscription 4:12, $R_{tor\_up} = 8$

Figure 4.12 – Overview over flow completion times for the best/worst case traffic scenario with SPF routing at 20% load



(a) Oversubscription 2:14, $R_{tor\_up} = 4$

(b) Oversubscription 2:14, $R_{tor\_up} = 8$

(c) Oversubscription 4:12, $R_{tor\_up} = 4$

(d) Oversubscription 4:12, $R_{tor\_up} = 8$

Figure 4.13 – Overview over flow completion times for the best/worst case traffic scenario with hierarchical routing at 20% load

the doubling of the uplinks and hence capacity of the topology given our load definition. The most interesting aspect here is that higher oversubscription leads to the static configuration not being able to sustain this load level, and requiring higher reconfiguration rates. Also, introducing more uplinks means that slightly higher reconfiguration rates are required for the same performance targets.

For hierarchical routing, the effect is significantly more pronounced for higher oversubscriptions. The flow completion times rise significantly with lower reconfiguration rates. Comparing Figure 4.10a and 4.10b, we can see the effect of increasing the number of uplinks per rack very clearly. As the number of reconfigurable links goes up, CRISS-CROSS needs higher reconfiguration rates to achieve the same flow completion time targets. In fact, doubling the number of uplinks translates into needing roughly twice as many reconfigurations/second for the same performance.

The situation for ABW is similar. For SPF routing in Figure 4.12, we see that again there is a significant reduction in FCT across the board for the reconfigurable setups. In fact, the lines for unlimited reconfiguration and the baseline are nearly indiscernible as the performance is extremely close. At an oversubscription ratio of 1:7, the static configuration cannot sustain the load, similar to the RATA scenario. The same is true for very low reconfiguration rates. For a low number of rack uplinks, the local flows can still finish; for large numbers of uplinks, even the local links become completely overloaded. Again, similar to what we saw for RATA, increasing the number of uplinks increases the FCT for a given reconfiguration rate. Also, similarly, a reconfiguration rate of 100 is significantly worse than the faster reconfigurations.

However, at a 2:14 oversubscription, the rate of 100 reconfigurations/second achieves similar FCTs to the non-reconfigurable topology at a 1:3 oversubscription. Even in the cases where performance towards the tail diverges from the baseline for slower reconfiguration rates, the majority of flows experience low FCTs. For example, in 4.12b, even though the FCT for a reconfiguration rate of 300 is nearly three times higher than the baseline, the FCTs at the 90th percentile are only about 50% higher.

As was the case with RATA, hierarchical routing simply raises the sensitivity of CRISS-CROSS for ABW. In Figure 4.13, FCTs are overall higher than for shortest path routing, but follow the same trends. On the whole, the speedups of CRISS-CROSS over the non-reconfigurable topology are more significant for ABW than for RATA.

In summary, with a rate of 500 reconfigurations/sec, CRISS-CROSS achieves a reduction in average flow completion times of 5.5x , and a reduction at the 99th percentile of 6.3x for ABW.

For RATA, CRISS-CROSS improves FCTs by 2.2x on average and 3x at the 99th percentile. A higher number of uplinks per ToR requires a higher rate of reconfigurations/second.

Lastly, we are able to demonstrate our main goals: *(1) introducing reconfiguration into the topology allows an increase in the oversubscription ratio while retaining the same performance as the non-reconfigurable topology*, or alternatively *(2), introducing reconfiguration produces better performance for the same oversubscription ratios.*

4.7 Related Work

CRISS-CROSS provides a reconfigurable topology that keeps the same hierarchy as typical data center deployments but augments it with circuit switching technology. In general, there is a trade-off between flexibility of a topology and its structure: the less structured it is, the more complicated is routing. Clos topologies are typical examples of inflexibility, but simple routing; random graphs are at the other extreme with difficult routing.

Optical circuit-based reconfigurable topologies: Helios kickstarted the development of reconfigurable topologies [50]. Helios uses a hybrid architecture of packet and circuit switching, where heavy flows are offloaded to direct circuits between ToRs. c-Through [140] improved on this by using a different, buffer-based method for identifying hot pairs. Reactor [92] and Mordia [51, 121] improved on the prior millisecond-scale reconfiguration times by optimizing the control plane. All these topologies use direct ToR-to-ToR connections and therefore do not scale well in grouped communication or high fan-in/fan-out patterns. OSA [134] took a different approach by using circuit switches to connect ToRs directly. OSA adjusts the topology to optimize for current workloads, but is limited by the cardinality of switches in the number of hosts. It also suffers from complex routing, as ToRs that are not directly connected need to traverse multiple circuits with delays of up to 10ms. The routing for these is made more complex by having structureless topologies. MegaSwitch [25] re-introduces a structure by creating circuits between all ToRs. However, this limits the performance of a circuit to that of one individual link, and it limits the scalability in the number of hosts.

