Dynamic Control of Software-Defined Networks

Xin Jin

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Jennifer Rexford

September 2016

Abstract

Network management is critical to provide fast, reliable and secure network services. Software-defined networking (SDN) is a new network architecture to simplify network management by integrating network control to a centralized control platform.

Network operators run various applications on the control platform to perform different management tasks, like routing, monitoring, load balancing and firewall. These applications have complex interactions with each other, making it difficult to deploy and reason about their behaviors. The frequent network events, such as traffic shifts, cyber attacks, and device failures, further exacerbates the problem. Each application needs to reconfigure the network, in order to react to the events. It is challenging to correctly and efficiently combine configuration changes from multiple applications, distribute these changes to a distributed collection of network devices, and coordinate changes across network devices in different layers.

In this thesis, we present a new control architecture that can efficiently handle network events for multiple applications and across the network and optical layers. We identify and study the following three key components of the architecture.

(i) CoVisor: A network hypervisor that can compose multiple applications and can efficiently merge configuration changes from these applications in the face of network events. To protect the network from malicious and buggy applications, CoVisor also provides topology virtualization and fine-grained access control to constrain what each application can see and do.

(ii) Dionysus: A network update scheduler that can quickly and consistently distribute configuration changes to multiple switches. Dionysus uses a dependency graph to capture the dependencies between update operations, and dynamically schedules the operations based on runtime conditions. The approach both eliminates undesirable transient behaviors like loops, blackholes and congestion, and reduces the update time.

(iii) Owan: A traffic management system that can jointly control the optical and network layers. Owan optimizes optical circuit setup, routing and rate allocation together, and dynamically adapts them to workload changes. The joint management significantly improves data transfers over the wide area network.

We have built software controllers and hardware testbeds, and evaluated them with prototype experiments and large-scale simulations using network topologies and traffic traces from production networks.

Acknowledgments

I am extremely grateful to my advisor, Jennifer Rexford, for her patience, guidance and encouragement in the past five years. She is the best advisor one can have in graduate school. Jen opened the world of computer networking to me, and taught me how to do top-notch research, from developing good research taste to designing elegant solutions. Besides the training on research, Jen also offered me with a lot of opportunities to attend conferences and do internships, and introduced me to many world-class scientists in the field. In addition to always being accessible and providing detailed help whenever I need them, Jen is also very open-minded and gives me enough freedom to pursue ideas I find most exciting. She was very supportive when I told her I wanted to work on optical networking, which was not within her research agenda. She connected me to many experts in optical networking and provided me with many useful suggestions on exploring unknown ideas. Working with her through my PhD journey is one of the most memorable experiences of my life.

I am very fortunate to have worked with David Walker. I specially thank Dave for his guidance on the CoVisor project. We started from a small idea of developing an efficient compilation algorithm for Frenetic, which is a domain-specific programming language for software-defined networks that Dave and Jen were working on. Dave encouraged me to continue exploring this idea and brainstormed with me to develop this small idea to a big system. I also thank Jennifer Gossels for her help on the design and evaluation of CoVisor.

I benefited immensely from my internship at Microsoft Research where I completed the Dionysus project. I would like to thank Ratul Mahajan, Ming Zhang, Srikanth Kandula and Jitu Padhye for being my mentors. Ratul greatly influenced me on his tremendous research passion and his rigorous research approach. He was also a great writer, and gave me enormous help on improving my writing skills. Ming taught me the importance for research projects to have real-world impact and provided me with many opportunities to interact with engineers at Microsoft Azure. I also thank Hongqiang Liu, Rohan Gandhi and Roger Wattenhofer for working with me on the Dionysus project. I also learned a lot

from other people at Microsoft, including Lihua Yuan, Chuanxiong Guo, Guohan Lu, Chao Zhang, George Chen, and David Maltz.

I would like to express my special thanks to Wei Xu for his help on the Owan project. His sharp intellect and rich systems experience made our collaboration a truly enjoyable and rewarding experience. Wei was also very generous to let me work together with his students and use his optical network testbed. I would also like to thank Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu and Guangzhi Li for their contributions to the Owan project.

I would like to thank Ratul Mahajan, David Walker, Nick Feamster, and Aarti Gupta to serve on my committee and provide feedback on this dissertation.

I enjoyed my time at Princeton. I thank current and previous members of the Cabernet research group: Nanxi Kang, Peng Sun, Xuan Zou, Laurent Vanbever, Joshua Reich, Zhenming Liu, Theo Benson, Praveen Naga Katta, Srinivas Narayana, Jennifer Gossels, Mojgan Ghasemi, Mina Tahmasbi Arashloo, Masaharu Morimoto, Ronaldo Ferreira, Minlan Yu, Changhoon Kim, Eric Keller, Wenjie Jiang, and Yaping Zhu. I thank Kai Li and Michael J. Freedman for providing suggestions and guidance to me over the years. I thank my other friends at Princeton for all the delicious food and fun social events we had together: Xiaozhou Li, Haoyu Zhang, Linpeng Tang, Feng Liu, Yichen Chen, Yingda Zhang, Xinyi Fan, Fisher Yu, Haipeng Luo, Guangda Hu, Yida Wang, Wei Dong, Zhe Wang, Jude Nelson, Matvey Arye, Yimin Liu, Bo Guo, Jie Feng, Yaosheng Fu, Changle Lin, Xinwo Huang, Yun Wang, Sen Tao, and Peng Peng.

This dissertation work is supported by DARPA N66001-11-2-4206, NSF grants CNS-1162112 and CNS-1247764, and Intel award AGMT dtd 09-09-09.

Above all, I would like to thank my parents, Jianwei Jin and Jiangping Li, for their enduring love and support. I dedicate this dissertation to them.

To my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Network Management

Computer networks play a critical role in modern society. Today, a lot of Internet services, such as search engines, social networks, and e-commerce, are hosted in data centers, where hundreds of thousands of computers are connected by large-scale data center networks. These data centers are interconnected with each other by wide area networks that span the entire planet. End users use their personal computers, mobiles phones and tablets to access these Internet services via Ethernets, WiFi networks, and cellular networks. Managing these networks to provide fast, reliable and secure network services is a central problem for computer networking research.

Network management includes many different tasks. Network operators configure network devices, e.g., switches and routers, to realize these tasks. The packet processing in network devices can be modeled as match-action processing where the network devices match on certain patterns of packet headers (e.g., destination IP address belongs to an IP prefix) and perform some actions (e.g., drop packets or forward packets to an output port) on the matched packets. We refer to the forwarding behavior of a switch as a *policy* of the switch. Similarly, we refer to the network-wide forwarding behavior as a *policy* of the

network, which is built from policies of all switches in the network. Policies change over time, because operators need to reconfigure network devices in face of various network events, such as traffic shifts, cyber attacks, device failures, host mobility, etc. Here, we use some concrete examples to illustrate network management tasks.

Routing: Routing is the most basic functionality of a network. The goal of routing is to deliver packets from one host to another. The header of a packet contains the source address and the destination address. Switches and routers match on these addresses and forward packets to their destinations. For example, in Ethernets, packets are forwarded based on destination MAC address. The following rule in an Ethernet switch indicates that packets with destination MAC address matching 01:00:00:00:00 are forwarded to port 2 of the switch.

match: dstMAC=01:00:00:00:00 *action:* fwd(2)

In IP networks, packets are forwarded based on destination IP address. The following rule in an IP router indicates that packets with destination IP address matching 1.0.0.0/24 are forwarded to port 5 of the router. This rule uses prefix matching. An IP address matches this rule if it has the same most significant 24 bits as 1.0.0.0.

match: dstIP=1.0.0.0/24 *action:* fwd(5)

Monitoring: Monitoring collects traffic statistics from network devices. Operators use these traffic statistics to identify network bottlenecks, debug network failures, optimize network routing, detect cyber attacks, etc. Operators need to configure switches and routers to collect these traffic statistics. For example, an operator may want to count all the web traffic in the network. To do so, the operator configures switches and routers in the network to count packets with protocol number matching 6 (protocol number 6 indicates TCP protocol) and TCP port matching 80 (port 80 is typically used by HTTP servers). We need the following two rules, one matching on source port and the other matching on destination port.

match: protocol=6, srcPort=80 *action:* count

match: protocol=6, dstPort=80 *action:* count

Server load balancing: Many Internet services today serve a large number of clients. In order to efficiently handle the clients, these services run as distributed systems on many servers. The network provides load balancing to spread traffic from clients among these servers. Clients simply use an anycast IP address to access a service without specifying which server to serve each request. The network divides traffic from clients among the servers, rewrites the destination IP address of each packet from the anycast IP address to a server IP address, and forwards packets to the corresponding server. For example, suppose a service with anycast IP address 1.2.3.4 is hosted on two servers, A and B. Server A has IP address 2.0.0.1, and server B has IP address 2.0.0.2. Suppose server A can be reached via port 1 of a network device and server B can be reached via port 2 of the device. To spread packets with destination IP address matching 1.2.3.4 between A and B, the operator can configure the device to send packets with source IP address matching 0.0.0.0/1 to server A with the following rule.

match: srcIP=0.0.0.0/1, dstIP=1.2.3.4 *action:* dstIP=2.0.0.1, fwd(1)

And the operator can send the other half of packets, i.e., packets with source IP address matching 128.0.0.0/1, to server B, with the following rule.

match: srcIP=128.0.0.0/1, dstIP=1.2.3.4 *action:* dstIP=2.0.0.2, fwd(2)

Firewall: Security is a big concern on today's Internet. Firewalls are widely deployed by operators to protect their networks. Operators configure firewalls to control the incoming and outgoing traffic of a network. For example, an operator may allow all web traffic (e.g., allow packets with TCP port matching 80) with the following two rules.

match: protocol=6, srcPort=80 *action:* permit

match: protocol=6, dstPort=80 *action:* permit

The operator can block all SSH traffic (e.g., drop packets with TCP port matching 22) with the following two rules.

match: protocol=6, srcPort=22, *action:* drop

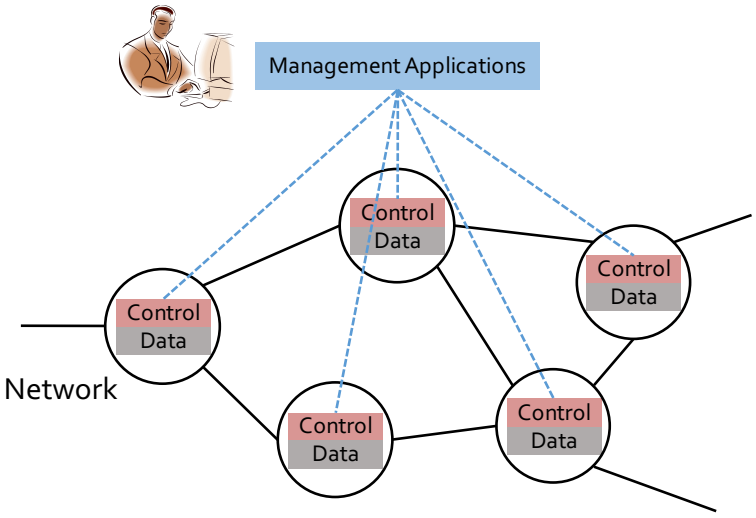


Figure 1.1: Today's network management.

match: protocol=6, dstPort=22, *action:* drop

1.2 Problems with Today's Network Management

Network management can be divided into two planes, i.e., control plane and data plane. The control plane makes packet forwarding decisions; the data plane forwards packets at high speed based on the configuration from the control plane. It is challenging to correctly and effectively implement management tasks on today's network. Operators spend tremendous effort and time on configuring network devices. Specifically, today's network management has the following problems.

Coupled control plane and data plane: In today's network, the control plane is coupled with the data plane, as shown in Figure 1.1. The control plane on each device exchanges information with each other, decides how the packets should be processed on the device, and configures the data plane. Since the control plane is distributed on the devices, it does not have a global view of the network and cannot make good network-wide decisions.

Closed, proprietary interface: The interface between the control plane and the data plane is closed and proprietary. Device vendors sell monolithic boxes that contain both the control plane and the data plane in a single box. Operators cannot only change the control plane or the data plane without changing the other. The closed interface also impedes innovations, since operators and third parties cannot easily develop new functionalities without being restricted by device vendors.

Heterogeneous devices and per-device configurations: There are different devices in the network performing different tasks. For example, switches forward packets based on MAC address, routers forward packets based on IP address, firewalls filter packets based on five tuple (i.e., source and destination IP addresses, source and destination port numbers, protocol number), and load balancers spread packets based on source and destination IP addresses. Although the packet processing on these devices can all be modeled as general match-action style processing, vendors implement different data planes and control planes for them. Each type of device has its own configuration interface, and the interface varies from vendor to vendor. Because of this, operators have to do per-device configurations and have to carefully plan the configurations across different devices to correctly implement network-wide policies.

These problems make it complicated to implement network management tasks, like the ones we describe in §1.1. For the routing task, operates have to configure switches for layer 2 routing and configure routers for layer 3 routing. Switches and routers only allow operators to use certain protocols, e.g., Spanning Tree Protocol (STP) for layer 2 routing, and Open Shortest Path First protocol (OSPF) for layer 3 routing. Operators cannot flexibly customize the control plane for new routing protocols. Similarly, operators have to use specialized devices for the load balancing task and the firewall task. To collect traffic statistics for the monitoring task, operators have to deal with different interfaces exposed by the devices. All the configurations are done on a per-device basis. It is a headache for operators to reason about the interactions between different devices in the network.

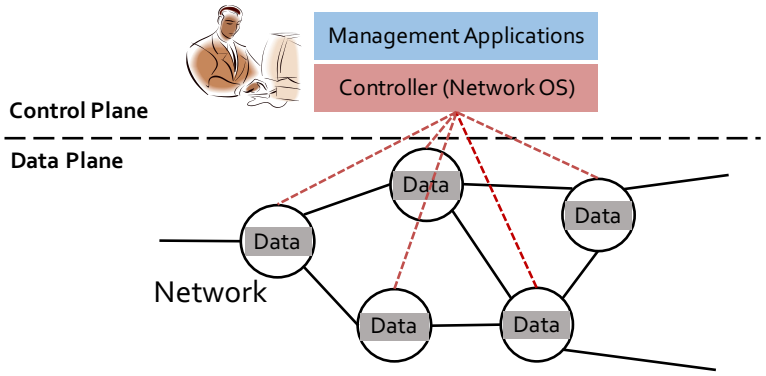


Figure 1.2: Network management with Software-Defined Networking (SDN).

1.3 The Rise of Software-Defined Networking

Software-Defined Networking (SDN) emerged in recent years to fundamentally change how we design, build and manage networks. It has the following distinct features from today's network architecture.

Decoupled control plane and data plane: SDN decouples the control plane from the data plane, as shown in Figure 1.2. The control plane is a logically centralized controller. It gathers information from the data plane and provides a global view to the operator. Management tasks are implemented as applications running on top of the controller. These applications make packet processing decisions based on the global view and distribute the decisions to the data plane via the controller.

Open, standard interface: The interface between the control plane and the data plane is open and standard (e.g., OpenFlow [91]). Device vendors only sell data plane devices without coupling them with the control plane. The devices can be hardware switches made with ASICs or FPGAs, or software switches running on servers. Software engineers can easily develop controllers and applications with different functionalities. Operators are able to mix and match devices, controllers and applications that can best meet their needs.

General packet processing model and unified configuration: SDN models network devices as general packet processing devices using match-action tables, regardless of whether a device acts as a layer 2 switch, a layer 3 router, a load balancer or a firewall.

Priority	Match	Action
2	$dstip = 1.0.0.0/24$	$fwd(3)$
1	$dstip = 1.0.0.0/16$	$fwd(2)$
0	$*$	$drop$

Table 1.1: Example flow table for a routing policy.

A match-action table contains a list of rules. Each rule has multiple components. The most important components are priority, match and action. The match component specifies the header pattern of packets, the action component specifies the processing of packets, and the priority specifies the order when a packet matches multiple rules. For example, Table 1.1 shows a match-action table for a routing policy that forwards packets based on destination IP address. SDN provides a unified interface for the control plane to configure the data plane to implement different management tasks. To simplify the presentation, we will call network devices *switches* in the rest of this thesis.

SDN simplifies the design and deployment of network management tasks. To perform the tasks we describe in §1.1, operators only need to install management applications on the controller. The routing application can use custom routing algorithms based on the global view provided by the control plane and can easily make packet forwarding decisions based on different header fields. Operators do not have to worry about whether the data plane only supports layer 2 routing or layer 3 routing. Firewall and load balancing do not have to use specialized components. They are implemented the same way as routing and can be deployed on any switches in the network. For the monitoring task, operators can collect traffic statistics from different points in the network and obtain network-wide traffic information.

1.4 Challenges of Control Platform Design

SDN is a new architecture for network management. Researchers have shown the benefits of SDN by designing various management applications for routing, monitoring, load bal-

ancing, firewall, and energy saving [55, 58, 63, 93, 123, 131, 101, 44, 94, 106, 53]. The control platform is critical to support these applications and fully realize the benefits of SDN. This section details some challenges involved in the design of the control platform.

1.4.1 Many Controller Applications

Composing multiple controller applications: Operators deploy many applications on the control platform to manage the network. Routing, monitoring, load balancing and firewall are a few example applications as described in §1.1. Each application needs to configure the data plane with a policy to realize its management objective. The control platform should provide support for operators to specify the relationship between multiple applications, and correctly compile policies from multiple applications into a single policy based on this specification. Without support from the control platform, applications have to handle the coordination with others by themselves, which not only complicates application logic but also puts a heavy burden on developers.

Defending against malicious and buggy applications: Since management applications can be from a third party, operators do not fully trust these applications. These applications can have malicious behaviors and process packets in a way they are not supposed to. It is also common for applications to contain bugs that can cause unexpected outcomes during runtime. Therefore, the control platform should allow operators to impose fine-grained access control on what each application can see and do, in order to protect the network from malicious and buggy applications.

Providing flexibility in choosing programming languages for development: Existing controllers like ONOS [99], OpenDaylight [100], Ryu [114], Floodlight [37] and POX [105] constrain developers to use the same language as the controller to develop applications. Ideally, the control platform should provide the flexibility for developers to choose their favorite programming languages, without being restricted to a specific programming language.

1.4.2 Many Network Events

Networks are full of dynamics. Example network events include traffic shifts, cyber attacks, device failures, device upgrades, host mobility, and server overloads. It is important for the control plane to quickly and effectively adapt to these events. Otherwise, these events can cause significant loss to operators. For example, traffic shifts can create congestion in the network and degrade user experience, and cyber attacks can turn down Internet services and cause data leakage. To react to these events, applications compute new policies and update the data plane to the new policies. Rather than having each application to update its policy in an ad-hoc way, the control platform should provide a general and efficient solution. There are mainly the following three problems for handling policy updates.

Composing updates from multiple applications: Each application generates an update for its own policy. Since we have multiple applications deployed in the network, the control plane needs to compose updates from multiple applications into a single update for the network. A straightforward solution is to recompute the network policy from the updated policies of applications, and then install the new policy to the network. However, such a solution incurs high computation overhead since the entire policy has to be recomputed while most parts of the policy may stay unchanged. Furthermore, simply recomputing a new policy without concerning the existing policy may cause unnecessary changes to rules that are already in the switch. We would have to update many existing rules besides adding new rules and deleting deprecated rules. To solve the problem, we need new algorithms that can compute the new network policy incrementally based on the existing policy and make as few changes to the existing policy as possible.

Distributing updates across multiple switches: A policy update often affects multiple switches in the network. Updates to different switches are dependent on each other, because reconfiguring the policy of one switch could affect the traffic to other switches. If the update is not executed carefully, serious problems like routing loops, blackholes, policy violations, and congestion can happen during the update period. Furthermore, because of

the disparities on switch hardware and CPU load, update operations on different switches are highly variable. This can introduce long delays for the control plane to complete an update for the entire network. The control plane should carefully schedule updates in order to eliminate undesirable transient problems and minimize update time.

Coordinating updates between multiple layers: Some network operators, like ISPs, not only control switches in the network layer, but also controls optical devices in the optical layer. Similar to updates to multiple switches, updates to different layers also have to be carefully coordinated, in order to eliminate transient problems. For example, if the operator turns down an optical circuit and establishes a new one in the optical layer without moving away traffic in the network layer first, then during the optical circuit update, all traffic that uses this circuit would be dropped. The control plane should carefully coordinate updates between multiple layers to avoid these problems.

1.4.3 Many Layers

Network management involves both the management of switches in the network layer and optical devices in the optical layer. Existing control platforms focus on the packet processing in the network layer where the devices perform match-action style processing on the traffic [58, 55, 121]. However, optical devices are widely deployed in wide area networks today, and there is a growing interest in integrating optical technologies into data centers for cost, performance and energy benefits [35, 129, 25, 50, 26]. ISPs have separate teams to manage optical devices in the optical layer and switches in the network layer. Optical devices are not frequently reconfigured together with switches in the network layer to optimize data transfers. There is a lack of support in the control platform for the joint management of the optical layer and the network layer.

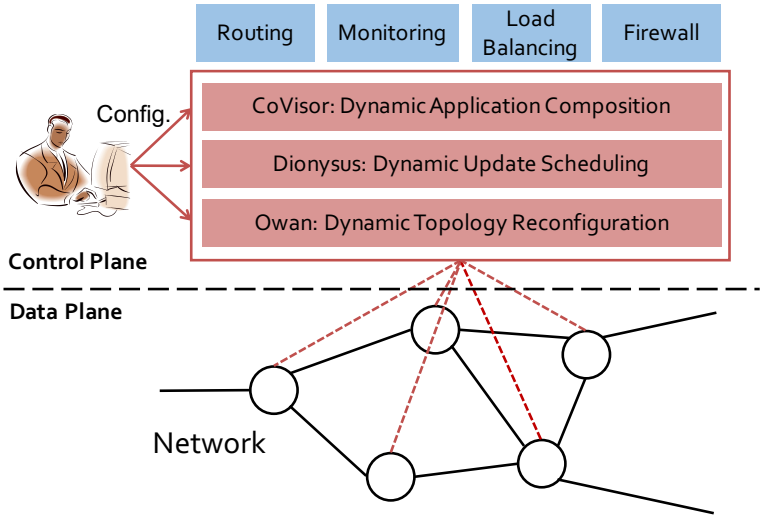


Figure 1.3: Overview of a dynamic network control architecture.

1.5 Contributions

In this thesis, we propose a new network control architecture to efficiently handle dynamics across multiple applications and layers. Figure 1.3 shows an overview of this architecture. We have identified and studied three key components of this architecture: CoVisor is a network hypervisor to efficiently compose multiple applications and handle policy updates from applications; Dionysus is an update scheduler to quickly and consistently distribute policy updates to multiple switches; Owan is a traffic management system to support joint control of the optical and network layers. We design an API for operators to configure each component. Operators use this API to specify how to compose multiple applications, what consistency property to maintain for network updates, and what objective to achieve when jointly managing the optical and network layers. We have designed efficient algorithms to optimize system performance, built software controllers and hardware testbeds, and evaluated them with data from real networks. Now we describe each component in detail.