Free space reconfigurable topologies: These designs replace wired links with wireless technology. This removes cabling layout as a limitation for their installation and deployement. There are many representatives of this category, utilizing 60Ghz links [65] with mirrors [145]. In the free-space designs, there are the same two approaches as in the cabled version. ProjecTor [57] creates two parallel architectures, one with connectivity between all ToRs, and direction connections for high-bandwidth requirements. Firefly [9] takes a similar approach

to OSA and allows any possible configuration of the topology made up only of ToRs, without any additional core switches. However, the free space designs suffer from the same problem that a lack of hierarchy causes, and in addition have high requirements on the operating environment such as dust-free data centers.

Reconfiguration in Clos Networks: FlatTree [141] combines the advantages of Clos topologies with those of random graphs by allowing partial reconfiguration. They randomize subsets of the topology according to traffic, removing all structure. On these subsets, this provides the same advantages and disadvantages as we describe under Random Graph Networks below. Lastly, Larry [24] is the closest in spirit to CRISS-CROSS, as it retains the Clos topology and pod structure. It groups the downlinks on a fabric switch between all ToRs in a subset of the pod, allowing flexible bandwidth assignment per ToR. As such, it aims the problem of oversubscription at the rack layer, not at the spine layer, and can be used in combination with the techniques in CRISS-CROSS.

Random Graph Networks: Jellyfish [133] and XPander [137] take a radically different approach and suggest random-graph-like networks to achieve good bisection bandwidth without reconfiguration. These random graphs have a very low diameter network, and therefore high bisection bandwidth per port in the network. Kassing et.al. [82] have argued that the additional costs of reconfigurability can better be invested in more networking devices to improve the total amount of available bandwidth. They are also (nearly) arbitrarily incrementally expendable because they do not need to add complete sets of elements to keep any structure. However, there is an open problem regarding efficient routing. To the best of the author's knowledge, these networks currently rely on k-shortest-path routing and MPTCP for load-balancing. These routes are expensive to calculate, especially in the presence of failures in a data center environment.

Traffic Engineering: Clos topologies have a high degree of path diversity. Traffic engineering tries to load balance traffic across these paths to get an even distribution of traffic and avoid hotspots. Normal ECMP-based load balancing is a static assignment that cannot react to changes in bandwidth needs of individual flows. There is a large body of research in reactive dynamic load balancing techniques including Hedera [2], MPTCP [54], or CONGA [4]. These solutions try to route elephant flows away from hotspots towards lightly loaded paths. Other solutions try to solve this problem at a finer granularity by either load balancing flowlets [135] or per-packet via spraying with Presto [70] or Juggler [56]. Traffic engineering has focused on balancing traffic between spines to remove relative imbalances between them, but does

nothing to increase the available bandwidth over the spines. CRISS-CROSS on the other hand completely offloads the spines.

4.8 Discussion

We briefly discuss some of the trade-offs in our design to explain how a real-world deployment might change certain design decisions based on operational requirements.

The ”application transparency” requirement imposes restrictions on the implementation, in particular to ensure that all traffic goes over non-affected spine groups before any links are to be taken down. This "draining" can be changed so that it either ignores drops, completely prevent reorderings, or one can use techniques such as Juggler [56] or Presto [70] to prevent the impact of reorderings. By deactivating draining, reconfigurations can be a lot quicker and higher reconfiguration rates can be achieved. Depending on the used technology for the OCS and the buffer capacity of the ASICs, it might even be feasible to simply buffer the packets during reconfiguration.

While we did not study it specifically, the use of modular, scalable OCS building blocks can be used for other purposes, and in particular to simplify the incremental deployment of data centers, or the hardware upgrade of fabric or spine switches in an existing production datacenter. In fact, cost-effective, programmable OCS technology will very likely be introduced first in data centers to address these administrative and operational concerns. Our approach to CRISS-CROSS addresses in fact the same problem, but at a totally different time scale (sub-seconds in our case vs. planned administrative operations).

We presented an implementation that serializes phases of the algorithm and limits decision to a single reconfiguration at a time. Depending on the technology bottlenecks, concurrent reconfigurations that consists of overlapping phases may be required to allow the network to adjust to rapid changes in network patterns. We leave this for future work.

Our implementation relies on hierarchy to calculate routes and ECMP to distribute the load. As a consequence, the shortest path is not always taken between two pairs of racks. Unlike standard Clos topologies, the strict use of shortest-path routing is a tradeoff as all traffic between two ToRs will flow via the only link that are connected to the same fabric switch. For CRISS-CROSS, it is a tradeoff, and shortest-path routes can be easily inserted for paths between selected ToR.

4.9 Conclusion

We present CRISS-CROSS to address the network challenge of adapting to changing traffic patterns in oversubscribed data-center networks. CRISS-CROSS is designed to respect the proven advantages of modularity, structure, hierarchy, cost and scalability found in oversubscribed Clos networks, but increase performance by dissolving pods and breaking the transitive 1-hop routing property of classic folded Clos networks. Our evaluation shows that dynamic reconfiguration reduces spine traffic in worst-case and best-case scenarios.

Our research contribution assumes that the emergence of scalable, programmable optical circuit switches will offer new reconfiguration opportunities. Assuming this to be the case, CRISS-CROSS offers a backward-compatible enhancement over the proven de-facto standard. While CRISS-CROSS is designed to work with existing commodity switch ASIC and firmware, we do expect that engineering changes will be required to increase the supported reconfiguration rates.

5 Discussion

5.1 Future work

There is a multitude of further research ideas for both presented projects. In the following paragraphs, we present some of the ideas for further development, first for the individual projects and afterwards for other uses of locality in a network context.

In recent years, there has been a rise in programmable data planes; both in SW [37] and FPGAs [53]at the end-host , as well as in the network itself, *i.e.*, in the form of programmable switches with P4 [15] being the most prominent representative. These new data planes provide novel opportunities for placing features in the network as well as improved programmability in general. This programmability allows new levels of introspection into the network, *e.g.*, by allowing finer grained and more accurate elephant flow identification [68]. Elephant flow identification is a fundamental aspect to utilize flows, so further developments in this area will be directly applicable to both projects.

VNTOR : VNTOR itself was implemented on an off-the-shelf switch using a Broadcom Trident ASIC with limited data plane programmability. Using programmable data planes such as P4 [15], it might be possible to move some of the logic into the data plane itself. This could be useful for improving the cache replacement algorithm. However, it is at the moment not possible to update tables from the data plane itself, so the additional value of P4 switches needs to be evaluated.

At the same time, P4 switches suffer from the same issues as traditional top-of-rack switches that the TCAM space is limited. It should therefore be possible to modify the strategies in VNTOR to modify the compiler and toolchain of P4 to split the rule space between the control CPU in a P4 switch and the match action tables.

Another approach to VNToR would be an implementation at different points in the network. Given the better programmability of P4, it might be possible to further split the state between multiple switches without too much overhead as a hybrid version of DIFANE [144] and VNToR.

Alternatively, the end-host might become a more interesting point of implementation with the advent of FPGAs in networking cards. From the available information, it is most likely that Amazon is using a variation of this [138]. Microsoft's SmartNIC [53] is similar in spirit to VNToR in that it has slow and fast-paths, where flows get moved into the fast-path as needed. The additional advantage of placing the implementation at the end-host is the larger number of hardware units across which state can be separated. The downside of such an approach is of course that the number of hardware units increases and therefore management overhead (and cost) as well.

From an algorithmic perspective, the identification of heavy-flows in VNToR is relatively basic compared to some of the literature, as this was not a focus of our work. Implementing different algorithms for this identification in particular with the limitations of the ASIC itself could provide optimization potential for higher cache hit rates for a given reconfiguration rate.

Lastly, the advent both of new interconnects between the control plane, different processors in the control plane, faster link speeds in the network and changes in workloads might shift the balance around for the best algorithms to use in VNToR.

Criss-Cross : One of the biggest open questions in Criss-Cross is both the actual feasibility of large OCSs as well as their cost. Therefore, further work is required to evaluate what the actual limits for OCS sizes might be, and for at which costs they can provide benefits versus non-oversubscribed topologies.

A subset of this research question is how a limitation of OCS sizes would influence the design of Criss-Cross. With smaller OCSs, the set of possible configurations for a topology would be reduced. However, within limits, this reduction could still provide benefits as it would support some loosening of the pod membership relationship of individual racks.

For a full implementation, significant optimizations could be made by parallelizing the changes in configuration to the OCS if the underlying technology supports it.

Another part of this equation is the scheduler. Given some flexibility in network configuration, would this allow to extend the scheduler so that it takes the current network configuration more into account? The scheduler (or even the load balancer) has direct impact on the

efficiency of the network, even more so in an asymmetrical design. Therefore, an integration might be worthwhile.