CoVisor [62]: A network hypervisor for composition of multiple management applications. CoVisor is a new kind of network hypervisor that enables the deployment of

multiple management applications that can be written in different programming languages and run on different controllers. Existing hypervisors focus on slicing, which divides the network into disjoint parts for separate control by separate controllers [5, 120]. Different from them, CoVisor enables multiple controllers to collaborate on processing the same shared traffic. CoVisor provides a composition interface for operators to specify the relationship between applications. Besides composition, CoVisor also virtualizes the physical topology to expose a custom virtual topology to each application, and allows operators to impose fine-grained access control on applications. We have designed a new set of efficient algorithms for composing application policies, for compiling policies from virtual networks into physical networks, and for efficiently processing application policy updates. We have built a CoVisor prototype, and shown that it is several orders of magnitude faster than a naive implementation.

Dionysus [65]: A network update scheduler for fast and consistent network updates. Management applications frequently update their policies in face of various network events. Operators have to carefully perform network updates in order to eliminate undesirable behaviors like loops, blackholes, policy violations and congestion that may happen during the transient period. Previous methods for consistent network updates are slow because they are based on static ordering of rule updates, and ignore the variations in the update times of individual update operations [55, 85, 112, 72]. Dionysus is a network update scheduler that adapts to runtime conditions. Dionysus encodes as a graph the consistency-related dependencies among updates at individual switches, and it then dynamically schedules these updates based on runtime differences in the update speeds of different switches. We have built a prototype of Dionysus. Testbed experiments and data-driven simulations show that Dionysus improves the median update speed by 53–88% in both wide area networks and data center networks as compared to prior methods.

Owan [64]: A traffic management system for joint control of the optical and network layers. Existing network control platforms focus on managing switches in the net-

work layer [58, 55, 121]. But optical devices are widely deployed in wide area networks, and they are a promising technology for future data centers because of benefits on performance, cost and power consumption [35, 129, 25, 50, 26]. We design a new control platform that supports the joint management of switches in the network layer and optical devices in the optical layer. To show the benefits of joint control, we have designed a new application built on top of the control platform that optimizes bulk data transfers over the wide area network. We instantiate our design in a system called Owan that jointly optimizes optical circuit setup, routing and rate allocation for bulk transfers, and dynamically adapts them to workload changes. We have built a prototype of Owan with commodity optical and electrical hardware. Testbed experiments and large-scale simulations on topologies and traffic from production networks show that Owan completes bulk transfers up to $4.45\times$ faster on average, and up to $1.36\times$ more transfers meet their deadlines, as compared to prior methods that only control the network layer.

The rest of the thesis is organized as follows. Chapter 2 describes CoVisor, the network hypervisor for management application compositions. Chapter 3 describes Dionysus, the network update scheduler for fast and consistent network updates. Chapter 4 describes Owan, the traffic management system for joint control of the optical and network layers. Chapter 5 discusses open issues and future work, and concludes this thesis.

Chapter 2

CoVisor: Dynamic Application Composition

This chapter focuses on supporting the composition of multiple management applications and compiling policies from multiple applications into a single policy. To fully realize the vision of SDN, operators should be able to assemble a collection of independently-developed "best of breed" applications written in different programming languages and operating on different controllers. While network hypervisors are able to host multiple controllers on the same network, existing hypervisors only support *slicing*, which divides the network into disjoint parts for separate control by separate controllers. In this chapter, we present CoVisor, a new kind of network hypervisor that allows multiple controllers to *cooperate* on managing the same shared traffic. Consequently, network administrators can use CoVisor to compose a collection of applications—a firewall, a load balancer, a gateway, a router, a traffic monitor—and can apply those applications in combination, or separately, to the desired traffic. CoVisor also abstracts concrete topologies, providing custom virtual topologies in their place, and allows operators to specify access controls that regulate the packets a given application may see, modify, monitor, or reroute. The central technical contribution of CoVisor is a new set of efficient algorithms for composing controller policies,

for compiling virtual networks into concrete OpenFlow rules, and for efficiently processing controller rule updates. We have built a CoVisor prototype, and shown that it is several orders of magnitude faster than a naive implementation.

2.1 Introduction

A foundational principle of Software-Defined Networking (SDN) is to decouple control logic from vendor-specific hardware. Such a separation allows network operators to deploy both the software and the hardware most suited to their needs, rather than being forced to compromise on one or both fronts because of the lack of availability of the perfect box. To fully realize this vision of freely assembling "best of breed" solutions, operators should be able to run any combination of controller applications on their networks. If the optimal monitoring application is written in Python on Ryu [114] and the best routing application is written in Java on Floodlight [37], the operator should be able to deploy both of them in the network.

A network hypervisor is a natural solution to this problem of bringing together disparate controllers. However, existing hypervisors [5, 120] restrict each controller to a distinct *slice* of network traffic. While useful in scenarios like multi-tenancy in which each tenant controls its own traffic, they do not enable multiple applications to collaboratively process the same traffic. Thus, an SDN hypervisor must be capable of more than just slicing. More specifically, in this chapter, we show how to bring together the following key hypervisor features and implement them *efficiently* in a single, coherent system.

(1) Assembly of multiple controllers. A network operator should be able to assemble multiple controllers in a flexible and configurable manner. Inspired by network programming languages like Frenetic [39], we compose data plane policies in three ways: *in parallel* (allow multiple controllers to act independently on the same packets at the same time), *sequentially* (allow one controller to process certain traffic before another), and *by over-*

riding (allow one controller to choose to act or to defer control to another controller). However, unlike Frenetic and related systems, our hypervisor is independent of the specific languages, libraries, or controller platforms used to construct client applications. Instead, the hypervisor intercepts and processes industry-standard OpenFlow messages, assembling and transforming them to match operator-specified composition policies. Doing so efficiently requires new incremental algorithms for processing rule updates.

(2) Definition of abstract topologies. To protect the physical infrastructure, an operator should be able to limit what each controller can see of the physical topology. Our hypervisor supports this by allowing the operator to provide a custom virtual topology to each controller, thereby facilitating reuse of (physical) topology-independent code. For example, to a firewall controller the operator may abstract the network as a "big virtual switch"; the firewall does not need to know the underlying topology to determine if a packet should be forwarded or dropped. In contrast, a routing controller needs the exact topology to perform its task effectively. In addition, topology abstraction helps the operator implement complex functionality in a modular manner. Some switches, such as a gateway between an Ethernet island and the IP core, may play multiple roles in the network. The hypervisor can create one virtual switch for each role, assign each to a controller application precisely tailored to its single task, and compile policies written for the virtual network into the physical network.

(3) Protection against misbehaving controllers. In addition to restricting what a controller can see of the physical topology, an operator may also want to impose fine-grained control on how a controller can process packets. This access control is important to protect against buggy or maliciously misbehaving third-party controllers. For example, a firewall controller should not be allowed to modify packets, and a MAC learner should not be able to inspect IP or TCP headers. The hypervisor enforces these restrictions by limiting the functionality of the virtual switches exposed to each controller.

The primary technical challenge surrounding the creation of such a fully featured hypervisor is efficiency. The hypervisor must host tens of controllers, each of which installs tens of thousands of rules. Complicating matters further, these controllers are updating rules constantly, as dictated by their application logic (e.g., traffic engineering, failure recovery, and attack detection [58, 55, 106, 93, 131, 53, 4, 15, 31]). The naive hypervisor design is to recompile the composed policy from scratch for every rule update, and then install in each switch's flow table the difference between the existing and updated policies. This strawman solution is prohibitively expensive in terms of both the time to compile the new policy and the time to install new rules on switches.

In this chapter we present CoVisor, a hypervisor that exploits efficient new algorithms to compile and update policies from upstream controllers, each of which has its own view of the network topology. Figure 2.1 illustrates the CoVisor architecture. CoVisor serves as a transparent layer between controllers and the physical network. Each of the five applications shown at the top of Figure 2.1 is an unmodified SDN program running on its own controller; each controller outputs OpenFlow rules for the virtual topology shown below it, without any knowledge that this virtual topology does not physically exist. CoVisor intercepts the OpenFlow rules output by all five controllers and compiles them into a single policy for the physical network via a two-phase process.

First, CoVisor uses a novel algorithm to incrementally compose applications in the manner specified by the operator. The key insight is that rule priorities form a convenient algebra to calculate priorities for new rules, obviating the need to recompile from scratch for every rule update. Second, CoVisor translates the composed policy into rules for the physical topology. Specifically, we develop a new compilation algorithm for the case of one physical switch mapped to multiple virtual switches. At both stages, CoVisor employs efficient data structures to further reduce compilation overhead by exploiting knowledge of the structure of policies provided by the access-control restrictions. After compiling the policy, CoVisor sends the necessary rule updates to switches.

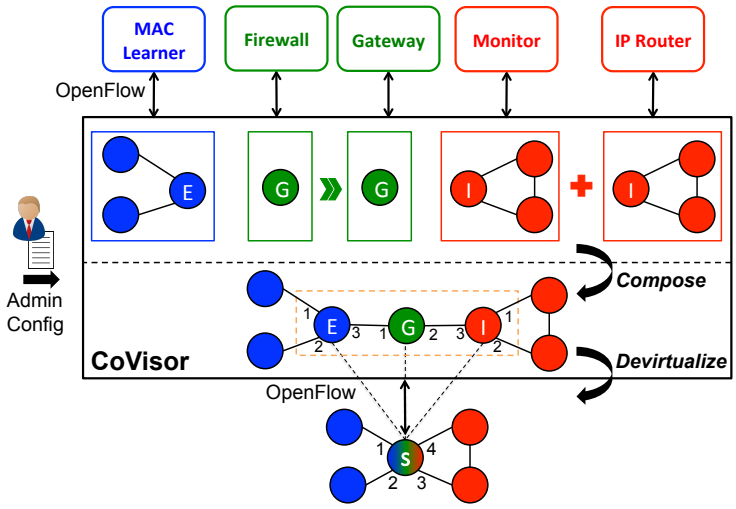


Figure 2.1: CoVisor overview.

At the far left of Figure 2.1, CoVisor takes configuration input from the operator. These configuration responsibilities are threefold: (1) define how the policies of the controllers should be assembled; (2) create each controller's virtual network by specifying the components to be included and the physical-virtual mapping; and (3) state access control limitations for each controller.

In summary, we make the following contributions.

- We define the architecture of a new kind of compositional hypervisor, which allows applications written in different languages and on different controllers to process packets collaboratively.
- We develop a new algorithm to compile the parallel, sequential, and override operators introduced in earlier work [39, 48, 133] incrementally (§2.3).
- We develop a new, incremental algorithm to compile policies written for virtual topologies into rules for physical switches (§2.4).
- We employ customized data structures that leverage access-control restrictions, often a source of overhead, to further reduce compilation time (§2.5).

We describe our prototype in §2.6 and evaluation in §2.7. We have a brief discussion in §2.8, followed by related work in §2.9 and the conclusion in §2.10.

2.2 CoVisor Overview

This section provides an overview of CoVisor. CoVisor's features fall into two categories: (i) those that combine applications running on multiple controllers to produce a single flow table for each physical switch (§2.2.1); and (ii) those that limit an individual controller's view of the topology and packet-processing capabilities (§2.2.2).

To implement these features, CoVisor relies on a two-phase compilation process. The first phase assembles the policies of individual controllers, written for their own virtual networks, into a composed policy for the whole virtual network. The second phase compiles this composed policy for the virtual network into a policy for the physical network that realizes the intent expressed by the virtual policy. Algorithms for these phases are covered in §2.3 and §2.4, respectively.

2.2.1 Composition of Multiple Controllers

CoVisor allows network operators to combine the packet-processing specifications of multiple controllers into a single specification for the physical network. We call these "packet-processing specifications" output by each controller *member policies* and the single specification a *composed policy*. In practice, the member policies are defined by OpenFlow commands issued from a controller to CoVisor. We use the terms *policy implementation* or just *implementation* to refer specifically to the list of OpenFlow rules used to express a policy.

The network operator configures CoVisor to compose controllers with a simple language of commands. Let T range over policies defined in the command language. This language allows operators to specify that some default action (a) should be applied to a set

of packets, that a particular member policy (x) should be applied, that two separate policies should be applied in parallel ($T_1 + T_2$), that two separate policies should be applied in sequence ($T_1 \gg T_2$), or that one member policy should be applied, and if it fails to match a packet, some other policy should act as a default ($x \rhd T$). The following paragraphs explain these policies in greater detail.

Action (a): The most basic composed policy is an atomic packet-processing action a. Such actions include any function from a packet to a set of packets implementable in OpenFlow, such as the actions to drop a packet ($drop$), to forward a packet out a particular port ($fwd(3)$), or to send a packet to the controller ($to\_controller(x)$).

Parallel operator ($+$): The parallel composition of two policies $T_1 + T_2$ operates by logically (though not necessarily physically) copying the packet, applying T_1 to one copy and T_2 to the other, and taking the union of the results. For example, let M be a monitoring policy and Q be a routing policy. If M counts packets based on source IP prefix and Q forwards packets based on destination IP prefix, $M+Q$ does both operations on all packets.

Sequential operator ($\gg$): The sequential operator enables two controllers to process traffic one after another. For example, let L be a load-balancing policy, and let Q be a routing policy. More specifically, for packets destined to anycast IP address 3.0.0.0, L rewrites the destination IP to a server replica's IP based on source IP prefix, and Q forwards packets based on destination IP prefix. To obtain the combined behavior of L and Q—to first rewrite the destination IP address and then forward the rewritten packet to the correct place—the network operator uses the policy $L \gg Q$.

Override operator ($\rhd$): Each controller x provides CoVisor with a member policy specifying how x wants the network to process packets. The policy $x \rhd T$ attempts to apply x's member policy to any incoming packet t. If x's policy does not specify how to handle t, then T is used as a default. For example, suppose one controller x is running an elephant flow routing application and another controller y is running an infrastructure routing appli-

Command	Parameters
createVSw	pSw_1 <pSw_2, ..., pSw_n>
createVPort	vSw <pSw pPort>
createVLink	vSw_1 $vPort_1$ vSw_2 $vPort_2$
connectHost	vSw vPort host

Table 2.1: API to construct a virtual network. Brackets <> indicate optional arguments.

```
E = createVSw S              // vswitch E
G = createVSw S              // vswitch G
I = createVSw S              // vswitch I
E1 = createVPort E S 1       // port 1 on E
E2 = createVPort E S 2       // port 2 on E
E3 = createVPort E           // port 3 on E
G1 = createVPort G           // port 1 on G
L1 = createVLink E 3 G 1     // link E − G
...      remaining commands omitted for brevity.
```

Figure 2.2: Administrator configuration to create (a subset of) the physical-virtual mapping shown in Figure 2.1.

cation. If we want x to override y for elephant flow packets, y to route all regular traffic, and any packet not covered by either policy to be dropped, we use the policy $x \rhd (y \rhd drop)$.

2.2.2 Constraints on Individual Controllers

In addition to composing member policies, CoVisor allows the operator to virtualize the underlying topology and restrict the packet-processing capabilities available to each controller. This helps operators hide infrastructure information from third-party controllers, reuse topology-independent algorithms, and provide security against malicious or buggy control software.

Constraints on Topology Visibility

Rather than exposing the full details of the physical topology to each controller, CoVisor provides each with its own virtual topology. Table 2.1 shows the API to construct a custom virtual network. `createVSw` creates a virtual switch. It can be used to create two kinds of physical-virtual mappings as follows. (1) *many-to-one* (many physical switches map

to one virtual switch): call the function once with a list of physical switch identifiers; (2) *one-to-many* (a single physical switch maps to many virtual switches): call the function multiple times with the same physical switch identifier. `createVPort` creates a virtual port. To map it to a physical port, the operator includes the corresponding physical switch and port number. `createVLink` creates a virtual link by connecting two virtual ports. `connectHost` connects a host to a virtual port.

Example. Consider the example physical-virtual topology mapping shown in Figure 2.1. The physical topology represents an enterprise network consisting of an Ethernet island (shown in blue in Figure 2.1) connected by a gateway router (multicolored and labeled S) to the IP core (red). We abstract gateway switch S to three virtual switches: E, G, and I. Figure 2.2 shows how the operator uses CoVisor's API to create the virtual mapping.

These four commands allow the operator to create one level of virtual topology on top of a physical network. To create multiple levels of topology abstraction, the operator can run one CoVisor instance on top of another. Supporting this behavior in a single instance of CoVisor is part of our future work.

Constraints on Packet Handling

CoVisor imposes fine-grained access control on how a controller can process packets by virtualizing switch functionality. The operator sets custom capabilities on each controller's virtual switches, thereby choosing which functionalities of the physical network to expose on a controller-by-controller basis.

Pattern: The operator specifies which header fields a controller can match and how each field can be matched (i.e., exact-match, prefix-match, or arbitrary wildcard-match). CoVisor currently supports the 12 fields in the OpenFlow 1.0 specification, with prefix-match an option only for source and destination IP addresses.

Action: The operator specifies the actions a controller can perform on matched packets. CoVisor currently supports the actions in the OpenFlow 1.0 specification, including for-

ward, drop (indicated by an empty action list), and modify (the operator determines which fields can be modified). The operator also controls whether a controller can query packets and counters from switches and send packets to switches.

Example. In the example in Figure 2.1, the operator can restrict the MAC learner to match only on source and destination MAC and inport and the firewall to match only on the five tuple. Also, the operator can disallow both applications from modifying packets.

2.2.3 Handling Failures

Controllers, switches, and CoVisor itself can fail during operation. We describe how CoVisor responds to them.

Controller failure: The operator configures CoVisor with a default policy for each controller to execute in the event of controller failure. The default policy is application-dependent. For example, a logical default for a firewall controller is $drop$ (erase all installed rules and install a rule that drops all packets), because a firewall should fail safe. In contrast, the default policy for a monitoring controller can be id (identical, i.e., leave all rules in the switch), as monitoring rules are not critical to the operation of a network, and the counters can be reused if the monitoring controller recovers.

Switch failure: If a switch fails, all its rules are removed and CoVisor notifies the relevant controllers. Moreover, in the case of many-to-one virtualization, CoVisor allows the virtual switch to remain functional by rerouting traffic around the failed physical switch (if possible in the physical network).

Hypervisor failure: We currently do not deal with hypervisor failure. Replication techniques in distributed systems may be applied to CoVisor, but a full exploration is beyond the scope of this work.

Monitoring M_R
$\big(1;\ srcip = 1.0.0.0/24;\ count\big)$
$\big(0;\ *;\ drop\big)$

Routing Q_R
$\big(1;\ dstip = 2.0.0.1;\ fwd(1)\big)$
$\big(1;\ dstip = 2.0.0.2;\ fwd(2)\big)$
$\big(0;\ *;\ drop\big)$

Load balancing L_R
$\big(3;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1\big)$
$\big(1;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2\big)$
$\big(0;\ *;\ drop\big)$

Elephant flow routing E_R
$\big(1;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3)\big)$

Parallel composition: $comp_{+}(M_R, Q_R)$
$\big(5;\ srcip = 1.0.0.0/24, dstip = 2.0.0.1;\ count, fwd(1)\big)$
$\big(4;\ srcip = 1.0.0.0/24, dstip = 2.0.0.2;\ count, fwd(2)\big)$
$\big(3;\ srcip = 1.0.0.0/24;\ count\big)$
$\big(2;\ dstip = 2.0.0.1;\ fwd(1)\big)$
$\big(1;\ dstip = 2.0.0.2;\ fwd(2)\big)$
$\big(0;\ *;\ drop\big)$

Sequential composition: $comp_{\gg}(L_R, Q_R)$
$\big(2;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1, fwd(1)\big)$
$\big(1;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2, fwd(2)\big)$
$\big(0;\ *;\ drop\big)$

Override composition: $comp_{\triangleright}(E_R, Q_R)$
$\big(3;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3)\big)$
$\big(2;\ dstip = 2.0.0.1;\ fwd(1)\big)$
$\big(1;\ dstip = 2.0.0.2;\ fwd(2)\big)$
$\big(0;\ *;\ drop\big)$

Figure 2.3: Example of policy compilation.

2.3 Incremental Policy Compilation

Network management is a dynamic process. Applications update their policies in response to various network events, like a change in the traffic matrix, switch and link failures, and detection of attacks [58, 55, 106, 93, 131, 53, 4, 15, 31]. Therefore, CoVisor receives streams of member policy updates from controllers and has to recompile and update the composed policy frequently. In this section, we first review policy compilation and introduce a strawman solution, and then we describe an efficient solution based on a convenient algebra on rule priorities.

2.3.1 Background on Policy Compilation

The first stage of policy compilation entails combining member policies into a single composed policy. Controllers implement member policies by sending OpenFlow rules to CoVisor. A rule r is a triple $r = (p; m; a)$ where p is a priority, m is a match pattern, and a is an action list. Given a rule $r = (p; m; a)$, we use the notation $r.priority$ to refer to p, $r.match$ to refer to m, and $r.action$ to refer to a. We denote the set of packets matching $r.match$ as $r.mSet$. Now we describe how to compile each composition operator outlined in §2.2.1. We assume all policy implementations include only OpenFlow 1.0 rules and that each switch has a single flow table.

Parallel operator ($+$): To compile T_1+T_2, we first compile T_1 and T_2 into implementations R_1 and R_2. (In practice, each controller communicates its member policy to CoVisor in an already compiled form. We explicitly include this step because it represents the base case of the recursive process.) Then, we compute $comp_+(R_1, R_2)$ by iterating over $(r_{1i}, r_{2j}) \in R_1 \times R_2$ where r_{1i} and r_{2j} are taken from R_1 and R_2, respectively, by priority in decreasing order. We produce a rule r in the composed implementation if the intersection of $r_{1i}.mSet$ and $r_{2j}.mSet$ is not empty. $r.match$ is the intersection of $r_{1i}.match$ and $r_{2j}.match$, and $r.actions$ is the union of $r_{1i}.actions$ and $r_{2j}.actions$. We defer priority assignment to later

discussion in this subsection. Consider the example of $comp_+(M_R, Q_R)$ in Figure 2.3. Let $M_R = m_1, \ldots, m_n$ and $Q_R = q_1, \ldots, q_k$. We begin by considering m_1 and q_1. Since $m_1.mSet \cap q_1.mSet \neq \emptyset$, we produce a first rule r_1 in $comp_+(M_R, Q_R)$ with match pattern $\{srcip = 1.0.0.0/24, dstip = 2.0.0.1\}$ and action list $\{count, fwd(1)\}$. Composing all (m_i, q_j) pairs gives the composed policy implementation $comp_+(M_R, Q_R)$ of the policy composition $M + Q$.

Sequential operator ($\gg$): To compile $T_1 \gg T_2$, we again begin by generating implementations R_1 and R_2. Then, we compute $comp_{\gg}(R_1, R_2)$. As with $comp_+(R_1, R_2)$, we iterate over $(r_{1i}, r_{2j}) \in R_1 \times R_2$ where r_{1i} and r_{2j} are taken from R_1 and R_2, respectively, by priority in decreasing order. However, now we produce a rule r in the composed policy if the intersection of $r_{2j}.mSet$ and the set of packets produced by applying $r_{1i}.action$ to all packets in $r_{1i}.mSet$ is not empty. Consider the example of $comp_{\gg}(L_R, Q_R)$ in Figure 2.3. Again, we begin iterating over $(l_i, q_j) \in L_R \times Q_R$ pairs by considering l_1 and q_1. Applying $l_1.action$ to all packets in $l_1.mSet$ gives the set of packets matching pattern $\{srcip = 0.0.0.0/2, dstip = 2.0.0.1\}$. The intersection of this set and $q_1.mSet$ is not empty. Hence, we generate the first rule in the composed policy implementation with match pattern $\{srcip = 0.0.0.0/2, dstip = 3.0.0.0\}$ and action list $\{dstip = 2.0.0.1, fwd(1)\}$. Repeating this process for all (l_i, q_j) pairs yields $comp_{\gg}(L_R, Q_R)$, the implementation of $L \gg Q$.