From an evaluation perspective, several directions should be further researched. The reconfiguration speed of the OCS itself has a big influence on the actual performance and the algorithms used in CRISS-CROSS. Slow reconfigurations might mean that link offloading is a useful technique, while very fast operation might allow to skip this step completely. Backup routes could reduce the impact even in the case that no off-loading exists.

Lastly, CRISS-CROSS looked at only a subset of Clos topologies with three layers. A generalization to more layers could provide even more opportunities for reconfigurations. In the same vain, a further study across different radices of switches is needed. Very large radices would allow for very shallow Clos topologies and therefore potentially reduce CRISS-CROSS' usefulness, while lower switch radices would potentially increase it.

Other uses of locality: The work presented here used network locality within the topology as well as within an individual ToR switch. However, network traffic touches a multitude of other areas as well. One area mentioned above already is that of scheduling and load balancing. Using either of these, operators of data centers have the opportunity to increase the amount of locality in the data center. Then, going down further in the stack, there might be additional uses of network locality within the server at all levels, within network cards, network acceleration cards such as FPGA based NICs, or within the networking stack, although many of these have already been heavily optimized.

5.2 Conclusion

This thesis describes how locality, particularly network locality, can be used in the data center as an opportunity for more efficiency in various metrics. We presented two projects showing the use of locality at different levels in the network, VNTOR and CRISS-CROSS

VNTOR shows that using the locality of flows at the top-of-rack, it is possible to implement the network virtualization abstraction in hardware. Specifically, we showed that such an implementation is possible at the top-of-rack. Moving the abstraction there enables bare-metal support for cloud data centers, avoids the performance overheads of an implementation in software at the host OS, and reduces the susceptibility to hypervisor breakouts.

CRISS-CROSS shows that using locality of flows at rack granularity allows to design a topology that adapts to the changing traffic patterns in the data center network, but also retains the advantages of current data center architectures such as modularity, and hierarchical structure.

Bibliography

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference*, pages 63–74, 2008.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 281–296, 2010.

[3] Albert Greenberg. SDN for the Cloud. http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/keynote.pdf, 2015.

[4] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 503–514, 2014.

[5] Amazon. Announcing amazon elastic compute cloud (amazon ec2) - beta. https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta/.

[6] Amazon. EC2 Enhanced Linux Networking. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.

[7] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network. https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, 2014.

[8] M. Arregoces and M. Portolani. *Data center fundamentals*. Cisco Press, 2003.

[9] N. H. Azimi, Z. A. Qazi, H. Gupta, V. Sekar, S. R. Das, J. P. Longtin, H. Shah, and A. Tanwer. FireFly: a reconfigurable wireless data center fabric using free-space optics. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 319–330, 2014.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[11] L. A. Barroso. Warehouse-Scale Computing: Entering the Teenage Decade. *SIGARCH Computer Architecture News*, 39(3), 2011.

[12] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 267–280, 2010.

[13] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2010.

[14] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. In *Proceedings of the 1st ACM SIGCOMM 2009 Workshop on Research on Enterprise Networking (WREN)*, pages 65–72, 2009.

[15] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *Computer Communication Review*, 44(3):87–95, 2014.

[16] Broadcom Corp. Broadcom BCM56840 Series. https://www.broadcom.com/products/Switching/Carrier-and-Service-Provider/BCM56840-Series, 2010.

[17] Broadcom Corp. Broadcom XLP-300 Lite Series Processor Family. http://www.broadcom.com/products/Processors/Data-Center/XLP300-Series, 2013.

[18] N. Brownlee and K. Claffy. Understanding internet traffic streams: dragonflies and tortoises. *Communications Magazine, IEEE*, 40(10):110–117, Oct 2002.

[19] N. Brownlee, C. Mills, and G. Ruth. Traffic Flow Measurement: Architecture. RFC 2722 (Informational), Oct. 1999.

[20] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, 2012.

[21] E. Bugnion, J. Nieh, and D. Tsafrir. *Hardware and Software Support for Virtualization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[22] Calient S320 optical circuit switching. http://www.calient.com, 2018.

[23] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, page 8. ACM, 2010.

[24] A. Chatzieleftheriou, S. Legtchenko, H. Williams, and A. I. T. Rowstron. Larry: Practical Network Reconfigurability in the Data Center. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 141–156, 2018.

[25] L. Chen, K. Chen, Z. Zhu, M. Yu, G. Porter, C. Qiao, and S. Zhong. Enabling Wide-Spread Communications on Optical Fabric with MegaSwitch. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 577–593, 2017.

[26] Cisco. Cisco global cloud index: Forecast and methodology, 2016–2021 white paper.

[27] Cisco. Cisco IOS NetFlow. http://www.cisco.com/web/go/netflow.

[28] Cisco. Cisco IOS Technologies. https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-technologies/index.html, April 2019.

[29] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[30] Common Vulnerabilities and Exposures. CVE-2008-0923. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0923, 2008.

[31] Common Vulnerabilities and Exposures. CVE-2009-1244 – "CloudBurst". http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1244, 2009.

[32] Common Vulnerabilities and Exposures. CVE-2011-1751 – "virtunoid". http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1751, 2011.

[33] Common Vulnerabilities and Exposures. CVE-2012-0217. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217, 2012.

[34] Cumulus. Cumulus Linux. https://cumulusnetworks.com/products/cumulus-linux/, April 2019.

[35] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *Proceedings of the 2011 IEEE Conference on Computer Communications (INFOCOM)*, pages 1629–1637, 2011.

[36] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 254–265, 2011.

[37] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 373–387, 2018.

[38] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[39] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

[40] Y. Deng. *Applied parallel computing*. World Scientific, 2013.

[41] P. J. Denning. Working Sets Past and Present. *IEEE Trans. Software Eng.*, 6(1):64–84, 1980.

[42] P. J. Denning. The locality principle. *Commun. ACM*, 48(7):19–24, 2005.

[43] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. *J. Parallel Distrib. Comput.*, 72(11):1471–1480, 2012.

[44] J. Erman, A. Mahanti, and M. F. Arlitt. Byte me: a case for byte accuracy in traffic classification. In *Proceedings of the 3rd Annual ACM Workshop on Mining Network Data (MineNet)*, pages 35–38, 2007.

[45] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula. CloudBuild: Microsoft's distributed and caching build service. In *ICSE (Companion Volume)*, pages 11–20, 2016.

[46] ESnet, LBNL. iperf3. http://software.es.net/iperf/.

[47] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *Proceedings of the ACM SIGCOMM 2002 Conference*, pages 323–336, 2002.

[48] Facebook Inc. Facebook Open Switching System (FBOSS). https://github.com/facebook/fboss, April 2019.

[49] N. Farrington and A. Andreyev. Facebook's data center network architecture. In *2013 Optical Interconnects Conference*, pages 49–50. Citeseer, 2013.

[50] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 339–350, 2010.

[51] N. Farrington, G. Porter, P.-C. Sun, A. Forencich, J. E. Ford, Y. Fainman, G. Papen, and A. Vahdat. A demonstration of ultra-low-latency data center optical circuit switching. In *Proceedings of the ACM SIGCOMM 2012 Conference*, pages 95–96, 2012.

[52] J. Fietz, S. Whitlock, G. Ioannidis, K. J. Argyraki, and E. Bugnion. VNToR: Network Virtualization at the Top-of-Rack Switch. In *Proceedings of the 2016 ACM Symposium on Cloud Computing (SOCC)*, pages 428–441, 2016.

[53] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 51–66, 2018.

[54] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental), Jan. 2013.

[55] Gartner, Inc. Gartner forecasts worldwide public cloud revenue to grow 21.4 percent in 2018. https://www.gartner.com/newsroom/id/3871416.

[56] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh. Juggler: a practical reordering resilient network stack for datacenters. In *Proceedings of the 2016 EuroSys Conference*, pages 20:1–20:16, 2016.

[57] M. Ghobadi, R. Mahajan, A. Phanishayee, N. R. Devanur, J. Kulkarni, G. Ranade, P.-A. Blanche, H. Rastegarfar, M. Glick, and D. C. Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 216–229, 2016.

[58] M. Glick, D. G. Andersen, M. Kaminsky, and L. Mummert. Dynamically reconfigurable optical links for high-bandwidth data center networks. In *Optical Fiber Communication-incudes post deadline papers, 2009. OFC 2009. Conference on*, pages 1–3. IEEE, 2009.

[59] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong. The Characteristics of Cloud Computing. In *ICPP Workshops*, pages 275–279, 2010.

[60] Google. GCE. Google Compute Engine. http://cloud.google.com/compute.

[61] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 51–62, 2009.

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

[63] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 63–74, 2009.

[64] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 Conference*, pages 75–86, 2008.

[65] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 38–49, 2011.

[66] J. Hamilton. Overall Data Center Costs. http://mvdirona.com/jrh/TalksAndPapers/PerspectivesDataCenterCostAndPower.xls. Accessed: 08-13-2016.

[67] J. Hamilton. AWS re:Invent 2014 | (SPOT301) AWS Innovation at Scale. https://www.youtube.com/watch?v=JIQETrFC_SQ, 2014.