Override operator ($\rhd$): To compile $T_1 \rhd T_2$, we again begin by generating implementations R_1 and R_2. Then, we compute $comp_{\rhd}(R_1, R_2)$ by stacking R_1 on top of R_2 with higher priority. For example in Figure 2.3, to compile $comp_{\rhd}(E_R, Q_R)$, we put E_R's rules above Q_R's rules. Thus, packets with source IP 1.0.0.0 and destination IP 2.0.0.1 will be forwarded to port 3, and other packets with destination IP 2.0.0.1 will be forwarded to port 1.

Priority assignment and policy update problem: Recall that a rule r is a triple $(r.priority; r.match; r.action)$. Thus far, we have explained how to generate a list of

Routing Q_R
$\big(1;\ dstip = 2.0.0.1;\ fwd(1)\big)$
$\big(1;\ dstip = 2.0.0.2;\ fwd(2)\big)$
$\big($**1**; **dstip=2.0.0.3**; **fwd(3)**$\big)$
$\big(0;\ *;\ drop\big)$

Parallel composition: $comp_+(M_R, Q_R)$
$\big($**7**; **srcip=1.0.0.0/24,dstip=2.0.0.1**; **fwd(1),count**$\big)$
$\big($**6**; **srcip=1.0.0.0/24,dstip=2.0.0.2**; **fwd(2),count**$\big)$
$\big($**5**; **srcip=1.0.0.0/24,dstip=2.0.0.3**; **fwd(3),count**$\big)$
$\big($**4**; **srcip=1.0.0.0/24**; **count**$\big)$
$\big($**3**; **dstip=2.0.0.1**; **fwd(1)**$\big)$
$\big($**2**; **dstip=2.0.0.2**; **fwd(2)**$\big)$
$\big($**1**; **dstip=2.0.0.3**; **fwd(3)**$\big)$
$\big(0;\ *;\ drop\big)$

Figure 2.4: Example of updating policy composition. Strawman solution.

$(match; action)$ pairs, or *pseudo-rules*. Our list of pseudo-rules is prioritized in the sense that each pseudo-rule's position indicates its relative priority, but we have not addressed how to assign a particular priority value to each pseudo-rule. Priority assignment is important for minimizing the overhead of policy update. Ideally, a single rule addition in one member policy implementation should not require recomputing the entire composed policy from scratch, nor should it require clearing the physical switch's flow table and installing thousands of flowmods. (A flowmod is an OpenFlow message to update a rule in a switch.) In concrete terms, the update problem involves minimizing the following two overheads:

- **Computation overhead:** The number of rule pairs over which the composition function $comp$ iterates to recompile the composed policy.
- **Rule update overhead:** The number of flowmods needed to update a switch to the new policy.

Strawman solution: The strawman solution is to assign priorities to rules in the composed implementation from bottom to top starting from 0 by increment of 1. Then, it installs the

Monitoring M_R
$(1;\ srcip = 1.0.0.0/24;\ count)$
$(0;\ *;\ drop)$

Routing Q_R
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
(**1**; **dstip=2.0.0.3**; **fwd(3)**)
$(0;\ *;\ drop)$

Load balancing L_R
$(3;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1)$
(**2**; **srcip=0.0.0.0/1,dstip=3.0.0.0**; **dstip=2.0.0.3**)
$(1;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2)$
$(0;\ *;\ drop)$

Elephant flow routing E_R
$(1;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$

Parallel composition: $comp_{+}(M_R, Q_R)$
$(2;\ srcip = 1.0.0.0/24, dstip = 2.0.0.1;\ fwd(1), count)$
$(2;\ srcip = 1.0.0.0/24, dstip = 2.0.0.2;\ fwd(2), count)$
(**2**; **srcip=1.0.0.0/24,dstip=2.0.0.3**; **fwd(3),count**)
$(1;\ srcip = 1.0.0.0/24;\ count)$
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
(**1**; **dstip=2.0.0.3**; **fwd(3)**)
$(0;\ *;\ drop)$

Sequential composition: $comp_{\gg}(L_R, Q_R)$
$(25;\ srcip = 0.0.0.0/2, dstip = 3.0.0.0;\ dstip = 2.0.0.1, fwd(1))$
(**17**; **srcip=0.0.0.0/1,dstip=3.0.0.0**; **dstip=2.0.0.3,fwd(3)**)
$(9;\ dstip = 3.0.0.0;\ dstip = 2.0.0.2, fwd(2))$
$(0;\ *;\ drop)$

Override composition: $comp_{\triangleright}(E_R, Q_R)$
$(9;\ srcip = 1.0.0.0, dstip = 2.0.0.1;\ fwd(3))$
$(1;\ dstip = 2.0.0.1;\ fwd(1))$
$(1;\ dstip = 2.0.0.2;\ fwd(2))$
(**1**; **dstip=2.0.0.3**; **fwd(3)**)
$(0;\ *;\ drop)$

Figure 2.5: Example of incremental update.

difference between the old implementation and the new one. For example, the priorities of rules in Figure 2.3 are assigned in this way. This approach incurs high computation and rule update overhead, because it requires recompiling the whole policy to determine each rule's new relative position and updates rules that only change priorities. For example, when a new rule is inserted to Q_R (in bold in Figure 2.4), although only the third and the seventh rules in $comp_+(M_R, Q_R)$ are new, five rules change their priorities. We have to update these five existing rules as well as add two new rules. Rules in bold in Figure 2.4 count toward this rule update overhead.

2.3.2 Incremental Update

Ideally, the priority of rule r in the composed implementation is a function solely of the rules in the member implementations from which it is generated. In this way, any updates of other rules in member implementations will not affect r. We observe that rule priorities form a convenient algebra which allows us to achieve this goal.

Add for parallel composition: Let R be the composed implementation of $comp_+(R_1, R_2)$. If rule $r_k \in R$ is composed from $r_{1i} \in R_1$ and $r_{2j} \in R_2$, then $r_k.priority$ is the sum of $r_{1i}.priority$ and $r_{2j}.priority$:

$$r_k.priority = r_{1i}.priority + r_{2j}.priority. \tag{2.1}$$

We show the example of $comp_+(M_R, Q_R)$ in Figure 2.5. The first rule in $comp_+(M_R, Q_R)$ is composed from m_1 and q_1. Hence, its priority is $m_1.priority + q_1.priority = 2$. Suppose a new rule (in bold in Figure 2.5) is inserted to Q_R. We only need to iterate over rule pairs (m_i, q_3) for all $m_i \in M_R$, rather than iterate over all the rule pairs. This generates two new rules (in bold in Figure 2.5). All existing rules do not change. Formally, we can prove

that as long as the member policies are not ambiguous[1], the composed policy is also not ambiguous and is correct, as stated in the following lemma.

Lemma 1. *Let p_i be the highest priority rule in P that matches a packet t and q_j be the highest priority rule in Q that matches t. Let r_k be composed from p_i and q_j in $R = P + Q$ with priority calculated by Equation 2.1. Then r_k is the highest priority rule in R that matches t.*

Proof. We prove this by contradiction. Suppose there is a rule $r_{k'} \in R$ that matches t and has a higher priority than r_k, i.e.,

$$r_{k'}.priority > r_k.priority.$$

Let $r_{k'}$ be computed by $p_{i'} \in P$ and $q_{j'} \in Q$. Since $r_{k'}$ matches t, so $p_{i'}$ matches t and $q_{j'}$ matches t. We have the following two equations for their priorities.

$$r_k.priority = p_i.priority + q_j.priority,$$

$$r_{k'}.priority = p_{i'}.priority + q_{j'}.priority.$$

Therefore, we have

$$p_{i'}.priority + q_{j'}.priority > p_i.priority + q_j.priority$$

With this inequality, we can derive that either

$$p_{i'}.priority > p_i.priority \text{ or } q_{j'}.priority > q_j.priority.$$

[1]A policy is called ambiguous if there are two rules that can match the same packet and have the same priority.

But this contradicts the fact that p_i is the highest priority rule in P that matches t and q_j is the highest priority rule in Q that matches t. □

Concatenate for sequential composition: Let R be the composed implementation of $comp_{\gg}(R_1, R_2)$. If $r_k \in R$ is composed from $r_{1i} \in R_1$ and $r_{2j} \in R_2$, then $r_k.priority$ is the concatenation of $r_{1i}.priority$ and $r_{2j}.priority$:

$$r_k.priority = r_{1i}.priority \circ r_{2j}.priority. \tag{2.2}$$

Symbol $\circ$ in Equation 2.2 represents the concatenation of two priorities, where each priority is represented as a fixed-width bit string. Concatenation enforces a *lexicographic ordering* on the pair of priorities. Specifically, let $a_1 = b_1 \circ c_1$ and $a_2 = b_2 \circ c_2$. Then $a_1 > a_2$ if and only if $\big(b_1 > b_2$ or $(b_1 = b_2$ and $c_1 > c_2)\big)$, and $a_1 = a_2$ if and only if $(b_1 = b_2$ and $c_1 = c_2)$. In practice, concatenation is computed as follows. Let r_{2j} be in the range $[0, MAX_{R_2})$ where $MAX_{R_2} - 1$ is the highest priority that R_2 may use[2]. Then $r_k.priority$ is computed by

$$r_k.priority = r_{1i}.priority \times MAX_{R_2} + r_{2j}.priority. \tag{2.3}$$

We show the example of $comp_{\gg}(L_R, Q_R)$ in Figure 2.5. Let $MAX_{Q_R} = 8$. The first rule in $comp_{\gg}(L_R, Q_R)$ is composed from l_1 and q_1. Thus, its priority is $l_1.priority \times 8 + q_1.priority = 25$. Suppose a new rule is inserted to L_R $\big($in bold in Figure 2.5$\big)$. We only need to iterate over rule pairs (l_3, q_j) for all $q_j \in Q_R$. This generates a new rule with priority 17 $\big($in bold in Figure 2.5$\big)$. All existing rules do not change. Similarly, we can prove that as long as the member policies are not ambiguous, the composed policy is also not ambiguous and is correct, as stated in the following lemma.

[2]CoVisor limits the priority space of each member policy, because the bits for priority in hardware are limited in practice.

Lemma 2. *Let p_i be the highest priority rule in P that matches a packet t and q_j be the highest priority rule in Q that matches the packet set after applying p_i to t. Let r_k be composed from p_i and q_j in $R = P \gg Q$ with priority calculated by Equation 2.2. Then r_k is the highest priority rule in R that matches t.*

Proof. We prove this by contradiction. Suppose there is a rule $r_{k'} \in R$ that matches t and has a higher priority than r_k, i.e.,

$$r_{k'}.priority > r_k.priority$$

Let $r_{k'}$ be constructed from $p_{i'} \in P$ and $q_{j'} \in Q$. Since $r_{k'}$ matches t, we know that $p_{i'}$ matches t and $q_{j'}$ matches the packet set after applying $p_{i'}$ to t. We have the following two equations for their priorities.

$$r_k.priority = p_i.priority \circ q_j.priority,$$

$$r_{k'}.priority = p_{i'}.priority \circ q_{j'}.priority.$$

Therefore, we have

$$p_{i'}.priority \circ q_{j'}.priority > p_i.priority \circ q_j.priority$$

With this inequality, we can derive that either

$$p_{i'}.priority > p_i.proirity$$

or

$$p_{i'}.priority = p_i.priority \;\&\&\; q_{j'}.priority > q_j.priority.$$

In the former case, it contradicts the fact that p_i is the highest priority rule in P that matches t. In the latter case, $p_{i'}.priority = p_i.proirity$ implicates $p_{i'} = p_i$ because policies are unambiguous. Then $q_{j'}$ becomes the highest priority rule in Q that matches the packet set after applying p_i to t. This contradicts the fact that q_j be the highest priority rule in Q that matches the packet set after applying p_i to t. □

Stack for override composition: Let R be the composed implementation of $comp_{\triangleright}(R_1, R_2)$, and let R_2's priority space be $[0, MAX_{R_2})$. To assign priorities in R, we increase the priorities of R_1's rules by MAX_{R_2} and keep the priorities of R_2's rules unchanged. This process essentially stacks R_1's priority space on top of R_2's priority space. Specifically, let $r_k \in R$. By definition of $comp_{\triangleright}$, r_k is in either R_1 or R_2. Let $r_k.mPriority$ be r_k's priority in the member implementation from which it comes. We assign priority to r_k as follows.

$$r_k.priority = \begin{cases} r_k.mPriority + MAX_{R_2} & \text{if } r_k \in R_1 \\ r_k.mPriority, & \text{if } r_k \in R_2 \end{cases} \tag{2.4}$$

We show the example of $E_R \triangleright Q_R$ in Figure 2.5. Let $MAX_{Q_R} = 8$. The first rule in $comp_{\triangleright}(E_R, Q_R)$ is generated from e_1, so it is assigned priority $e_1.priority + 8 = 9$. The second rule in $comp_{\triangleright}(E_R, Q_R)$ is generated from q_1, so it is assigned priority $q_1.priority = 1$. When a new rule q_3 that matches $dstip = 1.0.0.3$ is inserted to Q_R, we simply add a new rule with priority 1 (in bold in Figure 2.5) to $comp_{\triangleright}(E_R, Q_R)$ without affecting existing rules. The proof of the correctness for override composition is straightforward.

With the algebra on rule priorities above, CoVisor processes the three kinds of rule updates as follows. Let R be the composed policy implementation of R_1 and R_2.

Rule addition: When a new rule r_1^* is added to R_1 (or r_2^* to R_2), CoVisor composes this rule with each rule in R_2 (or R_1). It assigns priorities to new rules according to Equations 2.1, 2.2, and 2.4 and installs them to switches. All existing rules are untouched.

Algorithm 1 Symbolic path generation

```
1: function GENPATHS(pkt)
2:     pkt.children ← {evaluate policy on pkt}
3:     for all child in pkt.children do
4:         if not child.reachedEgress() then
5:             GENPATHS(child)
```

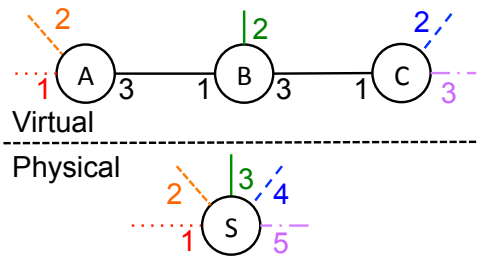
Rule deletion: When an old rule r_1^* is deleted from R_1 (or r_2^* from R_2), CoVisor finds all rules in R that are composed from this rule and deletes them from switches. All other rules are untouched.

Rule modification: Modifying a rule is equivalent to deleting an old rule and then inserting a new rule.

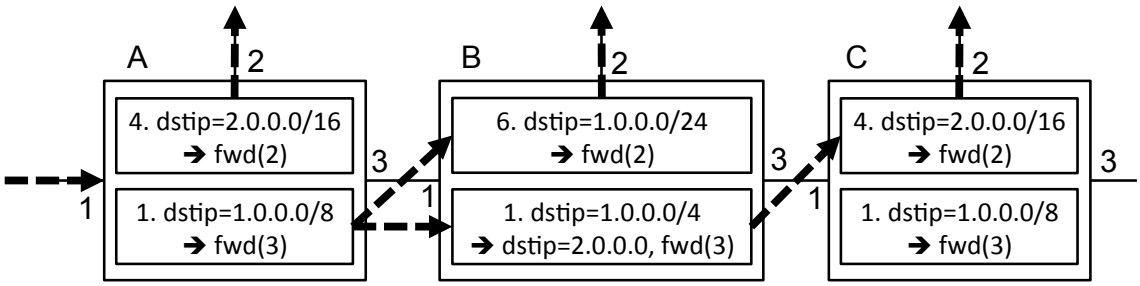
2.4 Compiling Topology Transformations

The first phase of compilation (§2.3) generates a composed policy for the virtual network. The second phase, which we describe in this section, compiles a policy for the virtual topology into one for the physical network. It comprises two sub-cases as described in §2.2.2: many-to-one and one-to-many. One-to-one is a degenerate case of these two. While previous work has explored compilation of the many-to-one case [70, 71], there does not exist any compilation algorithm for the one-to-many case. Pyretic [92] offers the one-to-many feature but implements it by sending the first packet of each flow to the controller and then installing micro-flow rules, a strategy which incurs prohibitive overhead. We present the first compilation algorithm for the one-to-many case.

Our algorithm is a novel combination of symbolic analysis [73] and incremental sequential composition. Intuitively, we inject a symbolic packet into the virtual network, follow all possible paths to egress ports, and sequentially compose the rules along each path. In this way, we derive rules for the physical switch to process traffic as intended by the controller's policy for the virtual network. To handle rule updates incrementally, we keep all the symbolic paths computed during this analysis and minimally modify them



(a) Topology. Colors and line types indicate physical-virtual mapping.



(b) Flow tables of virtual switches.

Figure 2.6: One-to-many virtualization.

when the virtual policy changes. We divide our description into three parts: symbolic path generation (§2.4.1), sequential composition on symbolic paths (§2.4.2), and incremental update (§2.4.3).

2.4.1 Symbolic Path Generation

For each ingress port of the virtual network, we inject a single symbolic packet with wildcards in all fields (except *inport*). At every hop, we evaluate the policy on the packet, which generates zero, one, or more symbolic packets. We follow the generated symbolic packets until they all reach egress ports. Together, these symbolic paths form a tree rooted at the ingress port.

Algorithm 1 shows pseudocode for the path generation algorithm. In Line 2, we create all child packets that can result from evaluating the policy on `pkt`—one child packet for each rule r that `pkt` matches. As we construct the tree, we update `child`'s header, which

denotes the subset of traffic represented by the symbolic packet, according to the information encoded in the rule responsible for generating `child` from `pkt`. By doing so, we avoid creating branches for paths that no packet could possibly follow.

We use the example in Figure 2.6 to illustrate the process. Figure 2.6(a) shows a physical-virtual topology mapping in which a physical switch S is virtualized to three virtual switches, A, B and C. The mapping between physical and virtual ports is color- and line type-coded. Figure 2.6(b) shows the policy of each virtual switch.

We inject a symbolic packet with header $*$, denoting wildcards in all fields, into port 1 of A. When we apply A's policy to this packet, we generate two child symbolic packets, p_1 and p_2. p_1 has destination IP 2.0.0.0/16, matches the first rule in A's policy, A_{R1}, and leaves the network at port 2 of A; p_2 has destination IP 1.0.0.0/8, matches A's second rule, A_{R2}, and reaches port 1 of B. We then evaluate B's policy on p_2, again generating two symbolic packets, p_{21} and p_{22}. p_{21} matches B_{R1} and leaves the network at port 2 of B; p_{22} matches B_{R2}, enters C at port 1, matches C_{R1}, and finally leaves the network at port 2 of C. In total, we get the following three symbolic paths: (1) $p_1 : A_{R1}$; (2) $p_{21} : A_{R2} \rightarrow B_{R1}$; and (3) $p_{22} : A_{R2} \rightarrow B_{R2} \rightarrow C_{R1}$.

2.4.2 Sequential Composition

For each symbolic path, we sequentially compose all the rules along its edges to generate a single rule. Then, we derive a final rule for the physical switch by adding a match on the $inport$ value of the symbolic packet at the root of the tree. Returning to our example in Figure 2.6, the first symbolic path contains only A_{R1}. By adding port 1 to its match, we get the first rule for physical switch S.

$$S_{R1} = \big(4;\ inport = 1, dstip = 2.0.0.0/16;\ fwd(2)\big).$$

Adding $inport = 1$ is necessary because traffic that enters port 3 of C (port 5 of S) with destination IP 2.0.0.0/16 will be forwarded to port 2 of C (port 4 of S). Similarly, for the second and third symbolic paths, we evaluate $comp_{\gg}(A_{R2}, B_{R1})$ and $comp_{\gg}\big(comp_{\gg}(A_{R2}, B_{R2}), C_{R1}\big)$, respectively. We assume the priority space for each switch is $[0, 8)$. After adding ingress port, we obtain two more rules.

$$S_{R2} = \big(14; inport = 1, dstip = 1.0.0.0/24; fwd(3)\big)$$
$$S_{R3} = \big(76; inport = 1, dstip = 1.0.0.0/8; dstip = 2.0.0.0, fwd(4)\big)$$

Priority assignment: Because symbolic paths may have different lengths, for the devirtualization phase of compilation we need to augment the priority assignment algorithm for sequential composition presented in §2.3.2. For example, from sequential composition we get priorities 4, 14, and 76 for rules S_{R1}, S_{R2}, and S_{R3}, respectively. But, with these priorities, traffic entering port 1 at S with source IP 1.0.0.0/24 would match S_{R3} rather than S_{R2}, even though S_{R2} should have a higher priority than S_{R3}. This mismatch happens because S_{R2} is calculated from a path with only two hops (its priority is $1 \circ 6 = 14$) and S_{R3} is calculated from one with three hops ($1 \circ 1 \circ 4 = 76$). To address the mismatch, we set a hop length l^*. If a path is fewer than l^* hops, we pad 0s to the concatenation of the rule priorities. In practice, we use the number of switches in the virtual topology as l^*, as a path will have more than that number of hops only if the virtual policy contains a loop. This modified algorithm correctly orders S_{R2} and S_{R3}, assigning them respective priorities of $1 \circ 6 \circ 0 = 112$ and $4 \circ 0 \circ 0 = 256$. Figure 2.7 shows the rules for S with priorities calculated in this manner. We repeat the above procedure for all ingress ports of the virtual topology to get the final policy for S.

Flow table of S
$\big(256;\ inport = 1, dstip = 2.0.0.0/16;\ fwd(2)\big)$
$\big(112;\ inport = 1, dstip = 1.0.0.0/24;\ fwd(3)\big)$
$\big(76;\ inport = 1, dstip = 1.0.0.0/8;\ dstip = 2.0.0.0, fwd(4)\big)$

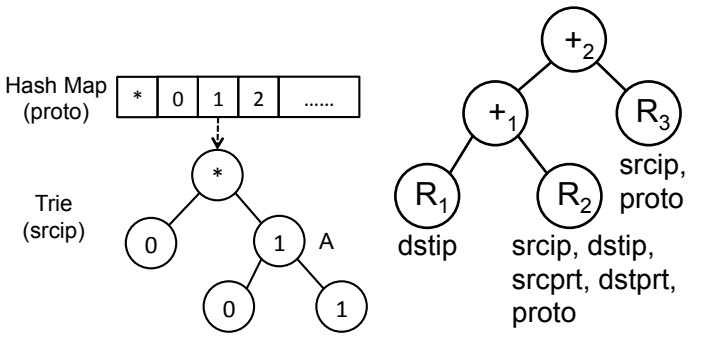
Figure 2.7: Flow table of switch S in Figure 2.6.

2.4.3 Incremental Update

By storing all the symbolic paths we generate when compiling a policy and partially modifying them upon a rule insertion or deletion, we can incrementally update a policy. This strategy obviates the need to compile the whole policy from scratch upon every rule update. In particular, when virtual switch V receives a rule update, we reevaluate V's policy on all symbolic packets that enter V. As a result, we may generate new symbolic packets, which we then follow until they reach egress ports. V's policy update may also modify the headers of or eliminate existing symbolic packets. Accordingly, we update the paths of modified symbolic packets and remove the paths of deleted packets. Then, we add and remove rules from the physical switch as described in §2.4.2. Our priority assignment algorithm ensures that these rule additions and deletions do not affect existing rules generated from symbolic paths that have not changed.

2.5 Exploiting Policy Structures

CoVisor imposes fine-grained access control on how each controller can match and modify packets. These restrictions both enhance security and provide hints that allow CoVisor to further optimize the compilation process. First, by knowing which fields individual policies match on and modify, we can build custom data structures to index rules, instead of resorting to general R-tree-based data structures for multi-dimensional classifiers as in [49, 122, 127]. Second, by correlating the matched or modified fields of two policies being composed, we can simplify their indexing data structures by only considering the fields they *both* care about.



(a) Example rule index. (b) Example syntax tree.

Figure 2.8: Example of exploiting policy structures.

We first describe the optimization problem, and then we show how to use the above two insights to solve it. For ease of explanation, we first assume that member policies are connected by the parallel operator. Later, we'll describe how to handle the sequential and override operators. Now suppose we have a parallel composition $T_1 + T_2$ with implementation $comp_+(R_1, R_2)$, and a new rule, r_1^*, is inserted into R_1. With our incremental update algorithm (§2.3.2), we need to iterate over all (r_1^*, r_{2j}) pairs where $r_{2j} \in R_2$. The iteration processes $|R_2|$ pairs in total, where $|R_2|$ denotes the number of rules in R_2. However, if we know the structure of R_2, we can index its rules in a way that allows us to skip the rules that don't intersect with r_1^*, thereby further reducing computation overhead.

Index policies based on structure hints: Our goal is to reduce the number of rule pairs to iterate in compilation. A policy's structure indicates which fields should be indexed and how. For example, if R_2 is permitted only to do exact-match on destination MAC, then we can store its rules in a hash map keyed on destination MAC. If r_1^* also does exact-match on destination MAC, we simply use the destination MAC as key to search for rules in R_2's hash map. No rules in R_2 besides those stored under this key can intersect with r_1^*, because they differ on destination MAC. If r_1^* wildcards destination MAC, we return all rules in R_2, as they all intersect with r_1^*.

The preceding example is a simple case in which R_2 matches on one field. In general, a policy may match on multiple fields. We use single-field indexes (hash table for exact-

match, trie for prefix-match, list for arbitrary wildcard-match) as building blocks to build a *multi-layer index* for multiple fields. Specifically, we first choose one field f_1 the policy can match and index the policy on this field. We store all rules with the same value in f_1 in the same bucket of the index. This forms the first layer of the index. Then we choose the second field f_2 and index rules in each f_1 bucket on f_2. We repeat this process for all the fields on which the policy can match. We choose the order of fields according to simple heuristics like preferring exact-match fields to prefix-match fields. In practice, a policy normally matches on a small number of fields, which means the number of layers is small.

Consider a policy that does exact-match on $proto$ (protocol number) and prefix-match on $srcip$. We first index the policy based on $proto$. All rules with the same value in $proto$ go to the same bucket, as shown in Figure 2.8(a). Note that the hash map contains a bucket keyed on $*$ for rules that do not match on $proto$. Then, we index all the rules that contain the same $proto$ value on $srcip$. Because our example policy does prefix match on $srcip$, the second level of our multi-layer index comprises a trie for each bucket in the hash map. Figure 2.8(a) shows this second level for rules with $proto = 1$; bucket A contains all the rules with $proto = 1$ and $srcip = 128.0.0.0/1$.

Correlate policy structures to reduce indexing fields: When composing policies, we can leverage the information we know about both to reduce the work we do to index each. Suppose R_1 matches on $dstip$ and R_2 matches on the five tuple ($srcip$, $dstip$, $srcport$, $dstport$, $proto$). Instead of storing R_2 in a five-layer index, we need only index the $dstip$. Because $dstip$ is the only field on which any rule r_1^* added to R_1 can match, r_1^* will intersect with a rule in R_2 as long as they intersect on $dstip$. Formally, let $R_i.fields$ be the set of fields on which R_i matches and $R_i.index$ be the set of fields R_i indexes. Given R_i and R_j in a composition, we have

$$R_i.index = R_j.index = R_i.fields \cap R_j.fields. \tag{2.5}$$

Back to our example, we have $R_1.index = R_2.index = R_1.fields \cap R_2.fields = \{dstip\}$.

A policy R_i itself may be composed from other policies R_j and R_k. Unlike in the previous example, we do not *a priori* know $R_i.fields$ and instead rely on the observation that a rule in a composed policy can match on a field f if and only if at least one of its component member policies can match on f. Hence, we get

$$R_i.fields = R_j.fields \cup R_k.fields. \tag{2.6}$$

Let's look at an example $(R_1 + R_2) + R_3$, which we show as a syntax tree in Figure 2.8(b). Initially, we know the match fields only for the leaf nodes. Then we calculate the match fields for node $+_1$ with $R_1.fields \cup R_2.fields = \{srcip, dstip, srcprt, dstprt, proto\}$. Then, we use Equation (2.5) to index $+_1$ and R_3 with $+_1.fields \cap R_3.fields = \{srcip, proto\}$.

Sequential and override composition: Suppose we have sequential composition $T_1 \gg T_2$ with implementation $comp_{\gg}(R_1, R_2)$. Then $R_1.fields$ not only contains the fields R_1 matches but also the fields it modifies in its action set. This is because, for $r_1 \in R_1$ and $r_2 \in R_2$, the pair (r_1, r_2) generates a rule for the composed policy if the intersection of $r_2.mSet$ and the set of packets resulting from applying $r_1.action$ to $r_1.mSet$ is not empty. Similarly, when we index R_1, the key for any rule r_{1i} is the value resulting from applying $r_1.action$ to $r_1.match$. We do not need to index policies for override composition, since we directly stack their rules.

2.6 Implementation

We implemented a prototype of CoVisor with 4000+ lines of Java code added to and modifying OpenVirteX [5]. We replaced the core logic of OpenVirteX, which isolates multiple controllers, with our composition and incremental update logic (§2.3). To OpenVirteX's built-in many-to-one virtualization, we added support for the one-to-many abstraction and

our proactive compilation algorithm (§2.4). We further optimized compilation by exploiting the structure of policies as described in §2.5. We used `HashMap` in the Java standard library [59] to index rules with exact-match fields and `RadixTree` in the Concurrent-trees library [30] to index rules with prefix-match fields. Given a key (e.g., 1.0.0.0/16), `RadixTree` in the Concurrent-trees library only returns values for keys starting with this key (e.g., 1.0.0.0/24 and 1.0.0.0/30). We modified it to also give values for keys included by this key (e.g., 1.0.0.0/8). CoVisor currently supports the OpenFlow flowmod message; other commands, such as barrier messages and querying counters, will be supported in later versions.

2.7 Evaluation

2.7.1 Methodology

Experiment Setup: We evaluate CoVisor under three scenarios, the first two of which evaluate composition efficiency and the third of which evaluates devirtualization efficiency. In each scenario, we stress CoVisor with a wide range of policy sizes. Since compiling policies to individual physical switches is independent in these scenarios, we show the results for a single physical switch. We run CoVisor on Mininet [51] and use Floodlight controllers [37]. The server is equipped with an Intel XEON W5580 processor with 8 cores and 6GB RAM. We describe each scenario in more detail below.

- **L2 Monitor + L2 Router:** L2 Monitor counts packets for source MAC and destination MAC pairs; L2 Router forwards packets based on destination MAC. The MAC addresses are randomly generated.

- **L3-L4 Firewall ≫ L3 Router:** L3-L4 Firewall filters packets based on the five tuple; L3 Router forwards packets according to destination IP prefix. The firewall policy is

generated from ClassBench [125], a tool for benchmarking firewalls. The L3 router policy is generated with IP prefixes extracted from the firewall policy.

- **Gateway virtualization:** This is the topology virtualization discussed in §2.2.2. A switch that connects an Ethernet island to the IP core is abstracted to three virtual switches, which operate as a MAC learner, gateway, and IP router.

Metrics: We use the following metrics to measure efficiency. The thick bars in Figures 2.9 and 2.11 indicate the median, and the error bars show the 10th and 90th percentiles.

- **Compilation time:** The time to compile the policy composition or topology devirtualization.

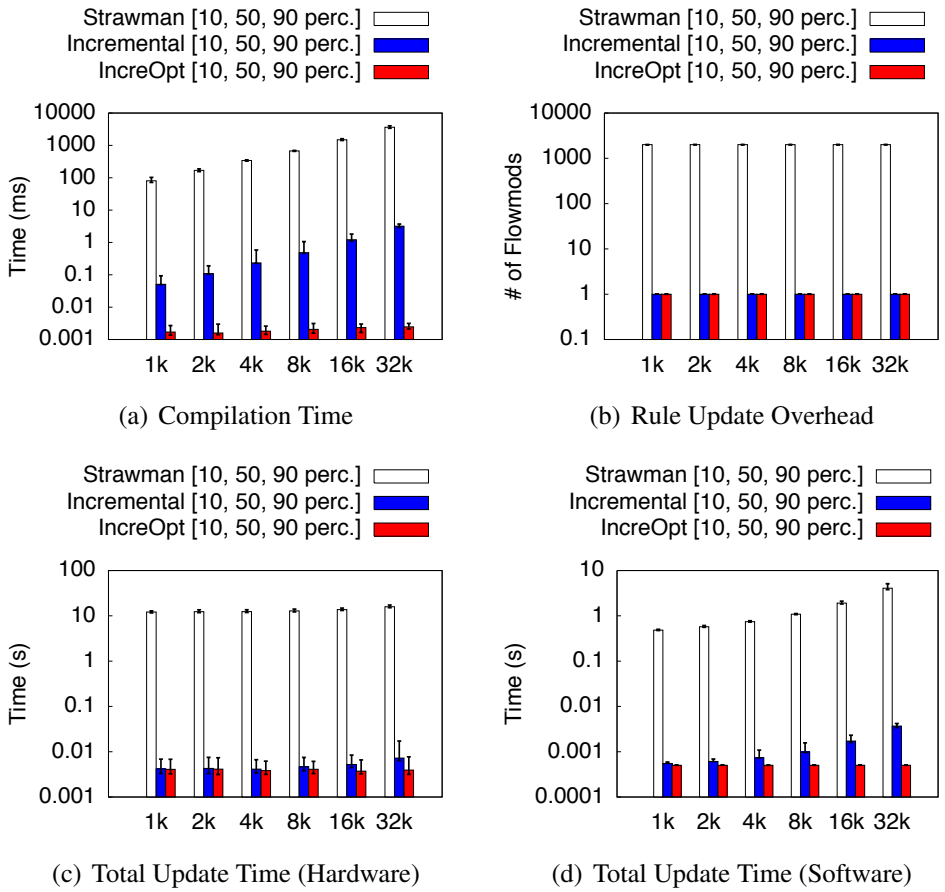
- **Rule update overhead:** The number of flowmods to update the switch to the new flow table.

- **Total update time:** The sum of compilation time, rule update time, and additional system overhead like OpenFlow message (un)marshalling. Since hardware switches and software switches takes very different time in rule updates, we show both of them. As the software switches in Mininet do not mimic the rule update latency of hardware switches and do not give accurate timing on the actual rule installation in software switches, we use the rule update latency in [65] for hardware switches and that in [113] for software switches when calculating rule update times.

Comparison: We compare the following approaches.

- **Strawman:** Recompile the new policy from scratch for every policy update.

- **Incremental:** Incrementally compile the new policy using our algebra of rule priorities for policy composition (§2.3) and keeping symbolic path information for topology devirtualization (§2.4).



(a) Compilation Time

(b) Rule Update Overhead

(c) Total Update Time (Hardware)

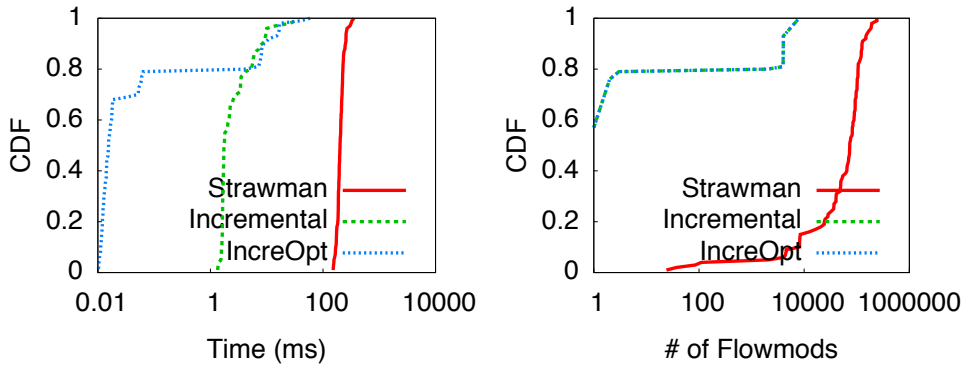
(d) Total Update Time (Software)

Figure 2.9: Per-rule update overhead of L2 Monitor + L2 Router as a function of L2 Router size (log-log scale).

- **IncreOpt:** Further optimize Incremental by exploiting the structures of policies (§2.5).

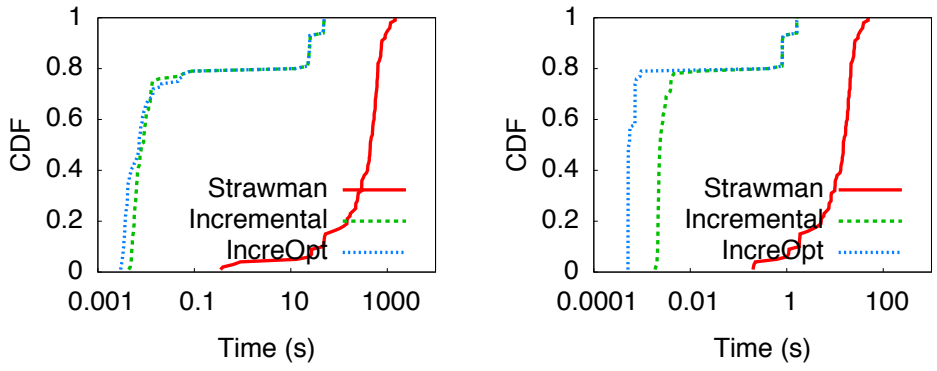
2.7.2 Composition Efficiency

Figure 2.9 shows the result of L2 Monitor + L2 Router. In this experiment, we initialize the L2 Monitor policy with 1000 rules, and then add 10 rules to measure the overhead for each. We repeat this process 10 times. We vary the size N of L2 Router policy from 1000 to 32,000 to show how overhead increases with larger policies. Figure 2.9(a) shows the compilation time. As expected, the compilation time of Strawman and Incremental increases with the policy size, because larger policies force our algorithm to consider more rule pairs.



(a) Compilation Time

(b) Rule Update Overhead



(c) Total Update Time (Hardware)

(d) Total Update Time (Software)

Figure 2.10: Per-rule update overhead of L3-L4 Firewall $\gg$ L3 Router (x-axis log scale).

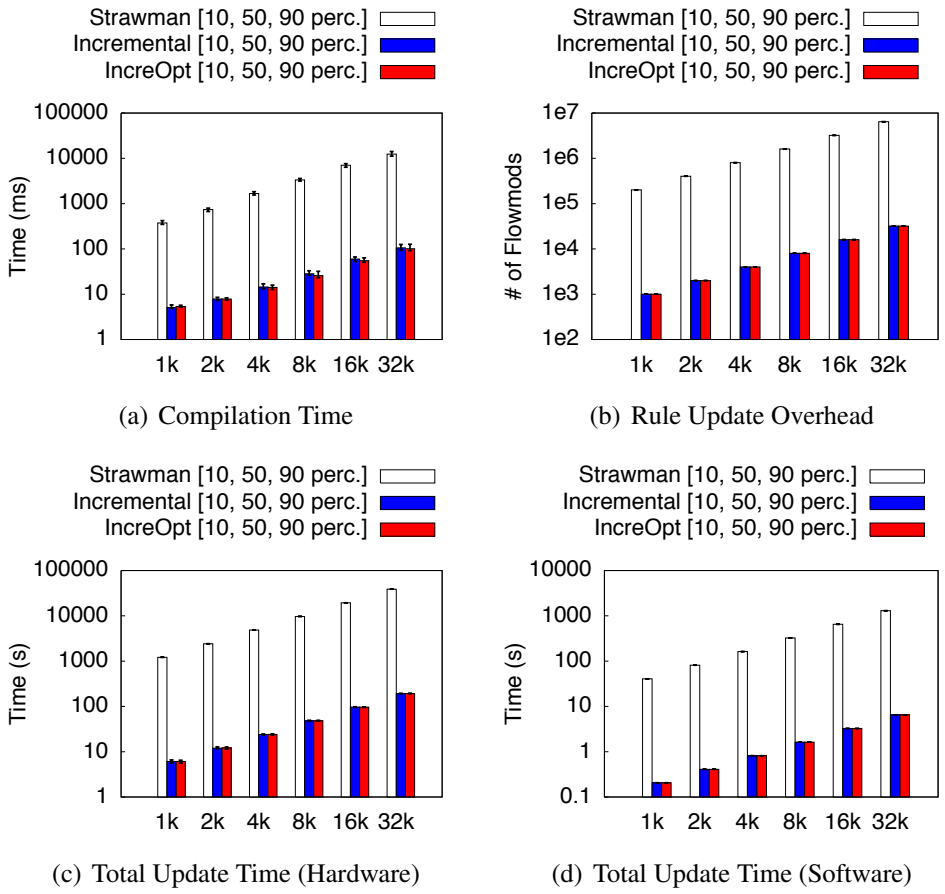
Since Strawman recompiles the whole policy, it is by far the slowest. On the other hand, IncreOpt has almost constant compilation time, because it indexes L2 Router's rules in a hash table keyed on destination MAC. When a rule is inserted to L2 Monitor's policy, the algorithm simply uses the rule's destination MAC to look up rules in the hash table.

Figure 2.9(b) shows the rule update overhead in terms of number of rules (same for hardware and software switches). Because of its naive priority assignment scheme, Strawman unnecessarily changes priorities of many existing rules and thus generates more flow-mods than Incremental and IncreOpt. Incremental and IncreOpt generate the same policy, and therefore they have the same rule update overhead. We also observe that the rule update overhead does not increase with the size of L2 Router's policy. This is because the

size of L2 Monitor's policy is fixed, and each monitor rule only intersects with one rule in L2 Router, since they both do exact-match on destination MAC.

Finally, Figures 2.9(c) and 2.9(d) show the total time. Notably, Incremental and IncreOpt are significantly faster than Strawman, and the gap between Incremental and IncreOpt is larger when using software switches. This is because software switches update rules faster than hardware switches, and therefore the compilation time accounts for a larger fraction of the total time for software switches.

Figure 2.10 shows the result of L3-L4 Firewall $\gg$ L3 Router. As before, we initialize L3-L4 Firewall's policy with 1000 rules and add 10 rules. Since the trend is similar to Figure 2.9 when we vary the size N of L3 Router, we instead show the CDF when L3 Router policy has 8,000 rules. Figure 2.10(a) shows the compilation time. Again, Strawman is several orders of magnitude slower than Incremental and IncreOpt. However, unlike in our previous experiment, we see a stepwise behavior of Incremental, and the difference between Incremental and IncreOpt also disappears after 80th percentile. This is an artifact of the content of L3-L4 Firewall from ClassBench. The firewall policy comprises approximately 80% rules matching on very specific destination IP prefix (/31, /32) and around 20% rules matching very general destination IP prefix (/1, /0). A firewall rule with a very specific destination IP prefix only composes with a few router rules, in which case IncreOpt processes fewer rule pairs in compilation than Incremental. On the other hand, a firewall rule with a very general destination IP prefix like /1 or /0 composes with half or all rules in the router policy, in which case Incremental and IncreOpt process a similar number of rule pairs and have similar compilation time. This reasoning also explains the shape of Incremental and IncreOpt in Figures 2.10(b), 2.10(c) and 2.10(d). Finally, note that the overhead of inserting a new rule to L3-L4 Firewall by Incremental and IncreOpt is bounded by the number of rules in L3 Router, while that by Strawman is bounded by the product of the number of rules in L3-L4 Firewall and L3 Router.



(a) Compilation Time

(b) Rule Update Overhead

(c) Total Update Time (Hardware)

(d) Total Update Time (Software)

Figure 2.11: The switch connecting an Ethernet island to the IP core is virtualized to switches that operate as MAC learner, gateway, and IP router. Figures show the overhead of adding a host to the Ethernet island as a function of IP router policy size (log-log scale).

2.7.3 Devirtualization Efficiency

We use the gateway scenario to evaluate the efficiency of the devirtualization phase of compilation. In this experiment, we have 100 hosts in the Ethernet island. The MAC learner installs forwarding rules for connections between host pairs. To the Ethernet island, switch G simply appears as another host; hosts use G's MAC as destination MAC when they want to reach hosts across the IP core. We initialize the MAC learner policy with 1000 rules in switch E. Then, we add a new host to the Ethernet island. When the new host tries to talk to another host across the IP core, the MAC learner adds two rules to establish a bidirectional connection between the host and switch G. To compile this update, we

compose the two new rules with the existing rules in switches G and I. The gateway policy at G is simply a MAC-rewriting repeater and ARP server. The IP router forwards packets based on destination IP prefix. We vary the size of the IP router policy at I from 1000 to 32,000 to evaluate how the overhead increases with larger policies.

Figure 2.11 shows the overhead. Strawman exhibits a long compilation time, as it has to recompile the policy from scratch. Strawman also generates more flowmods than necessary, because its priority assignment scheme may change the priorities of existing rules. In contrast, Incremental and IncreOpt incur significantly less overhead, because they keep all the symbolic paths and only need to change a few upon receiving the new rules. Finally, we notice that Incremental and IncreOpt do not show much difference in this experiment and the absolute values of total update time are high. This is because the MAC learner policy in switch E and the IP router policy in switch I match on different fields. Thus, when we do sequential composition on virtual paths, Incremental and IncreOpt iterate over a similar number of rule pairs and the result policy is almost a cross-product of the two policies at E and I. The cross-product is inevitable when compiling to a single flow table as the two policies match on different fields. Finally, we note that the multi-table support in OpenFlow 1.3 and newer hardware platforms like P4 [67] can make devirtualization more efficient. If multiple tables in a switch can be configured to in a pipeline to mirror the virtual network topology, then updating virtual switch tables can be directly mapped to updating physical tables. This can dramatically reduce compilation and rule update overhead. A complete exploration of this direction is part of our future work.