[68] R. Harrison, Q. Cai, A. Gupta, and J. Rexford. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proceedings of the Symposium on SDN Research (SOSR)*, pages 8:1–8:7, 2018.

[69] S. Hasan, P. Lapukhov, A. Madan, and O. Baldonado. Open/R: Open routing for modern networks.

[70] K. He, E. Rozner, K. Agarwal, W. Felter, J. B. Carter, and A. Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 465–478, 2015.

[71] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, K. N. B., C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev, S. Padgett, F. Rabe, S. Ray, M. Tewari, M. Tierney, M. Zahn, J. Zolla,

J. Ong, and A. Vahdat. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 74–87, 2018.

[72] IBM. IBM Cloud. http://www.ibm.com/cloud.

[73] IEEE 802. 802.1Qbb – Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/, April 2019.

[74] R. Jain and S. Paul. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, 51(11):24–31, 2013.

[75] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 3–14, 2013.

[76] X. Jin, N. Farrington, and J. Rexford. Your Data Center Switch is Trying Too Hard. In *Proceedings of the Symposium on SDN Research (SOSR)*, page 12, 2016.

[77] Juniper. Juniper Flow Monitoring. https://www.juniper.net/us/en/local/pdf/app-notes/3500204-en.pdf.

[78] Juniper Networks. Junos OS. https://www.juniper.net/uk/en/products-services/nos/junos/, April 2019.

[79] S. A. Jyothi, M. Dong, and B. Godfrey. Towards a flexible data center fabric with source routing. In *Proceedings of the Symposium on SDN Research (SOSR)*, pages 10:1–10:8, 2015.

[80] S. Kandula, S. Sengupta, A. G. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 202–208, 2009.

[81] N. Kang, Z. Liu, J. Rexford, and D. Walker. An Efficient Distributed Implementation of One Big Switch. Open Networking Summit, 2013.

[82] S. Kassing, A. Valadarsky, G. Shahaf, M. Schapira, and A. Singla. Beyond fat-trees without antennae, mirrors, and disco-balls. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 281–294, 2017.

[83] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks. In *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, 2016.

[84] A. Kivity. KVM: the Linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS)*, pages 225–230, July 2007.

[85] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[86] J. Koomey, K. Brill, P. Turner, J. Stanley, and B. Taylor. A simple model for determining true total cost of ownership for data centers. *Uptime Institute White Paper, Version*, 2:2007, 2007.

[87] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. J. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 203–216, 2014.

[88] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)*, pages 351–364, 2010.

[89] K.-C. Lan and J. S. Heidemann. A measurement study of correlations of Internet flow characteristics. *Computer Networks*, 50(1):46–62, 2006.

[90] P. Lapukhov, A. Premji, and J. Mitchell. Use of BGP for Routing in Large-Scale Data Centers. RFC 7938 (Informational), Aug. 2016.

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

[92] H. Liu, F. Lu, A. Forencich, R. Kapoor, M. Tewari, G. M. Voelker, G. Papen, A. C. Snoeren, and G. Porter. Circuit Switching Under the Radar with REACToR. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–15, 2014.

[93] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2010.

[94] Y. J. Liu, P. X. Gao, B. Wong, and S. Keshav. Quartz: a new design element for low-latency DCNs. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 283–294, 2014.

[95] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[96] G. Lu, R. Miao, Y. Xiong, and C. Guo. Using CPU as a Traffic Co-processing Unit in Commodity Switches. In *Proceedings of the ACM Workshop on Hot Topics in Software Defined Networks (HotSDN)*, 2012.

[97] M. Mahalingam, D. G. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. IETF RFC 7348, 2014.

[98] N. McKeown, T. Anderson, H. Balakrishnan, G. M. Parulkar, L. L. Peterson, J. Rexford, S. Shenker, and J. S. Turner. OpenFlow: enabling innovation in campus networks. *Computer Communication Review*, 38(2):69–74, 2008.

[99] W. M. Mellette and J. E. Ford. Scaling limits of mems beam-steering switches for data center networks. *Journal of Lightwave Technology*, 33(15):3308–3318, 2015.

[100] W. M. Mellette, R. McGuinness, A. Roy, A. Forencich, G. Papen, A. C. Snoeren, and G. Porter. RotorNet: A Scalable, Low-complexity, Optical Datacenter Network. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 267–280, 2017.

[101] W. M. Mellette, G. M. Schuster, G. Porter, G. Papen, and J. E. Ford. A scalable, partially configurable optical switch for data center networks. *Journal of Lightwave Technology*, 35(2):136–144, 2017.

[102] Microsoft Corporation. Microsoft Azure. http://azure.microsoft.com.

[103] Microsoft Corporation. Receive Side Scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx.

[104] Microsoft Corporation. Microsoft showcases the Azure Cloud Switch (ACS). https://azure.microsoft.com/en-us/blog/microsoft-showcases-the-azure-cloud-switch-acs/, April 2019.

[105] N. Mohan and M. Sachdev. Low-Leakage Storage Cells for Ternary Content Addressable Memories. *IEEE Trans. VLSI Syst.*, 17(5):604–612, 2009.

[106] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–13, 2013.

[107] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto. Identifying elephant flows through periodically sampled packets. In *Proceedings of the 4th ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 115–120, 2004.

[108] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan. Scalable Rule Management for Data Centers. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 157–170, 2013.

[109] R. N. Mysore, G. Porter, and A. Vahdat. FasTrak: enabling express lanes in multi-tenant data centers. In *Proceedings of the 2013 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 139–150, 2013.

[110] G. T. K. Nguyen, R. Agarwal, J. Liu, M. Caesar, B. Godfrey, and S. Shenker. Slick packets. In *Proceedings of the 2011 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 245–256, 2011.

[111] D. Nikolova, S. Rumley, D. Calhoun, Q. Li, R. Hendry, P. Samadi, and K. Bergman. Scaling silicon photonic switch fabrics for data center interconnection networks. *Optics express*, 23(2):1159–1175, 2015.

[112] OpenStack. Neutron Project. https://wiki.openstack.org/wiki/Neutron.

[113] OpenStack. Neutron SR-IOV Passthrough Networking Documentation. https://wiki.openstack.org/w/index.php?title=SR-IOV-Passthrough-For-Networking&oldid=93943.

[114] OpenStack. OpenStack Project. https://www.openstack.org/.

[115] K. Papagiannaki, N. Taft, S. Bhattacharyya, P. Thiran, K. Salamatian, and C. Diot. A pragmatic definition of elephants in internet backbone traffic. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement (IMC)*, pages 175–176, 2002.

[116] PCI-SIG. *Single Root I/O Virtualization and Sharing Specification Revision 1.1*. PCI-SIG, January 2010.

[117] L. Pedrosa, R. K. Iyer, A. Zaostrovnykh, J. Fietz, and K. J. Argyraki. Automated synthesis of adversarial workloads for network functions. In *Proceedings of the ACM SIGCOMM 2018 Conference*, pages 372–385, 2018.

[118] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 117–130, 2015.

[119] Polatis Series 6000. http://www.polatis.com, 2018.

[120] D. A. Popescu, G. Antichi, and A. W. Moore. Enabling Fast Hierarchical Heavy Hitter Detection using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research (SOSR)*, pages 191–192, 2017.

[121] G. Porter, R. D. Strong, N. Farrington, A. Forencich, P.-C. Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating microsecond circuit switching into the data center. In *Proceedings of the ACM SIGCOMM 2013 Conference*, pages 447–458, 2013.

[122] C. Raiciu, M. Ionescu, and D. Niculescu. Opening Up Black Box Networks with CloudTalk. In *Proceedings of the 4th workshop on Hot topics in Cloud Computing (HotCloud)*, 2012.

[123] R. M. Ramos, M. Martinello, and C. E. Rothenberg. SlickFlow: Resilient source routing in Data Center Networks unlocked by OpenFlow. In *Proceedings of the 38th IEEE Conference on Local Computer Networks (LCN)*, pages 606–613, 2013.

[124] Rick Jones. NetPerf: a network performance benchmark. http://www.netperf.org/netperf/, 1996.

[125] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. *Computer Communication Review*, 45(5):123–137, 2015.

[126] J. H. Saltzer and M. F. Kaashoek. *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[127] R. Seifert and J. Edwards. *The All-New Switch Book: The Complete Guide to LAN Switching Technology*. John Wiley &, 2008.

[128] T. J. Seok, N. Quack, S. Han, R. S. Muller, and M. C. Wu. Highly scalable digital silicon photonic mems switches. *Journal of Lightwave Technology*, 34(2):365–371, 2016.

[129] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. M. Parulkar. Can the Production Network Be the Testbed? In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)*, pages 365–378, 2010.

[130] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the ACM SIGCOMM 1995 Conference*, pages 231–242, 1995.

[131] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, H. Liu, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: a decade of clos topologies and centralized control in Google's datacenter network. *Commun. ACM*, 59(9):88–97, 2016.

[132] A. Singla, P. B. Godfrey, and A. Kolla. High Throughput Data Center Topology Design. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–41, 2014.

[133] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 225–238, 2012.

[134] A. Singla, A. Singh, and Y. Chen. OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 239–252, 2012.