2.8 Proposed OpenFlow Extensions

The current language for communicating between CoVisor and its controllers is OpenFlow. We made this choice because OpenFlow is the current *lingua franca* of software-defined networks. Nevertheless, it is well-known that OpenFlow is not the ultimate SDN control

protocol; both researchers and practitioners have been exploring extensions and revisions to the protocol for years. However, our use of OpenFlow in CoVisor has highlighted additional limitations that researchers might consider when revising the OpenFlow standard or when designing future protocols.

In particular, a single OpenFlow rule can only express *positive* properties of packets in a compact manner. For example, a single rule can forward a packet with type SSH out port 3, but it cannot forward a packet that does *not* have type SSH out port 3. This lack of expressiveness can be problematic if one would like to construct a hypervisor that allows controller A to choose to handle some traffic, while other traffic falls through to controller B. Such a situation is expressed naturally as $A \rhd B$ in our system. However, if A chooses (during the course of operation) to control forwarding for packets that do *not* have type SSH, it can only do so by providing rules for all types of packets other than SSH packets. If OpenFlow supplied a *don't care* action (analogous to Pyretic's *pass-through* action [92]), controllers could generate just two rules to deal with such situations: a high-priority rule for SSH traffic with a don't care action and a lower priority rule that forwards all other traffic as desired. Of course, it would be possible for us to "hack" the OpenFlow protocol so controllers can transmit such information coded somehow, but hacking protocols in this fashion is brittle and leads to long-term software engineering nightmares.

2.9 Related Work

Slicing: Existing network hypervisors mostly focus on slicing; they target multi-tenancy scenarios in which each tenant operates on a disjoint subset, or *slice*, of the traffic [5, 120, 75, 34]. In contrast, CoVisor allows multiple controllers to collaborate on processing the same traffic.

Topology abstraction: Many projects studied the many-to-one case [92, 70, 71, 22]. Pyretic [92] explored the one-to-many case, but its implementation reactively installs

micro-flow rules. CoVisor provides the first proactive compilation algorithm by leveraging symbolic analysis to build symbolic paths [73] and applying incremental sequential composition to generate the rules.

Composition: The parallel and sequential operators are proposed in the Frenetic project [39, 92], and the override operator is described in [48, 133, 20]. An incremental compilation algorithm for Frenetic policies is introduced in [133]. CoVisor is novel in using these composition operators to compose policies written on a variety of controller platforms, rather than just Frenetic. Furthermore, CoVisor takes advantage of the OpenFlow rules' explicit priorities; it uses a convenient algebra to calculate priorities for composed rules, thereby eliminating the need to build dependency graphs for rules and maintain scattered priority distributions [133]. Moreover, [133] only optimizes priority assignment for Frenetic policies; it is not a hypervisor to compose controllers, and does not have algorithms to compile topology virtualizations and optimizations by exploiting policy structures. Finally, it is an open problem to design a good interface for Frenetic to aid incremental update.

Switch table type patterns: Table Type Patterns [3] and P4 [17] provide a syntax for describing flow table capabilities (e.g., fields that can be matched and modified). CoVisor uses this kind of information to build a customized data structure to optimize compilation. CoVisor's optimization technique differs from existing ways to index and accelerate multi-dimensional classifiers that don't know policy structures $a\ priori$ [49, 122, 127, 119].

2.10 Conclusion

We present CoVisor, a compositional hypervisor that allows administrators to combine multiple controllers to collaboratively process a network's traffic. CoVisor uses a combination of novel algorithms and data structures to efficiently compile policies in an incremental

manner. Evaluations on our prototype show that it is several orders of magnitude faster than a naive implementation.

Chapter 3

Dionysus: Dynamic Update Scheduling

This chapter presents Dionysus, a system for fast, consistent network updates in software-defined networks. Dionysus encodes as a graph the consistency-related dependencies among updates at individual switches, and it then dynamically schedules these updates based on runtime differences in the update speeds of different switches. This dynamic scheduling is the key to its speed; prior update methods are slow because they pre-determine a schedule, which does not adapt to runtime conditions. Testbed experiments and data-driven simulations show that Dionysus improves the median update speed by 53–88% in both wide area and data center networks compared to prior methods.

3.1 Introduction

Many researchers have shown the value of centrally controlling networks. This approach can prevent oscillations due to distributed route computation [19]; ensure that network paths are policy compliant [46, 21]; reduce energy consumption [53]; and increase throughput [4, 15, 31, 58, 55]. Independent of their goal, such systems operate by frequently updating the data plane state of the network, either periodically or based on triggers such as failures. This state consists of a set of rules that determine how switches forward packets.

A common challenge faced in all centrally-controlled networks is consistently and quickly updating the data plane. Consistency implies that certain properties should not be violated during network updates, for instance, packets should not loop (*loop freedom*) and traffic arriving at a link should not exceed its capacity (*congestion freedom*). Consistency requirements impose dependencies on the order in which rules can be updated at switches. For instance, for congestion freedom, a rule update that brings a new flow to a link must occur after an update that removes an existing flow if the link cannot support both flows simultaneously. Not obeying update ordering requirements can lead to inconsistencies such as loops, blackholes, and congestion.

Current methods for consistent network updates are slow because they are based on *static* ordering of rule updates [55, 85, 112, 72]. They pre-compute an order in which rules must be updated, and this order does not adapt to runtime differences in the time it takes for individual switches to apply updates. These differences inevitably arise because of disparities in switches' hardware and CPU load and the variabilities in the time it takes the centralized controller to make remote procedure calls (RPC) to switches. In B4, a centrally-controlled wide area network, the ratio of the 99th percentile to the median delay to change a rule at a switch was found to be over five (5 versus 1 second) [58]. Further, some switches can "straggle," taking substantially more time than average (e.g., 10-100x) to apply an update. Current methods can stall in the face of straggling switches.

The speed of network updates is important because it determines the agility of the control loop. If the network is being updated in response to a failure, slower updates imply a longer period during which congestion or packet loss occurs. Further, many systems update the network based on current workload, both in the wide area [58, 55] and the data center [4, 15, 31], and their effectiveness is tied to how quickly they adapt to changing workloads. For example, recent works [58, 55] argue for frequent traffic engineering (e.g., every 5 minutes) to achieve high network utilization; slower network updates would lower network utilization.

We develop a new approach for consistent network updates. It is based on the observations that $i)$ there exist multiple valid rule orderings that lead to consistent updates; and $ii)$ dynamically selecting an ordering based on update speeds of switches can lead to fast network updates. Our approach is general and can be applied to many consistency properties, including all the ones that have been explored by prior work [55, 85, 112, 72, 89].

We face two main challenges in practically realizing our approach. The first is devising a compact way to represent multiple valid orderings of rule updates; there can be exponentially many such orderings. We address this challenge using a dependency graph in which nodes correspond to rule updates and network resources, such as link bandwidth and switch rule memory capacity, and (directed) edges denote dependencies among rule updates and network resources. Scheduling updates in any order, while respecting dependencies, guarantees consistent updates.

The second challenge is scheduling updates based on dynamic behavior of switches. This problem is NP-complete in the general case, and making matters worse, the dependency graph can also have cycles. To schedule efficiently, we develop greedy heuristics based on preferring critical paths and strongly connected components in the dependency graph [80].

We instantiate our approach in a system called Dionysus and evaluate it using experiments on a modest-sized testbed and large-scale simulations. Our simulations are based on topology and traffic data from two real networks, one wide-area network and one data center network. We show that Dionysus improves the median network update speed by 53–88%. We also show that its faster updates lower congestion and packet loss by over 40%.

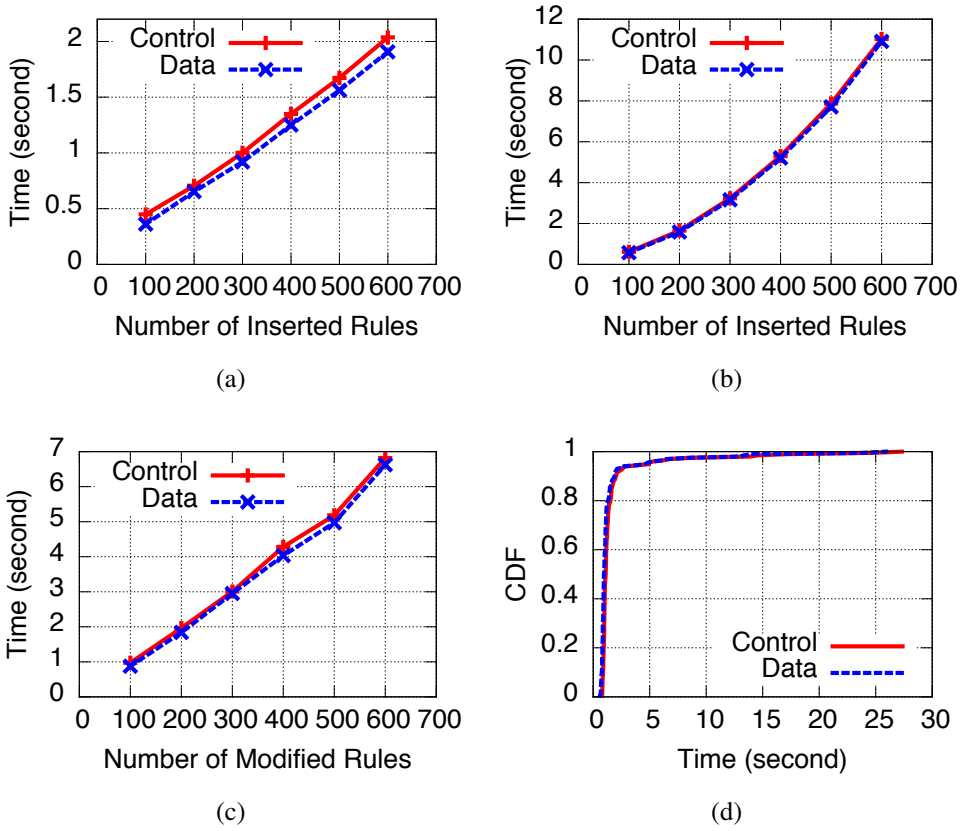


Figure 3.1: Rule update times on a commodity switch. (a) Inserting single-priority rules. (b) Inserting random-priority rules. (c) Modifying rules in a switch with 600 single-priority rules. (d) Modifying 100 rules in a switch with concurrent control plane load.

3.2 Motivation

Our work is motivated by the observations that the time to update switch rules varies widely and that not accounting for this variation leads to slow network updates. We illustrate these observations using measurements from commodity switches and simple examples.

3.2.1 Variability in Update Time

Several factors lead to variable end-to-end rule update times, including switch hardware capabilities, control load on the switch, the nature of the updates, RPC delays (which include network path delays), etc. [31, 58, 113, 36]. To illustrate this variability, we perform controlled experiments on commodity switches. In these experiments, RPC delays are neg-

ligible and identical switch hardware and software are used, yet significant variability is evident.

The experiments explore the impact of four factors: i) the number of rules to be updated; ii) the priorities of the rules; iii) the types of rule updates (e.g., insertion vs. modification); and iv) control load on the switch. We measure switches from two different vendors and observe similar results. Figure 3.1 shows results for one switch vendor. We build a customized switch agent on the switch and obtain confirmation of rule updates in both the control and data planes. The control plane confirmation is based on the switch agent verifying that the update is installed in the switch's TCAM (ternary content addressable memory), and the data plane confirmation is based on observing the impact of the update in the switch's forwarding behavior (e.g., changes in which interface a packet is sent out on).

Figure 3.1(a) shows the impact of the number of rules by plotting the time to add different numbers of rules. Here, the switch has no control load besides rule updates, the switch starts with an empty TCAM, and all rule updates correspond to adding new rules with the same priority. We see that, as one might expect, that the update time grows linearly with the number of rules being updated, with the per-rule update time being 3.3 ms.

Figure 3.1(b) shows the impact of priorities. As above, the switch has no load and starts with an empty TCAM. The difference is that the inserted rules are assigned random priorities. We see that the per-rule update time is significantly higher than before. The slope of the line increases as the number of rules increase, and the per-rule update time reaches 18 ms when inserting 600 rules.

This variability stems from the fact that TCAM packing algorithms do different amounts of work, depending on the TCAM's current content and the type of operation performed. For instance, the TCAM itself does not encode any rule priority information. The rules are stored from top to bottom in decreasing priority and when multiple rules match a packet, the one with the highest place is chosen. Thus, when a new rule is inserted,

it may cause existing rules to move in the table. Although the specific packing algorithms are proprietary and vary across vendors, the intrinsic design of a TCAM makes the update time variable.

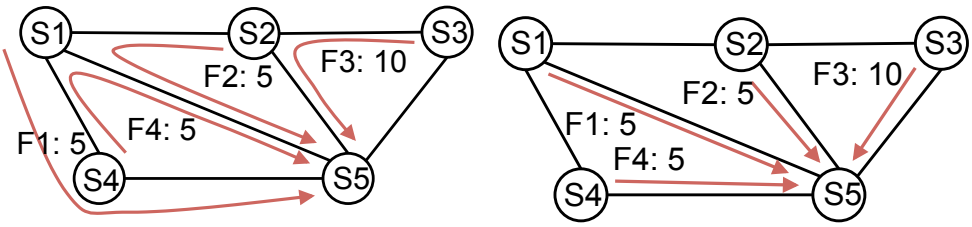
Figure 3.1(c) shows the impact of the type of rule update. Rather than inserting rules into an empty TCAM, we start with 600 rules of the same priority and measure the time for rule modifications. We modify only match fields or actions, not rule priorities. The graph is nearly linear, with a per-rule modification latency of 11 ms. This latency is larger than the per-rule insertion latency because a rule modification requires two operations in the measured switch: inserting the new rule and deleting the old rule.

Finally, Figure 3.1(d) shows the impact of control load, by engaging the switch in different control activities while updates are performed. Here, the switch starts with the 600 same-priority rules and we modify 100 of them. Control activities performed include reading packet and byte counters on rules with OpenFlow protocol, querying SNMP counters, reading switch information with CLI commands, and running BGP protocol (which SDN systems use as backup [58]). We see that despite the fact that update operations are identical (100 new rules), the time to update highly varies, with the 99th percentile 10 times larger than the median. Significant rule update time variations are also reported in [58, 36].

In summary, we find that even in controlled conditions, switch update time varies significantly. While some sources of this variability can be accounted for statically by update algorithms (e.g., number of rule updates), others are inherently dynamic in nature (e.g., control plane load and RPC delays). Accounting for these dynamic factors ahead of time is difficult. Our work thus focuses on adapting to them at runtime.

3.2.2 Consistent Updates amid Variability

We illustrate the downside of static ordering of rule updates with the example of Figure 3.2. Each link has a capacity of 10 units and each flow's size is marked. The controller wants to update the network configuration from Figure 3.2(a) to 3.2(b). Assume for simplicity



(a) Current State (b) Target State

Figure 3.2: A network update example. Each link has 10 units of capacity; flows are labeled with their sizes.

that the network uses tunnel-based routing and all necessary tunnels have already been established. So, moving a flow requires updating only the ingress switch.

If we want a congestion-free network update, we cannot update all the switches in "one shot" (i.e., send all update commands simultaneously). Since different switches will apply the updates at different times, such a strategy may cause congestion at some links. For instance, if $S1$ applies the update for moving $F1$ before $S2$ moves $F2$ and $S4$ moves $F4$, link $S1$-$S5$ will be congested.

Ensuring that no link is congested requires us to carefully order the updates. Two valid orderings are:

Plan A: $[F3 \rightarrow F2]$ $[F4 \rightarrow F1]$

Plan B: $[F4]$ $[F3 \rightarrow F2 \rightarrow F1]$

Plan A mandates that $F2$ be done after $F3$ and $F1$ be done after $F4$. Plan B mandates that $F1$ be done after $F2$ and that $F2$ be done after $F3$. In both plans, $F3$ and $F4$ have no pre-requisites and can be done anytime and in parallel.[1]

Which plan is faster? In the absence of update time variability, if all updates take unit time, Plan A will take 2 time units and Plan B will take 3. However, with update time variability, no plan is a clear winner. For instance, if $S4$ takes 3 time units to move $F4$, and other switches take 1, Plan A will take 4 time units and Plan B will take 3. On the other

[1]Some consistent update methods [55] use stages, a more rigid version of static ordering. They divide updates into multiple stages, and all updates in the previous stage must finish before any update in the next stage can begin. In this terminology, Plan A is a two-stage solution in which the first stage will update $F3$ and $F4$ and the second will update $F2$ and $F1$. Plan B is a three-stage solution. Since SWAN minimizes the number of stages, it will prefer Plan A.

hand, if $S2$ is slow and takes 3 time units to move $F2$, while other switches take 1, Plan A will take 4 time units and Plan B will take 5.

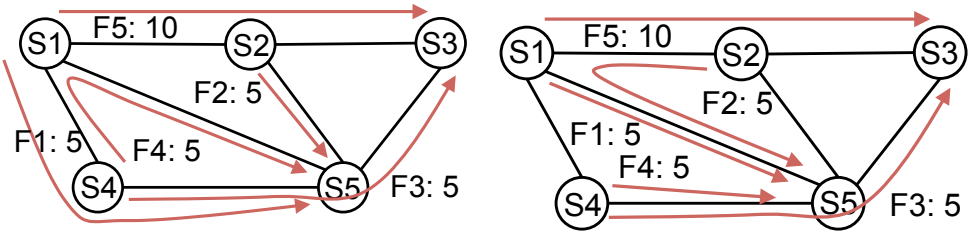
Now consider a dynamic plan that first issues updates for $F3$ and $F4$, issues an update for $F2$ as soon as $F3$ finishes, and issues an update for $F1$ as soon as $F2$ *or* $F4$ finishes. This plan dynamically selects between the two static plans above and will thus equal or beat those two plans regardless of which switches are slow to update. Practically implementing such plans for arbitrary network topologies and updates is the goal of our work.

3.3 Dionysus Overview

We achieve fast, consistent network updates through dynamic scheduling of rule updates. As in the example above, there can be multiple valid rule orderings that lead to consistent updates. Instead of statically selecting an order, we implement on-the-fly ordering based on the realtime behavior of the network and the switches.

Our focus is on flow-based traffic management applications for the network core (e.g., ElasticTree, MicroTE, B4, SWAN [53, 15, 58, 55]). As is the case for these applications, we assume that any forwarding rule at a switch matches at most one flow, where a flow is (a subset of) traffic between ingress and egress switches that uses either single or multiple paths. This assumption does not hold in networks that use wild-card rules or longest prefix matching. Increasingly, such rules are being moved to the network edge or even hosts [96, 23, 108], keeping the core simple with exact match rules.

The primary challenge is to tractably explore valid orderings. One difficulty is that there are combinatorially many such orderings. Conceivably, one may formulate the problem as an ILP (Integer Linear Program). But this approach would be too slow and does not scale to large networks with a lot of flows. Also it is static and not incrementally computable; one has to rerun the ILP every time the switch behaviors change. Another difficulty is that the extreme approach of being completely opportunistic about rule ordering does not always



(a) Current State (b) Target State

Figure 3.3: An example in which a completely opportunistic approach to scheduling updates leads to a deadlock. Each link has 10 units of capacity; flows are labeled with their sizes. If $F2$ is moved first, $F1$ and $F4$ get stuck.

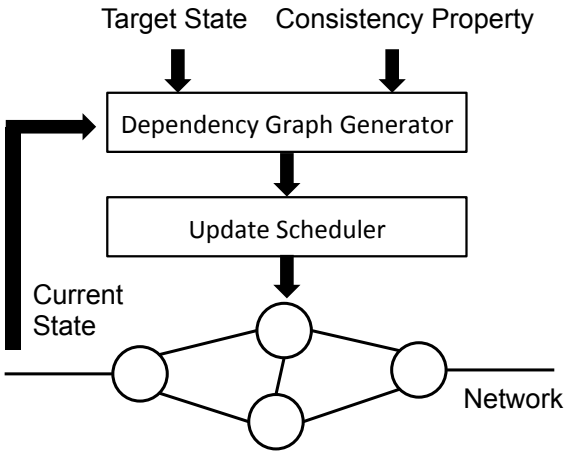
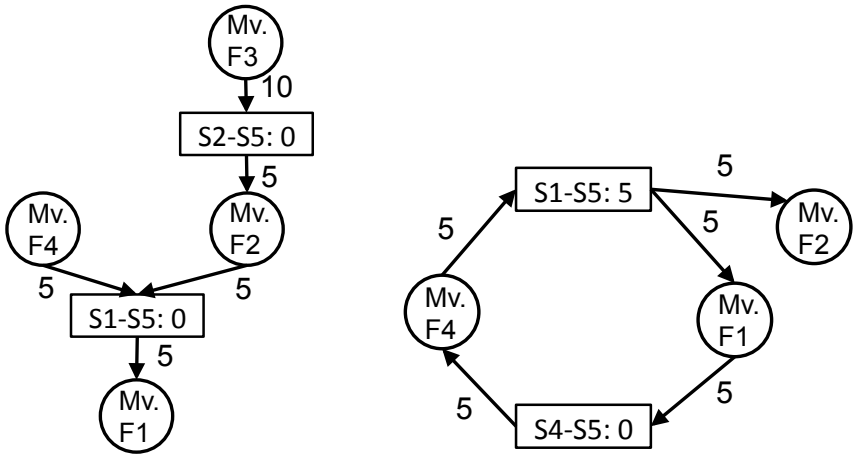


Figure 3.4: Our approach.

work. In such an approach, the controller will immediately issue any updates that are not gated (per consistency requirements) on any other update. While this approach works for the simple example in the previous section, in general, it can result in deadlocks (that are otherwise resolvable). Figure 3.3 shows an example. Since $F2$ can be moved without waiting for any other flow movement, an opportunistic approach might make that move. But at this point, we are stuck, because no flow can be moved to its destination without overloading at least some link. This is avoidable if we move other flows first. It is because of such possibilities that current approaches carefully plan transitions, but they err on the side of not allowing any runtime flexibility in rule orderings.



(a) Dependency graph for Figure 3.2 (b) Dependency graph for Figure 3.3

Figure 3.5: Example dependency graphs.

We balance planning and opportunism using a two-stage approach, shown in Figure 3.4. In the first stage, we generate a *dependency graph* that compactly describes many valid orderings. In the second stage, we schedule updates based on the constraints imposed by the dependency graph. Our approach is general in that it can maintain any consistency property that can be described using a dependency graph, which includes all properties used in prior work [58, 55, 112]. The scheduler is independent of the consistency property.

Figure 3.5(a) shows a simplified view of the dependency graph for the example of Figure 3.2. In the graph, circular nodes denote update operations, and rectangular nodes represent link capacity resources. The numbers within rectangles indicate the current free capacity of resources. A label on an edge from an operation to a resource node shows the amount of resource that will be released when the operation completes. For example, link $S2$-$S5$ has 0 free capacity, and moving $F3$ will release a capacity of 10 to it. Labels on edges from resource to operation nodes show the amount of free resource needed to conduct these operations. As moving $F1$ requires 5 free capacity on link $S1$-$S5$, $F1$ cannot move until $F2$ or $F4$ finishes.

Given the dependency graph in Figure 3.5(a), we can dynamically generate good schedules. First, we observe that $F3$ and $F4$ don't depend on other updates, so they can be

scheduled immediately. After $F3$ finishes, we can schedule $F2$. Finally, we schedule $F1$ once *one* of $F2$ or $F4$ finishes. From this example, we see that the dependency graph captures dependencies but still leaves scheduling flexibility, which we leverage at runtime to implement fast updates.

There are two challenges in dynamically scheduling updates. The first is to resolve cycles in the dependency graph. These arise due to complex dependencies between rules. For example, Figure 3.5(b) shows that there are cycles in the dependency graph for the example of Figure 3.3. Second, at any given time, multiple subsets of rule updates can be issued, and we need to decide which ones to issue first. As described later, the greedy heuristics we use for these challenges are based on critical-path scheduling and the concept of SCC (strongly connected component) in graph theory.

3.4 Network State Model

This section describes the model of network forwarding state that we use in Dionysus. The following sections describe dependency graph generation and scheduling in detail.

The network G consists of a set of switches S and a set of directed links L. A flow f is from an ingress switch s_i to an egress switch s_j with traffic volume t_f, and its traffic is carried over a set of paths P_f. The forwarding state of f is defined as $R_f = \{r_{f,p} | p \in P_f\}$ where $r_{f,p}$ is the traffic load of f on path p. The network state NS is then the combined state of all flows, i.e., $NS = \{R_f | f \in F\}$. For example, consider the network in Figure 3.6(a) that is forwarding a flow across two paths, with 5 units of traffic along each. Here, $t_f = 10$, $P_f = \{p_1 = S_1S_2S_3S_5, p_2 = S_1S_2S_5\}$, and $R_f = \{r_{f,p_1} = 5, r_{f,p_2} = 5\}$.

The state model above captures both tunnel-based forwarding that is prevalent in WANs and also WCMP (weighted cost multi path) forwarding that is prevalent in data center networks. In tunnel-based forwarding, a flow is forwarded along one or more tunnels. The ingress switch matches incoming traffic to the flow, based on packet headers, and splits

Forwarding schemes	WCMP forwarding Tunnel-based forwarding
Consistency properties	Blackhole-freedom Loop-freedom Packet coherence Congestion-freedom

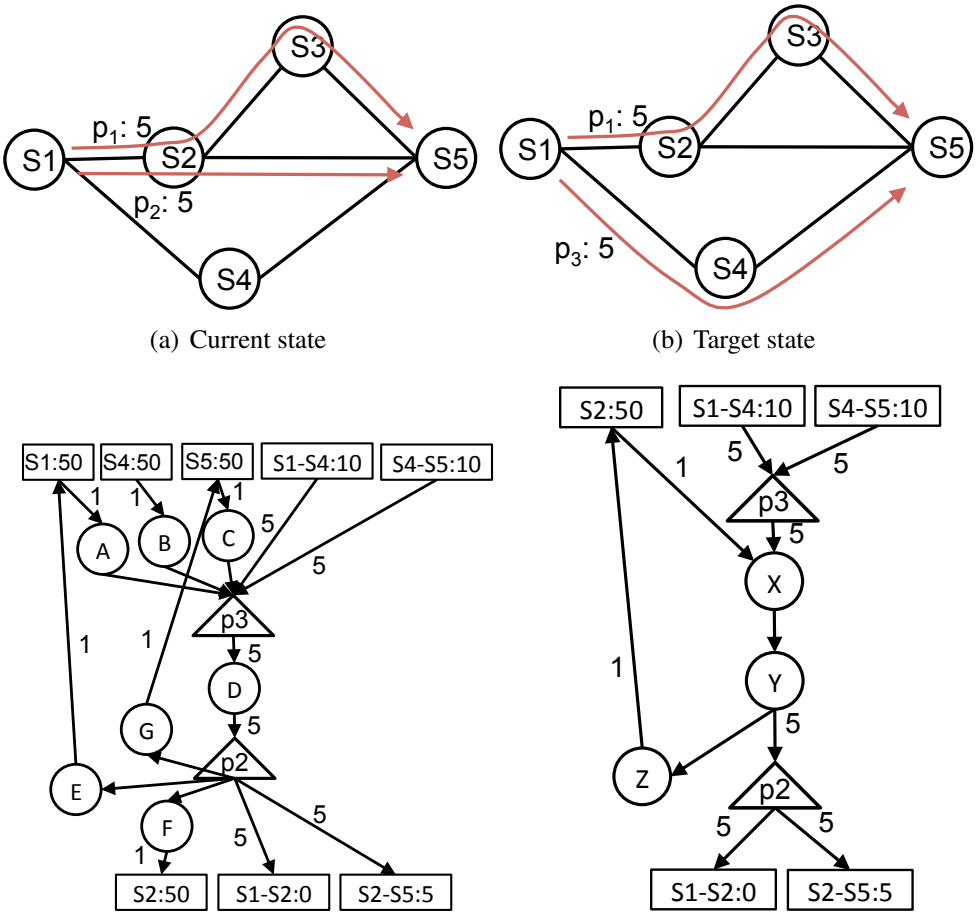
Table 3.1: Forwarding schemes and consistency properties that can currently be handled by the dependency graph generation.

it across the tunnels based on configured weights. Before forwarding a packet along a tunnel, the ingress switch tags the packet with the tunnel identifier. Subsequent switches only match on tunnel tags and forward packets, and the egress switch removes the tunnel identifier. Representing tunnel-based forwarding in our state model is straightforward. P_f is the set of tunnels and the weight of a tunnel is $r_{f,p}/t_f$.

In WCMP forwarding, switches at *every hop* match on packet headers and split flows over multiple next hops with configured weights. Shortest-path and ECMP (equal cost multipath) forwarding are special cases of WCMP forwarding. To represent WCMP routing in our state model, we first calculate the flow rate on link l as $r_f^l = \sum_{l \in p, p \in P_f} r_{f,p}$. Then at switch s_i, the weight for next-hop s_j is: $w_{i,j} = r_f^{l_{ij}} / \sum_{l \in L_i} r_f^l$ where l_{ij} is the link from s_i to s_j and L_i is the set of links starting at s_i. For instance, in Figure 3.6(a), $w_{1,2} = 1, w_{1,4} = 0, w_{2,3} = 0.5, w_{2,5} = 0.5$.

3.5 Dependency Graph Generation

As shown in Figure 3.4, the dependency graph generator takes as input the current state NS_c, the target state NS_t, and the consistency property. The network states includes the flow rate, and as in current systems [53, 15, 58, 55], we assume that flows obey this rate as a result of rate limiting or robust estimation. A static input to Dionysus is the rule capacity of each switch, relevant in settings where this resource is limited. Since Dionysus manages all rule additions and removals, it then knows how much rule capacity is available on each



(a) Current state

(b) Target state

(c) Dependency graph using tunnel-based rules (Table 3.2)

(d) Dependency graph using WCMP-based rules (Table 3.3)

Figure 3.6: Example of building dependency graph for updating flow f from current state (a) to target state (b).

switch at any given time. This information is used such that rule capacity is not exceeded at any switch.

Given NS_c and NS_t, it is straightforward to compute the set of operations that would update the network from NS_c to NS_t. The goal of dependency graph generation is to interlink these operations based on the consistency property. Our dependency graph has three types of nodes: *operation nodes*, *resource nodes*, and *path nodes*. Operation nodes represent addition, deletion, or modification of a forwarding rule at a switch, and resource nodes correspond to resources such as link capacity and switch memory and are labeled with the amount of resource currently *available*. An edge between two operation nodes captures an

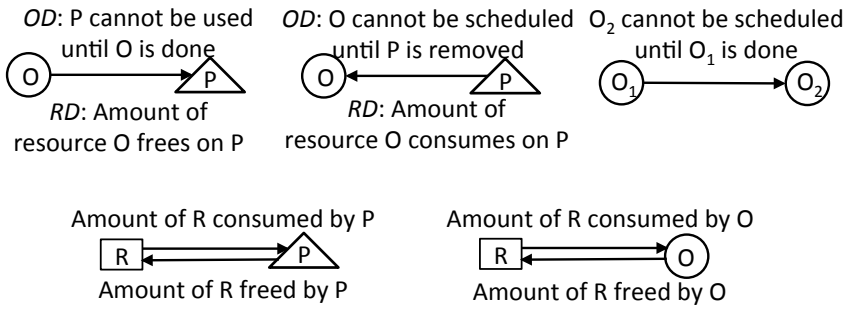


Figure 3.7: Links and relationships among path, operation, and resource nodes; *RD* indicates a resource dependency and *OD* indicates an operation dependency.

operation dependency and implies that the parent operation must be done before the child. An edge between a resource and an operation node captures a *resource dependency*. An edge from a resource to an operation node is labeled with the amount of resource that must be available before the operation can occur. An edge from an operation to a resource node is labeled with the amount of the resource that will be freed by that operation. There are no edges between resource nodes.

Path nodes help group operations and link capacity resources on a path. Path nodes can connect to operation nodes as well as to resource nodes. An edge between an operation and a path node can be either an operation dependency (un-weighted) or a resource dependency (weighted). The various types of links connecting different types of nodes are detailed in Figure 3.7.

During scheduling, each path node that frees link resources has a label $committed$ that denotes the amount of traffic that is moving away from the path; when the movement finishes, we use $committed$ to update the free resource of its child resource nodes. We do not need to keep $committed$ for path nodes that require resource, because we always reduce free capacity on its parent resource nodes first before we move traffic into the path.

In this thesis, we focus on four consistency properties from prior work [89] and show how our dependency graphs capture them. The properties are i) *blackhole-freedom*: no packet should be dropped at a switch (e.g., due to a missing rule); ii) *loop-freedom*: no packet should loop in the network; iii) *packet coherence*: no packet should see a mix of

old and new rules in the network; and iv) *congestion-freedom*: traffic arriving at a link should be below its capacity. Table 3.1 gives a list of forwarding schemes and consistency properties we can handle. Dionysus cannot handle network updates across layers, i.e., updates of forwarding rules in switches in the network layer and optical circuit setup in optical switches in the optical layer. §4.3.3 describes how to extend Dionysus to coordinate the updates between the network and optical layers. In cases where it is urgent to update some switches, i.e., blocking data leakage, operators may sacrifice consistency and choose the enforce the new policy right away. The dependency graph is not useful in such cases.

We now describe dependency graph generation. We first focus on tunnel-based forwarding without resource limits and then discuss WCMP forwarding and resource constraints. Similar to SWAN [55], we remove old flows at the beginning of the update and add new flows at the end of the update. This gives us most free resources to play with during the update. The following description mainly concerns with flows that change their forwarding states in the update.

Tunnel-based forwarding: Tunnel-based forwarding offers loop freedom and packet coherence by design; it is not possible for packets to loop or to see a mix of old and new rules during updates. We defer discussion of congestion freedom until we discuss resource constraints. The remaining property, blackhole freedom, is guaranteed as long as we ensure that i) a tunnel is fully established before the ingress switch puts any traffic on it, and ii) all traffic is removed from the tunnel before the tunnel is deleted.

A dependency graph that encodes these constraints can be built as follows. For each flow f, using NS_c and NS_t, we first calculate the tunnels to be added and deleted and generate a path node for each. Then, we generate an operation node for every hop, adding an edge from each of them to the path node (or from the path node to each of them), denoting adding (or deleting) this tunnel at the switch. Then, we generate an operation node that changes the tunnel weights to those in NS_t at the ingress switch. To ensure blackhole freedom, we add an edge from each path node that adds new tunnels to the operation node

Index	Operation
A	Add p_3 at S1
B	Add p_3 at S4
C	Add p_3 at S5
D	Change weight at S1
E	Delete p_2 at S1
F	Delete p_2 at S2
G	Delete p_2 at S5

Table 3.2: Operations to update f with tunnel-based rules.

Index	Operation
X	Add weights with new version at S2
Y	Change weights, assign new version at S1
Z	Delete weights with old version at S2

Table 3.3: Operations to update f in WCMP forwarding.

that changes tunnel weights, and an edge from the operation node that changes tunnel weights to each path node that deletes old tunnels.

We use the example in Figure 3.6 to illustrate the steps above. Initially, we set the tunnel weights on p_1 and p_2 with 0.5 and 0.5 respectively. In the target state, we add tunnel p_3, delete tunnel p_2, and change the tunnel weights to 0.5 on p_1 and 0.5 on p_3. To generate the dependency graph for this transition, we first generate path nodes for $p2$ and $p3$ and the related switch operations as in Table 3.2. Then we add edges from the tunnel-addition operations (A, B and C) to the corresponding path node ($p3$), and edges to the tunnel-deletion operations (E, F and G) from the corresponding path node ($p2$). Finally, we add an edge from the path node of the added path ($p3$) to the weight-changing operation (D) and from D to the path node for the path to be deleted ($p2$). The resulting graph is shown in Figure 3.6(c). The resource nodes in this graph are discussed later.

WCMP forwarding: With NS_c and NS_t, we calculate for each flow the weight change operations that update the network from NS_c to NS_t. We then create dependency edges between these operations based on the consistency property. Algorithm 2 shows how to do that for packet-coherence, using version numbers [85, 112]. In this approach, the ingress

Algorithm 2 Dependency graph for packet coherence in a WCMP network

- v_0: old version number
- v_1: new version number

1: **for** each flow f **do**
2: $s^* = GetIngressSwitch(f)$
3: $o^* = GenRuleModifyOp(s^*, v_1)$
4: **for** $s_i \in GetAllSwitches(f) - s^*$ **do**
5: **if** s_i has multiple next-hops **then**
6: $o_1 = GenRuleInsertOp(s_i, v_1)$
7: $o_2 = GenRuleDeleteOp(s_i, v_0)$
8: Add edge from o_1 to o^*
9: Add edge from o^* to o_2

switch tags each packet with a version number and downstream switches handle packets based on the embedded version number. This tagging ensures that each packet either uses the old configuration or the new configuration, and never a mix of the two. The algorithm generates three types of operations: i) the ingress switch tags packets with the new version number and uses new weights (Line 3); ii) downstream switches have rules for handling the packets with the new version number and new weights (Line 6); and iii) downstream switches delete rules for the old version number (Line 7). Packet coherence is guaranteed if Type i operation occurs after Type ii (Line 8) and Type iii operations occur after Type i (Line 9). Line 5 is an optimization; no changes are needed at switches that have only one next hop for the flow in both the old and new configurations.

We use the example in Figure 3.6 again to illustrate the algorithm above. For flow f, we need to update the flow weights at $S1$ from $[(S2, 1), (S4, 0)]$ to $[(S2, 0.5), (S4, 0.5)]$, and weights at $S2$ from $[(S3, 0.5), (S5, 0.5)]$ to $[(S3, 1), (S5, 0)]$. This translates to three operations (Table 3.3): add new weights with new version numbers at $S2$ (X), change to new weights and new version numbers at $S1$ (Y), and delete old weights at $S2$ (Z). We connect X to Y and Y to Z as shown in Figure 3.6(d).

Blackhole-freedom and loop-freedom do not require version numbers. For the former, we must ensure that every switch that may receive a packet from a flows always has a rule for it. For the latter, we must ensure that downstream switches (per new configuration) are updated before updating a switch to new rules [89]. These conditions are easy to en-

code in a dependency graph. For space constraints, we omit detailed description of graph construction.

Resource constraints: We introduce resource nodes to the graph corresponding to resources of interest, including link bandwidth and switch memory. These nodes are labeled with their current free amount or with infinity if that resource can never be a bottleneck.

We connect link bandwidth nodes with other nodes as follows. For each path node and bandwidth node for links along the path: if the traffic on the path increases, we add an edge from the bandwidth node to the path node with a label indicating the amount of traffic increase; if the traffic decreases, we add edges in the other direction. For a tunnel-based network, we add an edge from each path node on which traffic increases to the operation node that changes weight at the ingress switch with a label indicating the amount of traffic increase; similarly, we add an edge in the other direction if the traffic decreases. For a WCMP network, we add an edge from each path node on which traffic increases to each operation node that *adds weights with new versions* with a label indicating the amount of increase; similarly, we add an edge from the operation node that *changes weight at the ingress switch* to each path node on which traffic decreases with a label indicating the amount of decrease. This difference is due to that tunnels offer packet coherence by design, while WCMP networks need version numbers.

Connecting switch memory resource nodes with other nodes is straightforward. We add an edge from a resource node to an operation node if the operation consumes that switch memory with an weight indicating the amount of consumption; we add an edge from an operation node to a resource node if the operation releases that switch memory with an weight indicating the amount of release.

For example, in Figure 3.6(c) node D, which changes tunnel weights at $S1$, increases 5 units of traffic on p_3 which includes link $S1$-$S4$ and $S4$-$S5$, and decreases 5 units of traffic on p_2 which includes link $S1$-$S2$ and $S2$-$S5$. Node A that adds tunnel p_3 consumes 1 rule

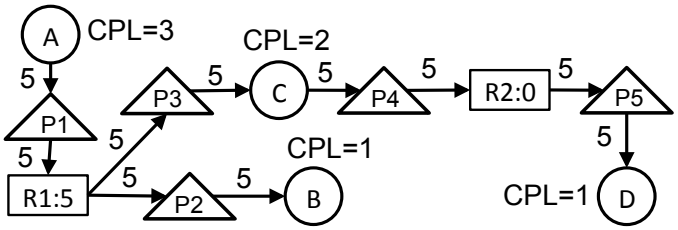


Figure 3.8: Critical-path scheduling. C has larger CPL than B, and is scheduled.

at $S1$. In Figure 3.6(d), we link p_3 to X and link X to Y. X and Y essentially takes the same effect as D in Figure 3.6(d).

Post-processing: After generating the dependency graph, we reduce it by deleting edges from non-bottlenecked resources. For each resource node R_i, we check the edges to its child nodes N_j. If the free resource $R_i.free$ is no smaller than $\sum_j l_{ij}$ where l_{ij} is the edge weight, we delete all the edges from R_i to its children and decrease the free capacity by $\sum_j l_{ij}$. The idea is that R_i has enough free resource to accommodate all operations that need it, so it's not a bottleneck resource and the scheduling will not consider it. For example, if $S1$-$S4$ has over 5 units of free capacity, we can delete the edge from $S1$-$S4$ to p_3 in Figures 3.6(c) and 3.6(d).

3.6 Dionysus Scheduling

We now describe how updates are scheduled in Dionysus. First, we discuss the hardness of the scheduling problem, which guided our approach. Then, we describe scheduling algorithm for the special case where the dependency graph is a DAG (directed acyclic graph). Finally, we extend this algorithm to handle cycles.

3.6.1 The Hardness of the Scheduling Problem

Scheduling is a resource allocation problem, that is, how to allocate available resources to operations to minimize the update time. For example, resource node $R1$ in Figure 3.8 has 5 units of free resource. It cannot cover both B and C. We must decide to schedule i)

B, $ii)$ C, or $iii)$ part of B and C. Every time we make a scheduling decision, we decide how to allocate a resource to its child operations and which parent operation to execute to obtain a resource. Additional constraints on scheduling are placed by dependencies between operations.

We can prove the following about network update scheduling.

Theorem 1. *In the presence of both link capacity and switch memory constraints, finding a feasible update schedule is NP-complete.*

Proof. Given a network, where all flow demands from a set of sources to a set of destinations must go through either switch u or v. Each switch has a memory limit for k rules (flows), and each switch has a bandwidth capacity limit of c. We have $2k-1$ flows, one big flow with capacity $c/2$, $k-1$ flows with capacity ϵ (think of $\epsilon = 0$), plus a set S of $k-1$ flows, all with integer capacity, in total c. Currently, the set S goes through switch v, all other flows (the big one and the tiny ones) go through switch u. The target state is to swap the switches of all flows, i.e. the big and the tiny flows should go through switch v, the set S through switch u. Note that both current and target solution are feasible regarding both capacity and memory. Initially, we cannot move any flow from v to u, not even partially, because the rule limit on switch u is already maxed out. So we can only (partially) move a single flow from u to v. If we move the big flow, we need a new rule on switch v, which will also max out the rule limit on switch v, at which point we are stuck as both memory limits are maxed out. However, we can move an ϵ flow from u to v, first creating an additional rule at v, then moving the flow, then removing one rule at u. At this stage have used all rules on v, but we have one spare rule at u, which gives us the possibility to (partially) move a flow from v to u. Again, partially moving a flow is not a good idea as we are maxing out regarding rules on both switches. However, there is enough spare capacity on switch u to completely move one of the flows in S. We do that, as it is the only thing we can do. We continuing moving ϵ-flows from u to v and then S S-flows from v to u. However, since we cannot move flows partially, we always must move complete flows, and

at some point, capacity on u will become a problem. In order to be able to move the big flow from u to v, we must have moved a subset S' of S from v to u such that this subset has exactly a total capacity $c/2$. In order to figure out the set S', we need to *partition* the flows into two equal-capacity sets. This is equivalent to the so-called partition problem, an NP-complete problem that must partition of set of n integers into two sets with the same sum. □

The hardness stems from the fact that memory constraints involve integers and memory cannot be allocated fractionally. Scheduling is simpler if we only have link capacity constraints, but finding the fastest schedule is still hard because of the huge search space.

Theorem 2. *In the presence of link capacity constraints, but no switch memory constraints, finding the fastest update schedule is NP-complete.*

Proof. We use the same network as above, i.e. all flow demands from a set of sources to a set of destinations must go through either switch u or v. Each switch has a bandwidth capacity limit of c. We have k flows, one with capacity $c/2$, plus a set S of $k-1$ flows, all with integer capacity, in total c. The big flow initially goes through switch u, the set S through switch v. Again, as above, we want to swap all flows. If we could solve partition, we would in a first step move a set S', subset of S with total capacity of $c/2$ from v to u, then the big flow from u to v, and finally all the other flows ($S \setminus S'$) from v to u. All flows are properly moved and touched only once. If we cannot solve partition, at least one flow must first be split (some part of the flow going through switch u while the other part going through switch v). Eventually this flow is properly moved as well, but in addition to touching each flow once, we need to touch at least one flow at least twice, which costs time. □

3.6.2 Scheduling DAGs

We first consider the special case of a DAG. Scheduling a DAG is, expectedly, simpler:

Lemma 3. *If the dependency graph is a DAG, finding a feasible update schedule is in P.*

While it is easy to find a feasible solution for a DAG, we want to find a fast one. Different scheduling orders lead to different finishing times. For example, if all operations take the same amount of time Figure 3.8, scheduling C before B will be faster.

We use *critical-path scheduling*. The intuition is that the critical path decides the completion time, and we thus want to schedule operations on the critical path first. Since resource nodes and path nodes in the dependency graph are only used to express constraints, we assign weight w=0 to them when calculating critical paths; for operation nodes, we assign weight w=1. With this, we calculate a critical-path length CPL for each node i as:

$$CPL_i = w_i + \max_{j \in children(i)} CPL_j \tag{3.1}$$

To calculate CPL for all the nodes in the graph, we first topologically sort all the nodes and then iterate over them to calculate CPL with Equation 3.1 in the reverse topological order. In Figure 3.8, for example, CPL_D=1, CPL_C=2, CPL_B=1, CPL_A=3. The CPL for each node can be computed efficiently in linear time.

Algorithm 3 shows how Dionysus uses CPL to schedule updates, with key notations summarized in Table 3.4. Each time we enter the scheduling phase, we first update the graph with finished operations and delete edges from unbottlenecked resources (line 2). Then, we calculate CPL for every node (Line 3) and sort nodes in decreasing order of CPL (Line 4). Then, we iterate over operation nodes and schedule them if their operation dependency and resource dependency are satisfied (Lines 6, 7). Finally, the scheduler waits for some time for all scheduled operations to finish before starting the next round (Line 10).