[135] S. Sinha, S. Kandula, and D. Katabi. Harnessing tcp's burstiness with flowlet switching. In *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*. Citeseer, 2004.

[136] Synergy Research Group. Hyperscale Capex Reached $22 billion in Q4 and is Still Growing Rapidly.

[137] A. Valadarsky, G. Shahaf, M. Dinitz, and M. Schapira. Xpander: Towards Optimal-Performance Datacenters. In *Proceedings of the 2016 ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, pages 205–219, 2016.

[138] W. Vogels. 10 Lessons from 10 Years of Amazon Web Services. http://www.allthingsdistributed.com/2016/03/10-lessons-from-10-years-of-aws.html, 2016.

[139] J. Wallerich, H. Dreger, A. Feldmann, B. Krishnamurthy, and W. Willinger. A methodology for studying persistency aspects of internet flows. *Computer Communication Review*, 35(2):23–36, 2005.

[140] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. P. Ryan. c-Through: part-time optics in data centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 327–338, 2010.

[141] Y. Xia, X. S. Sun, S. Dzinamarira, D. Wu, X. S. Huang, and T. S. E. Ng. A Tale of Two Topologies: Exploring Convertible Data Center Network Architectures with Flat-tree. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 295–308, 2017.

[142] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao. CAB: a reactive wildcard rule caching system for software-defined networks. In *Proceedings of the 3rd workshop on Hot topics in software defined networking (HotSDN)*, pages 163–168, 2014.

[143] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE, 2008.

[144] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 351–362, 2010.

[145] X. Zhou, Z. Zhang, Y. Zhu, Y. Li, S. Kumar, A. Vahdat, B. Y. Zhao, and H. Zheng. Mirror mirror on the ceiling: flexible wireless links for data centers. In *Proceedings of the ACM SIGCOMM 2012 Conference*, pages 443–454, 2012.

Jonas Fietz ✉ Ch. de Veilloud 9, 1024 Ecublens ✆ +41 (78) 6582998 ✉@ info@jonasfietz.de

Curriculum Vitae

Jonas Fietz

Nationality:	German
Date of Birth:	03.12.1985, Georgsmarienhütte
Languages:	German (native), English (fluent), French (basic)

Education

09/2013 – today	**PhD student** École Polytechnique Fédérale de Lausanne Networking Architecture Lab (Advisor: Prof. Katerina Argyraki) Data Center Systems Lab (Co-Advisor: Prof. Edouard Bugnion)
10/2005 – 01/2013	**Diploma in Computer Science** (1.1 - Top 10%) Karlsruhe Institute of Technology (KIT), Diploma Computer Science Areas of Specialisation: *Operating Systems* and *Software Engineering and Compiler Construction* Diploma Thesis: *Performance Optimization of Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries;* Advisors: Prof. Dr. Peter Sanders, Prof. Dr. Vincent Heuveline
10/2006 – 12/2012	**Diploma in Mathematics** (1.7) Karlsruhe Institute of Technology (KIT), Diploma Mathematics

Publications

ATC 2019	*R2P2: Making RPCs first-class datacenter citizens* Marios Kogias, Adrien Ghosn, George Prekas, Jonas Fietz, Edouard Bugnion
SIGCOMM 2018	*Automated Synthesis of Adversarial Workloads for Network Functions* Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, Katerina Argyraki
SOCC 2016	*VNToR: Network virtualization at the top-of-rack switch* Jonas Fietz, Sam Whitlock, George Ioannidi, Katerina Argyraki, Edouard Bugnion
ICSE 2016	*CloudBuild: Microsoft's distributed and caching build service* Hamed Esfahani, Jonas Fietz, et.al.
EuroPar 2012	*Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries* J. Fietz, M.J. Krause, C. Schulz, P. Sanders, V. Heuveline

Work Experience

09/2014 – today	**PhD Student** at École Polytechnique Fédérale de Lausanne
07/2016 – 09/2016	**Google Intern -** GMail Backend Performance Team - Mountain View
06/2015 – 09/2015	**Microsoft Research Intern** - Redmond
09/2013 – 08/2014	**PhD Fellow** at École Polytechnique Fédérale de Lausanne
01/2006 – 08/2013	**Freelancing Developer** (PHP/MySQL; C++; Python/PostgreSQL/MySQL) e.g. for Müller New Media, Synlife
08/2009 – 10/2010	**Research Assistant** at the Engineering Mathematics and Computing Lab (C/C++)
07/2007 – 01/2008	Member of **Information Systems Team**; Student Space Exploration and Technology Initiative - European Space Agency (ESA)

Lausanne - 15.06.2019

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**