To simplify presentation, we first show the related pseudo code of $CanScheduleOperation(O_i)$ and $UpdateGraph(G)$ for tunnel-based networks and describe them below. Then, we briefly describe how the WCMP case differs.

Symbol	Description
O_i	Operation node i
R_j	Resource node j
$R_j.free$	Free capacity of R_j
P_k	Path node k
$P_k.committed$	Traffic that is moving away from path k
l_{ij}	Edge weight from node i to j

Table 3.4: Key notation in our algorithms.

Algorithm 3 ScheduleGraph(G)

```
1: while true do
2:     UpdateGraph(G)
3:     Calculate CPL for every node
4:     Sort nodes by CPL in decreasing order
5:     for unscheduled operation node O_i ∈ G do
6:         if CanScheduleOperation(O_i) then
7:             Schedule O_i
8:     Wait for time t or for all scheduled operations to finish
```

CanScheduleOperation (Algorithm 4): This function decides if an operation O_i is ready to be scheduled and updates the resource levels for resource and path nodes accordingly. If O_i is a tunnel addition operation, we can schedule it either if it has no parents (Lines 2, 3) or its parent resource node has enough free resource (Lines 4–8). If O_i is a tunnel deletion operation, we can schedule it if it has no parents (Lines 11–12); tunnel deletion operations do not have resource nodes as parents because they always release (memory) resources. If O_i is a weight change operation, we gather all free capacities on the paths where traffic increases and moves traffic to them (line 14-34). We iterate over each parent path node and obtain the available capacity ($available$) of the path (Lines 16–27). This capacity limits the amount of traffic that we can move to this path. We sum them up to $total$, which is the total traffic we can move for this flow (Line 26). Then, we iterate over child path nodes (Lines 30–33). Finally, we decrease $P_j.committed$ traffic on path represented by P_j (Line 31).

UpdateGraph (Algorithm 5): This function updates the graph before scheduling based on operations that successfully finished in the last round. We get all such operations and update related nodes in the graph (Lines 1–22). If the operation node adds a tunnel, we

Algorithm 4 CanScheduleOperation(O_i)

```
    // Add tunnel operation node
 1: if O_i.isAddTunnelOp() then
 2:     if O_i.hasNoParents() then
 3:         return true
 4:     R_j ← parent(O_i) // AddTunnelOp only has 1 parent
 5:     if R_j.free ≥ l_ji then
 6:         R_j.free ← R_j.free − l_ji
 7:         Delete edge R_j → O_i
 8:         return true
 9:     return false
    // Delete tunnel operation node
10: if O_i.isDelTunnelOp() then
11:     if O_i.hasNoParents() then
12:         return true
13:     return false
    // Change weight operation node
14: total ← 0
15: canSchedule ← false
16: for path node P_j ∈ parents(O_i) do
17:     available ← l_ji
18:     if P_j.hasOpParents() then
19:         available ← 0
20:     else
21:         for resource node R_k ∈ parents(P_j) do
22:             available ← min(available, l_kj, R_k.free)
23:         for resource node R_k ∈ parents(P_j) do
24:             l_kj ← l_kj − available
25:             R_k.free ← R_k.free − available
26:     total ← total + available
27:     l_ji ← l_ji − available
28: if total > 0 then
29:     canSchedule ← true
30: for path node P_j ∈ children(O_i) do
31:     P_j.committed ← min(l_ij, total)
32:     l_ij ← l_ij − P_j.committed
33:     total ← total − P_j.committed
34: return canSchedule
```

delete the node and its edges (Lines 2, 3). If the operation node deletes a tunnel, it frees rule space. So, we update the resource node (Lines 5, 6) and delete it (Line 7). If the operation node changes weight, for each child path node, we release resources to links on it (Lines 11–12) and delete the edge if all resources are released (Lines 13, 14). We reset

Algorithm 5 UpdateGraph(G)

1: **for** finished operation node $O_i \in G$ **do**
    // Finish add tunnel operation node
2: **if** $O_i.isAddTunneOp()$ **then**
3: Delete O_i and all its edges
    // Finish delete tunnel operation node
4: **else if** $O_i.isDelTunnelOp()$ **then**
5: $R_j \leftarrow child(O_i)$
6: $R_j.free \leftarrow R_j.free + l_{ij}$
7: Delete O_i and all its edges // DelTunnelOp only has 1 child
    // Finish change weight operation node
8: **else**
9: **for** path node $P_j \in children(O_i)$ **do**
10: **for** resource node $R_k \in children(P_j)$ **do**
11: $l_{jk} \leftarrow l_{jk} - P_j.committed$
12: $R_k.free \leftarrow R_k.free + P_j.committed$
13: **if** $l_{jk} = 0$ **then**
14: Delete edge $P_j \rightarrow R_k$
15: $P_j.committed \leftarrow 0$
16: **if** $l_{ij} = 0$ **then**
17: Delete P_j and its edges
18: **for** path node $P_j \in parents(O_i)$ **do**
19: **if** $l_{ji} = 0$ **then**
20: Delete P_j and its edges
21: **if** $O_i.hasNoParents()$ **then**
22: Delete O_i and its edges
23: **for** resource node $R_i \in G$ **do**
24: **if** $R_i.free \geq \sum_j l_{ij}$ **then**
25: $R_i.free \leftarrow R_i.free - \sum_j l_{ij}$
26: Delete all edges from R_i

the amount of traffic that is moving away from this path, $P_j.committed$, to 0 (Line 15). If we have moved all the traffic away from this path, we delete this path node (Lines 16, 17). Similarly, we check all the parent path nodes (Lines 18–20). If we have moved all the traffic into a path, we delete the path node (Lines 19, 20). Finally, if all parent path nodes are removed, the weight change for this flow finishes; we remove it from the graph (Line 22). After updating the graph with finished operations, we check all resource nodes (Lines 23–26). We delete edges from unbottlenecked resources (Lines 24–26).

Algorithm 6 CanScheduleOperation(O_i) — WCMP Network

```
1: if !O_i.isChangeWeightOp() then
2:     return false
3: canSchedule ← false
   // Check link capacity resource
4: total ← 0
5: O_{i_0} ← parents(O_i)[0]
6: for path node P_j ∈ parents(O_{i_0}) do
7:     P_j.available ← l_{ji_0}
8:     for resource node R_k ∈ parents(P_j) do
9:         P_j.available ← min(available, l_kj, R_k.free)
10:    total ← total + P_j.available
11: if total > 0 then
12:    canSchedule ← true
   // Check switch memory resource
13: for operation node O_j ∈ parents(O_i) do
14:    R_k ← resourceParent(O_j)
15:    if R_k ≠ null && R_k.free < l_kj then
16:        canSchedule ← false
17: if canSchedule then
       // Update link capacity resource
18:    O_{i_0} ← parents(O_i)[0]
19:    for path node P_j ∈ parents(O_{i_0}) do
20:        for resource node R_k ∈ parents(P_j) do
21:            l_kj ← l_kj − P_j.available
22:            R_k.free ← R_k.free − P_j.available
23:        for operation node O_k ∈ children(P_j) do
24:            l_jk ← l_jk − P_j.available
       // Update switch memory resource
25:    for operation node O_j ∈ parents(O_i) do
26:        R_k ← resourceParent(O_j)
27:        if R_k ≠ null then
28:            R_k.free ← R_k.free − l_kj
29: return canSchedule
```

WCMP network: Algorithms 4 and 5 for WCMP-based networks differ in two respects. First, WCMP networks do not have tunnel add or delete operations. Second, unlike tunnel-based networks that can simply change the weights at the ingress switches, WCMP networks perform a two-phase commit using version numbers to maintain packet coherence (node X and Y in Figure 3.6(d)). The code related to the weight change operation in the

Algorithm 7 UpdateGraph(G) — WCMP Network

```
1: for finished operation node O_i ∈ G do
2:     if O_i.isDelOldVerOp() then
3:         R_j ← child(O_i)
4:         R_j.free ← R_j.free + l_ij
5:     else if O_i.isChangeWeightOp() then
6:         for path node P_j ∈ children(O_i) do
7:             for resource node R_k ∈ children(P_j) do
8:                 l_jk ← l_jk − P_j.committed
9:                 R_k.free ← R_k.free + P_j.committed
10:                if l_jk = 0 then
11:                    Delete edge P_j → R_k
12:            P_j.committed ← 0
13:            if l_ij = 0 then
14:                Delete P_j and its edges
15:        if O_i.hasNoChildren() then
16:            Delete O_i, related nodes and edges
17: for resource node R_i ∈ G do
18:     if R_i.free ≥ Σ_j l_ij then
19:         R_i.free ← R_i.free − Σ_j l_ij
20:         Delete all edges from R_i
```

two algorithms has minor difference accordingly. Algorithm 6 and 7 show the pseudo code of $CanScheduleOperation(O_i)$ and $UpdateGraph(G)$ for WCMP networks.

CanScheduleOperation (Algorithm 6): Different from tunnel-based networks, WCMP networks don't have tunnel add or delete operation. Instead, every hop have weights to split a flow among multiple next-hops. To update a flow, all switches of this flow have to be touched to implement a two-phase update. Therefore, this function checks on a per-flow basis by examining the change weight operation at the ingress switch for every flow, e.g., Y in Figure 3.6(d) (Lines 1, 2). Similar to tunnel-based networks, it gathers all free capacities on the paths where traffic increases (Lines 4-12). It iterates over path nodes and obtain the available capacity ($P_j.available$) of the path (Lines 6-10). This capacity limits the amount of traffic that we can move to this path. Note that these path nodes are the parents of O_i's parents (e.g., parents of X rather than parents of Y in Figure 3.6(d)) since O_i is the change weight operation at the ingress switch (e.g., Y in Figure 3.6(d)). This flow can only be scheduled if there is any free capacity on these paths (Lines 11, 12). Then we

check all the switches to see if they have free memory to accommodate the operations that add weights with new version, e.g., S2 in Figure 3.6(d) (Lines 13-16). If we have both link and switch resource, we can schedule update to this flow (Lines 17-28). We update link resource (Lines 18-24) and switch resource (Lines 25-28) accordingly.

The function finally returns $canSchedule$ denoting whether the flow can be scheduled. In the schedule part (Line 7 in Algorithm 3), different from tunnel-based networks, we do a two-phase update, where we first add weights with new version (e.g., X in Figure 3.6(d)), change weights and assign new version at ingress switch (e.g., Y in Figure 3.6(d)) then delete weights with old version (e.g., Z in Figure 3.6(d)).

UpdateGraph (Algorithm 7): This function updates the dependency graph based on finished operations in the last round. We iterate over all finished operations (Lines 1-15). If the operation deletes weights with old version, we free rule space (Lines 2-4). If the operation changes weights with new version, for each child path node, we release resources to links on the path (Lines 8, 9) and delete the edge if all resources are released (Lines 10, 11). We reset $P_j.committed$ to 0 (Line 12) and delete it if all traffic to be moved has been moved (Lines 13-14). After this, we check whether O_i has any children left. If so, we keep these nodes in order to move the remaining traffic. Otherwise, it means all traffic has been moved, and we delete O_i and all the related two-phase commit nodes and edges (e.g., X, Y, Z , the related path nodes and edges in Figure 3.6(d)). Finally, we iterate over all resource nodes and remove edges from unbottlenecked resources (Lines 17-20).

3.6.3 Handling Cycles

Cycles in the dependency graph pose a challenge because inappropriate scheduling can lead to deadlocks where no progress can be made, as we saw for Figure 3.5(b) if $F2$ is moved first. Further, many cycles may intertwine together, which makes the problem even more complicated. For instance, A, B and C are involved in several cycles in Figure 3.9.

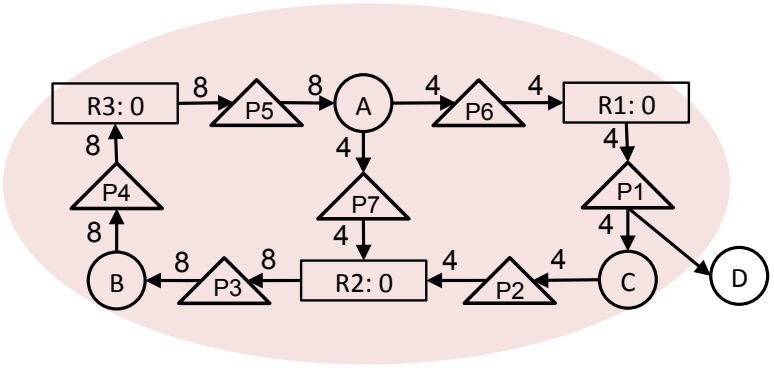


Figure 3.9: A deadlock example where the target state is valid but no feasible solution exists.

We handle dependency graphs with cycles by first transforming them into a virtual DAG and then using the DAG scheduling algorithm above. We use the concept of a strongly connected component (SCC), a subgraph where every node has a path to every other node [80]. One can think of an SCC as a set of intertwined cycles. If we view each SCC as a virtual node in the graph, then the graph becomes a virtual DAG, which is called the component graph in graph theory. We use Tarjan's algorithm [124] to efficiently find all SCCs in the dependency graph. Its time complexity is $O(|V| + |E|)$ where $|V|$ and $|E|$ are the number of nodes and edges.

With each SCC being a virtual node, we can use critical-path scheduling on the component graph. While calculating $CPLs$, we use the number of operation nodes in an SCC as the weight of the corresponding virtual node, which makes the scheduler prefer paths with larger SCCs.

We make two modifications to the scheduling algorithm to incorporate SCCs. The first is that the *for* loop at Line 5 in Algorithm 3 iterates over all nodes in the virtual graph. When a node is selected, if it is a single node, we directly call $CanScheduleOperation(O_i)$. If it is a virtual node, we iterate over the operation nodes in its SCC and call the functions accordingly. We use centrality [95] to decide the order of the iteration; the intuition is that a central node of an SCC is on many cycles, and if we can schedule this node early, many cycles will disappear and we can finish the SCC quickly. We use the popular outdegree-based

Algorithm 8 RateLimit(SCC, k^*)

```
1: O* ← weight change nodes ∈ SCC
2: for i=0 ; i¡k* && O*≠∅ ; i++ do
3:     O_i ← O*.pop()
4:     for path node P_j ∈ children(O_i) do
           // f_i is the corresponding flow of O_i
5:         Rate limit flow f_i by l_ij on path P_j
6:         for resource node R_k ∈ children(P_j) do
7:             R_k.free ← R_k.free + l_ij
8:         Delete P_j and its edges
```

definition of centrality, but other definitions may also be used. The second modification is that when path nodes consume link resources or tunnel add operations consume switch resources, they can only consume resources from nodes that either are in the same SCC or are independent nodes (not in any SCC). This heuristic prevents deadlocks caused by allocating resources to nodes outside the SCC ("Mv. F2") before nodes in the SCC are satisfied as in Figure 3.5(b).

Deadlocks: The scheduling algorithm resolves most cycles without deadlocks (§3.9). However, we may still encounter deadlocks in which no operations in the SCC can make any progress even if the SCC have obtained all resources from outside nodes. This can happen because (1) given the hardness of the problem, our scheduling algorithm, which is basically an informed heuristic, doesn't find the feasible solution among the combinatorially many orderings and gets stuck, or (2) there does not exist a feasible solution even if the target state is valid, like the example in Figure 3.9. One should note that deadlocks stem from the need for consistent network updates. Previous solutions face the same challenge but are much slower and cause more congestion than Dionysus (§3.9.4).

Our strategy for resolving deadlocks is to reduce flow rates (e.g., by informing rate limiters). Reducing flow rate frees up link capacity; and reducing it to zero on a path allows removal of the tunnel, which in turn frees up switch memory. Freeing up these resources allows some of the operations that were earlier blocked on resources to go through. In the extreme case, if we rate limit all the flows involved in the deadlocked SCC, the deadlock

can be resolved in one step. However, this extreme remedy leads to excessive throughput loss. It is also unnecessary because often rate limiting a few strategically selected flows suffices.

We thus rate limit a few flows to begin with, which enables some operations in the SCC to be scheduled. If that does not fully resolve the SCC, we rate limit a few more, until the SCC is fully resolved. The parameter k^* determines the maximum number of flows that we rate limit each time, and it controls the tradeoff between the time to resolve the deadlock and the amount of throughput loss. Algorithm 8 shows the procedure to resolve deadlocks for tunnel-based networks. It iterates over up to k^* weight change nodes in the SCC, each of which corresponds to a flow (Lines 2–8). The order of iteration is based on centrality value as above.

We use Figure 3.9 to illustrate deadlock resolution. Let k^*=1. The procedure first selects node A. It reduces 4 units of traffic on path $P6$ and 4 units on $P7$, which releases 4 units of free capacity to $R1$ and 4 units to $R2$, and deletes $P6$ and $P7$. At this point, node A has no children and thus does not belong to the SCC any more. After this, we call $ScheduleGraph(G)$ to continue the update. It schedules C, and partially schedules B (i.e., moves 4 units of traffic from path $P3$ to $P4$). After C finishes, it schedules the remainded of operation B and finishes the update. Finally, for node A and its corresponding flow f_A, we increase its rate on $P5$ as long as $R3$ receives free capacity released by $P4$.

We have the following theorem to prove that as long as the target state is valid (i.e., no resource is oversubscribed), we can fully resolve a deadlock using the procedure above.

Theorem 3. *If the target state is valid, a deadlock can be always resolved by calling* $RateLimit$ *a finite number of steps.*

Proof. Each time we call $RateLimit(SCC, k^*)$, the deadlock reduces by at least k^* number of operations. Let O^* be the number of operations in the deadlock. We can resolve the deadlock by at most $\lceil O^*/k^* \rceil$ iterations of $RateLimit$. □

We find experimentally that often the number of steps needed is a lot fewer than the bound above.

3.7 Implementation

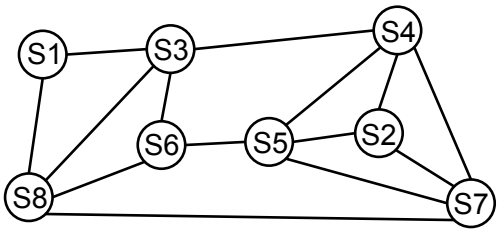
We have implemented a prototype of Dionysus with 5,000+ lines of C# code. It receives current state from the network and target state from applications as input, generates a dependency graph, and schedules rule updates. We implemented dependency graph generators for both tunnel-based and WCMP networks and all the scheduling algorithms discussed above. For accurate control plane confirmations of rule updates (not available in most OpenFlow agents today), we run a custom software agent on our switches.

3.8 Testbed Evaluation

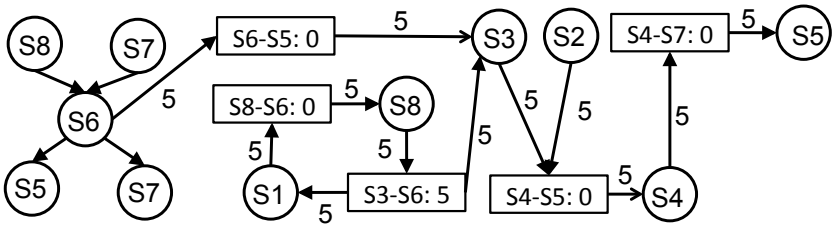
We evaluate Dionysus using testbed experiments in this section and using large-scale simulations in the next section. We use two update cases, a WAN TE case and a WAN failure recovery case. To show its benefits, we compare Dionysus against SWAN [55], a static solution.

Methodology: Our testbed consists of 8 Arista 7050T switches as shown in Figure 3.10(a). It emulates a WAN scenario. The switches are connected by 10 Gbps links. With the help of our switch agents, we log the time of sending updates and receiving confirmation. We use VLAN tags to implement tunnels and use prefix-splitting to implement weights when a flow uses multiple tunnels. We let $S2$ and $S4$ be straggler switches and inject 500 ms latency for rule updates on them. The remaining switches update at their own pace.

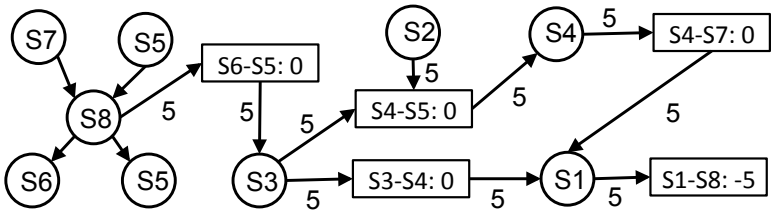
WAN TE case: In this experiment, the update is triggered by a traffic matrix change. TE calculates a new allocation for the new matrix, and we update the network accordingly. A simplified dependency graph for this update is shown in Figure 3.10(b). Numbers in



(a) Testbed topology



(b) Dependency graph for WAN traffic engineering case



(c) Dependency graph for WAN failure recovery case

Figure 3.10: Testbed setup. Path nodes are removed from the dependency graphs ((b) and (c)) for brevity.

the circles correspond to the switch to which the rule update is sent. For example, the operation node with annotation "S8" means a rule update at switch $S8$. The graph contains a cycle that includes nodes "S8", "S3-S6", "S1" and "S8-S6". Careless scheduling, e.g., one that schedules node "S3" before "S1" may cause a deadlock. There are also operation dependencies for this update: to move a flow at $S6$, we have to install a new tunnel at $S8$ and $S7$; after the movement finishes, we delete an old tunnel at $S5$ and $S7$.

Figure 3.11 shows the time series of this experiment. The x-axis is the time, and the y-axis is the switch index. A rectangle represents a rule update on a switch (y-axis) for some period (x-axis). Different rectangular patterns show different rule update operations (add rule, change rule, or delete rule). Rule updates on straggler switches, $S2$ and $S4$, take

longer than those on other switches. But even on non-straggler switches, the rule update time varies—the lengths of the rectangles are not identical—between 20 and 100 ms.

Dionysus dynamically performs the update as shown in Figure 3.11(a). First it finds the SCC and schedules node "S1". It also schedules "S2", "S8" and "S7" as they don't have any parents. After they finish, Dionysus schedules "S6" and "S8", then "S3", "S5" and "S7". Rather than waiting for "S2," which is a straggler, Dionysus schedules "S4" after "S3" finishes—"S3" releases enough capacity for it. Finally Dionysus schedules "S5". The update finishes in 842 ms.

SWAN uses a static, multi-step solution to perform the update (Figure 3.11(b)). It first installs the new tunnel (node "S8" and "S7"). Then, it adjusts tunnel weights with a congestion-free plan with the minimal number of steps, as follows:

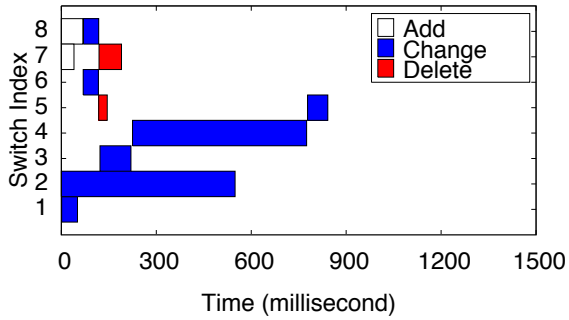
Step 1: "S1", "S6", "S2"

Step 2: "S4", "S8"

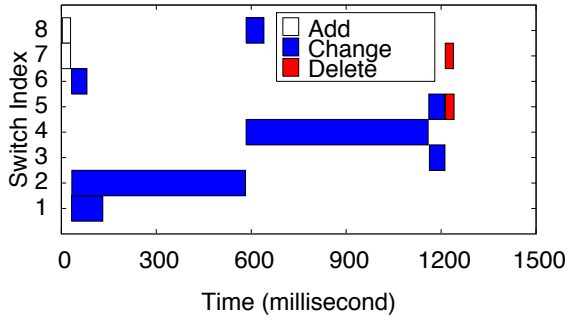
Step 3: "S3", "S5"

Due to stragglers $S2$ and $S4$, SWAN takes a long time on both Steps 1 and 2. Finally, SWAN deletes the old tunnel (node "S5" and "S7"). It does not start the tunnel addition and deletion steps with the weight change steps. The whole update takes 1241 ms, 47% longer than Dionysus.

WAN failure recovery case: In this experiment, the network update is triggered by a topology change. Link S3-S8 fails; flows that use this link rescale their traffic to other tunnels. This causes link S1-S8 to get overloaded by 50%. To address this problem, TE calculates a new traffic allocation that eliminates the link overload. The simplified dependency graph for this network update is shown in Figure 3.10(c). To eliminate the overload on link S1-S8, a flow at $S1$ is to be moved away, which depends on several other rule updates. Doing all the rule updates in one shot is undesirable as it may cause more link overloads and affect more flows. For example, if "S1" finishes faster than "S3" and "S4", then it causes 50% link overload on link S3-S4 and S4-S7 and unnecessarily brings congestions to flows on



(a) Dionysus



(b) SWAN

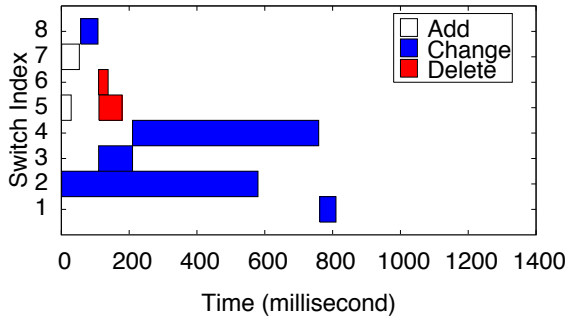
Figure 3.11: Time series for testbed experiment of WAN TE.

these links. We present extensive results in §3.9.3 to show that one-shot updates can cause excessive congestion.

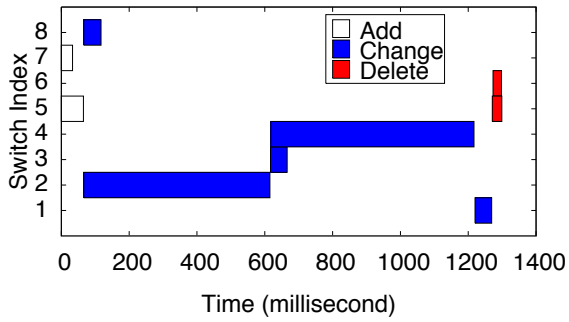
Figure 3.12(a) shows the time series of the update performed by Dionysus. It first schedules nodes "S7", "S5" and "S2". After "S7" and "S5" finish, a new tunnel is established and it safely schedules "S8". Then it schedules "S3", "S5" and "S6". Although "S2" is on a straggler switch and is delayed, Dionysus dynamically schedules "S4" once "S3" finishes. Finally, it schedules "S1". It finishes the update in 808 ms, which eliminates the overload on S1-S8, as shown in Figure 3.12(c).

Figure 3.12(b) shows the time series of the update performed by SWAN. It first installs the new tunnel (node "S7" and "S5"), then calculates an update plan with minimal steps as follows.

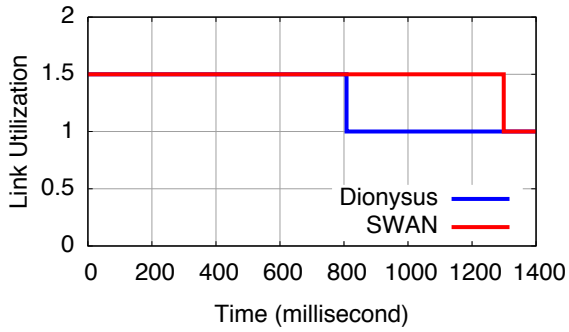
Step 1: node "S2", node "S8"



(a) Dionysus



(b) SWAN



(c) Link Utilization on Link $S1$-$S8$

Figure 3.12: Time series for testbed experiment of WAN failure recovery.

Step 2: node "S3", node "S4"

Step 3: node "S1"

This static plan does not adapt, and it is delayed by straggler switches at both Steps 1 and 2. It misses the opportunity to dynamically reorder rule updates. It takes 1299 ms to finish the update and eliminate the link overload, 61% longer than Dionysus.

3.9 Large-Scale Simulations

We now conduct large-scale simulations to show that Dionysus can significantly improve update speed, reduce congestion, and effectively handle cycles in dependency graphs. We focus on congestion freedom as the consistency property, a particularly challenging property and most relevant for the networks we study.

3.9.1 Datasets and Methodology

Wide area network: This dataset is from a large WAN that interconnects $O(50)$ sites. Inter-site links have tens to hundreds of Gbps capacity. We collect traffic logs on routers and aggregate them into site-to-site flows over 5-minute intervals. The flows are classified into 3 priorities: interactive, elastic and background [55]. We obtain 288 traffic matrices on a typical working day, where each traffic matrix consists of all the site-to-site flows in one interval.

The network uses tunnel-based routing, and we implement the TE algorithm of SWAN [55] which maximizes network throughput and approximates max-min fairness among flows of the same priorities. The TE algorithm produces the network configuration for successive intervals and we compute the time to update the network from one interval to the next.

Data center network: This dataset is from a large data center network with several hundred switches. The topoogy has 3 layers: ToR (Top-of-Rack), Agg (Aggeration), and Core. Links between switches are 10 Gbps. We collect traffic traces by logging the socket events on all servers and aggregate them into ToR-to-ToR flows over 5-minute intervals. As for the WAN, we obtain 288 traffic matrices for a typical working day.

Due to the large scale, we do elephant-flow routing [4, 15, 31]. We choose the 1500 largest flows, which account for 40–60% of all traffic. We use an LP to calculate their traffic allocation and use ECMP for other flows. This method improves the total throughput by up

to 30% as compared to using ECMP for all flows. We run TE and update WCMP weights for elephant flows every interval. Since mice flows use default ECMP entries, nothing is updated for them.

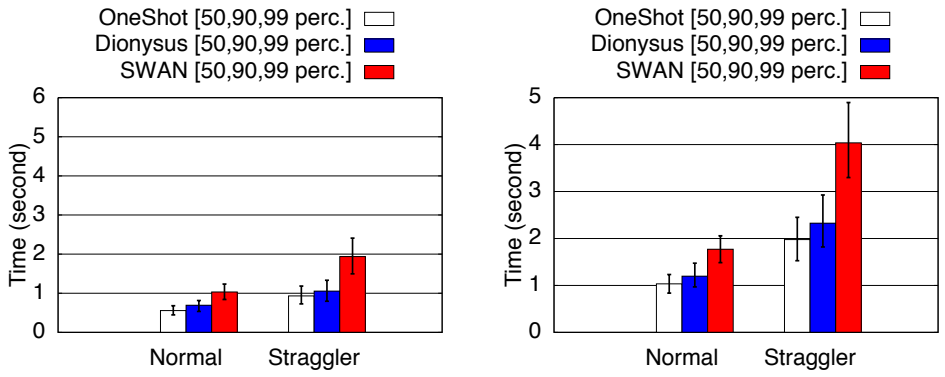
For both settings, we leave 10% scratch capacity on links to aid transitions [55], and we use 1500 as switch rule memory size. This memory size means that the memory slack (i.e., unused capacity) is at least 50% in our experiments in §3.9.2 and §3.9.3. In §3.9.4, we study the impact of memory limitation by reducing memory size.

Alternative approaches: We compare Dionysus with two alternative approaches. First, *OneShot* sends all updates in one shot. It does not maintain any consistency, but serves as the lower bound for update time. Second, *SWAN* is the state-of-the-art approach in maintaining congestion freedom [55]. It uses a heuristic to divide the update into multiple phases based on memory constraints so that each intermediate phase can fit all rules in switches. SWAN may rate limit flows in intermediate phase as the paths in the network cannot carry all the traffic. Between consecutive phases, it uses a linear program to calculate a congestion-free multi-step plan based on capacity constraints.

Rule update time: The rule update time at switches is based on switch measurement results (§3.2). We show results in both *normal* setting and *straggler* setting. In the former case, we use the median rule update time in §3.2; in the latter case, we draw rule update time from the CDF in §3.2. We use 50 ms as RTT in WAN scenario.

3.9.2 Update Time

WAN TE: Figure 3.13(a) shows the 50^{th}, 90^{th}, 99^{th} percentile update time across all intervals for the WAN TE scenario. Dionysus outperforms SWAN in both normal and straggler settings. In the normal setting, Dionysus is 57%, 49%, and 52% faster than SWAN in the 50^{th}, 90^{th}, 99^{th} percentile, respectively. The gain is mainly from pipelining: in every step, different switches receive different number of rules to update and thus takes different



(a) WAN TE (b) Data center TE

Figure 3.13: Dionysus is faster than SWAN and close to OneShot.

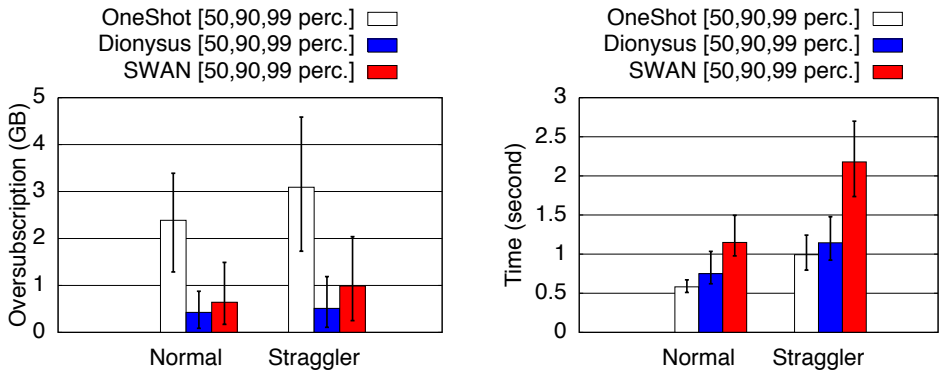
amount of time to finish. While SWAN has to wait for the switch with the most number of rules to finish, Dionysus begins to issue new operations as soon as some switches finish.

In the straggler setting, Dionysus reduces update time even more. It is 88%, 84%, and 81% faster than SWAN in the 50^{th}, 90^{th}, 99^{th} percentile, respectively. This advantage is because stragglers provide more opportunities for dynamic scheduling which SWAN cannot leverage. Dionysus also performs close to OneShot. It is only 25% and 13% slower than OneShot in the 90^{th} percentile in normal and straggler settings, respectively.

Data center TE: Figure 3.13(b) shows results for the data center TE scenario. Again, Dionysus significantly outperforms SWAN. In the normal setting, it is 53%, 48%, and 40% faster than SWAN in the 50^{th}, 90^{th}, and 99^{th} percentile; in the straggler setting, it is 81%, 74%, and 67% faster. Data center TE takes more time because it involves a two-phase commit across multiple switches for each flow; WAN TE only needs to update the ingress switch if all tunnels are established.

3.9.3 Link Oversubscription

We use a WAN failure recovery scenario to show that Dionysus can reduce link oversubscription and shorten recovery time. We use the same topology and traffic matrices as in the WAN TE case. For each traffic matrix, we first use TE to calculate a state NS_0. Then we



(a) Link Oversubscription

(b) Update time

Figure 3.14: In WAN failure recovery, Dionysus significantly reduces oversubscription and update time as compared to SWAN. OneShot, while fast, incurs huge oversubscription.

fail a randomly selected link, which causes the ingress switches to move traffic away from the failed tunnels to the remaining ones. For example, if flow f originally uses tunnels T_1, T_2 and T_3 with weights w_1, w_2 and w_3 and the failed link causes T_1 to break, then f carries its traffic using T_2 and T_3 with weights $w_2/(w_2 + w_3)$ and $w_3/(w_2 + w_3)$. We denote the network state that emerges after the failure and rescaling as NS_1. Since rescaling is a local action, NS_1 may have overloaded links. The TE calculates a new state NS_2 to eliminate congestion. The network update that we study is the update from NS_1 to NS_2.

If the initial state NS_1 already has congestion, there will be no congestion-free update plan. For Dionysus and SWAN, we generate plans in which, during updates, no oversubscribed link carries more load than its current load. In such plans, the capacity of congested links is virtually increased to its current load, to make each link appear non-congested. For Dionysus, we increase the weight of overloaded links to the overloaded amount in CPL calculation (Equation 3.1). Then, Dionysus will prefer operations that move traffic away from overloaded links. For SWAN, we use the linear program to compute the plan such that total oversubscription across all links is minimized at each step. Of all possible static plans, this modification makes SWAN prefer one that minimizes congestion quickly. OneShot operates as before because it does not care about congestion.

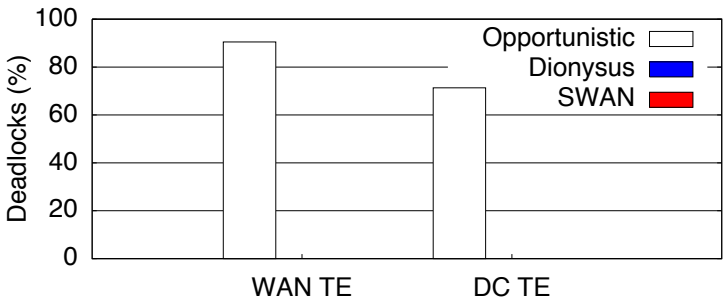


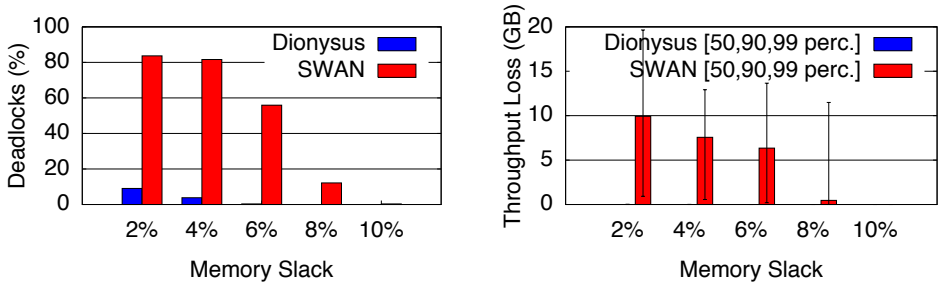
Figure 3.15: Opportunistic scheduling frequently deadlocks. Dionysus and SWAN have no deadlocks.

Figure 3.14 shows the update time and link oversubscription—the amount of data above capacity arriving at a link. Dionysus has the least oversubscription among the three. OneShot, while quick, has huge oversubscription. SWAN incurs 1.49 GB and 2.04 GB oversubscription in the 99^{th} percentile in normal and straggler settings, respectively. As even high-end switches today only have hundreds of MB buffer [11], such oversubscription will cause heavy packet loss. Dionysus reduces oversubscription to 0.88 GB and 1.19 GB, which are 41% and 42% less than SWAN. For update time, Dionysus is 45% and 82% faster than SWAN in the 99^{th} percentile in normal and straggler setting, respectively.

3.9.4 Deadlocks

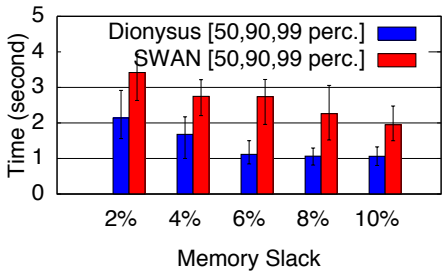
We now study the effectiveness of Dionysus in handling circular dependencies, which can lead to deadlocks. First, we show that, as mentioned in §3.3, completely opportunistic scheduling can lead to frequent deadlocks even in a setting that is not resource-constrained. Then, we show the effectiveness of Dionysus in handling resource-constrained settings.

Figure 3.15 shows the percentage of network updates finished by Dionysus, SWAN, and an opportunistic approach without deadlocks, that is, without having to reduce flow rates during updates. The opportunistic approach immediately issues any updates that do not violate consistency (§3.3), instead of planning using a dependency graph. The data in the figure corresponds to the WAN and data center TE scenarios in §3.9.2, where the



(a) Percentage of rate limited cases

(b) Throughput Loss



(c) Update Time

Figure 3.16: Dionysus only occasionally runs into deadlocks and uses rate limiting, and experiences little throughput loss. It also consistently outperforms SWAN in update time.

memory slack was over 50%. We do not show results for OneShot; it does not deadlock by design as it does not worry about consistency.

We see that planning-based approaches, Dionysus and SWAN, lead to no deadlocks, but the opportunistic approach deadlocks 90% of the time for WAN TE and 70% of the time for data center TE. It performs worse for WAN TE because the WAN topology is less regular than the data center topology, which leads to more complex dependencies.

We now evaluate Dionysus and SWAN in resource-constrained settings. To emulate such a setting, instead of using 1500 as memory size, we vary switch memory slack; 10% memory slack means we set the memory size as 1100 when the switch is loaded with 1000 rules. We show three metrics in the WAN TE setup: (1) the percentage of cases that deadlock and use rate limiting to finish the update, (2) the throughput loss caused by rate limiting (i.e., the product of the limited rate and the rate limited time), and (3) the update time. We set k^*=5 in Algorithm 8 for Dionysus.

Figure 3.16 shows the results for the straggler setting. The results with the normal setting are similar. Figure 3.16(a) shows the percentage of cases that use rate limiting under different levels of memory slack. Dionysus only occasionally runs into deadlocks and uses resorts to rate limiting more sparingly than SWAN. Even with only 2% memory slack, Dionysus uses rate limiting in fewer than 10% cases. SWAN, on the other hand, uses rate limiting in more than 80% of the cases. This difference is because the heuristics in Dionysus strategically account for dependencies during scheduling. SWAN uses simplistic metrics, such as the amount of traffic that a tunnel carries and the number of hops of the tunnel, to decide which tunnel to add or delete.

Figure 3.16(b) shows the throughput loss. The throughput loss with SWAN can be as high as 20 GB, while that with Dionysus is only tens of MB. Finally 3.16(c) shows the update time. Dionysus is 60%, 145%, and 84% faster than SWAN in the 90^{th} percentile under 2%, 6% and 10% memory slack respectively.

3.10 Related Work

In the domain of distributed protocols, there is a lot of work on avoiding transient misbehavior during network updates. Much focuses on maintain properties like loop-freedom for specific protocols or scenarios. For example, Francois *etal.* [41], John *etal.* [66] and Kushman *etal.* [77] focus on BGP, Francois *etal.* [42, 40] and Raza *etal.* [111] focus on link-state protocols, and Vanbever *etal.* [128] focus on migration from one routing protocol to another.

With the advent of SDN, many recent works propose solutions to maintain different consistency properties during network updates. Reitblatt *etal.* [112] provide a theoretical foundation and propose a two-phase commit protocol to maintain packet coherence. Katta *etal.* [72] and McGeer *etal.* [90] propose solutions to reduce the memory requirements to maintain packet coherence. SWAN [55], zUpdate [85] and Ghorbani and Caesar [45]

provide solutions for congestion-free updates. Noyes *etal.* [97] propose a model checking based approach to generate update orderings that maintain invariants specified by the operator. Mahajan and Wattenhofer [89] present an efficient solution for maintaining loop freedom. As mentioned earlier, unlike these works, the key characteristic of our approach is dynamic scheduling, which leads to faster updates.

Mahajan and Wattenhofer [89] also analyze the nature of dependencies among switches induced by different consistency properties and outline a general architecture for consistent updates. We build on their work by developing a concrete system.

Petri net [134] is a model to describe distributed systems. At a high level, the dependency graph in Dionysus is similar to Petri net in the sense that they both intends to capture state transitions and resource consumptions. But the details on how the graph is constructed and used are different between the two approaches. Petri net only has two types of nodes, i.e., place nodes and transition nodes. The dependency graph in Dionysus is tailored to network forwarding rule updates. It contains three types of nodes, i.e., operation nodes, resource nodes, and path nodes. We incorporate domain-specific knowledge to define how these nodes are connected and how the resources are consumed in the dependency graph.

Some works develop approaches that spread traffic such that the network stays congestion-free after a class of common failures [132, 83], and thus no network-wide updates are needed to react to these failures. These approaches are complementary to our work. They help reduce the number of network updates needed. But network updates are still be needed to adjust to changing traffic demands and reacting to failures that are not handled by these approaches. Dionysus ensures that these updates will be fast and consistent.

3.11 Conclusion

Dionysus enables fast, consistent network updates in SDNs. The key to its speed is dynamic scheduling of updates at individual switches based on runtime differences in their update speeds. We showed using testbed experiments and data-driven simulations that Dionysus improves the median network update speed by 53%-88% over static scheduling. These faster updates translates to a more nimble network that reacts faster to events like failures and changes in traffic demand.

Chapter 4

Owan: Dynamic Topology Reconfiguration

This chapter focuses on supporting the joint management of the optical and network layers. Traditionally, ISPs manage the optical layer and the network layer separately. Advancements in software-defined networking and optical hardware make it feasible to build centralized systems to dynamically reconfigure optical devices in the optical layer together with switches in the network layer. We use bulk transfer as a specific application to show the benefits of the joint control. Bulk transfer on the wide-area network is a fundamental service to many globally-distributed applications. It is challenging to efficiently utilize expensive WAN bandwidth to achieve short transfer completion time and meet mission-critical deadlines. This chapter presents Owan, a novel traffic management system that optimizes wide-area bulk transfers with centralized joint control of the optical and network layers. Owan can dynamically change the network-layer topology by reconfiguring the optical devices. We develop efficient algorithms to jointly optimize optical circuit setup, routing and rate allocation, and dynamically adapt them to traffic demand changes. We have built a prototype of Owan with commodity optical and electrical hardware. Testbed experiments and large-scale simulations on two ISP topologies and one inter-DC topology

show that Owan completes transfers up to $4.45\times$ faster on average, and up to $1.36\times$ more transfers meet their deadlines, as compared to prior methods that only control the network layer.

4.1 Introduction

Many globally-distributed applications have bulk data to transfer over the wide-area network (WAN). For example, search engines need to synchronize search indexes between data centers; financial institutions need to backup everyday transactions over remote sites; media companies need to deliver high-definition video content to multiple distribution areas. Bulk transfers have large size (terabytes to petabytes) and account for a big proportion of traffic, e.g., 85–95% for some inter-datacenter (inter-DC) WANs [58, 55, 69, 140].

Optimizing bulk transfers is important to network operators. Although bulk transfers are not as delay-sensitive as interactive traffic like web queries, it is beneficial and sometimes necessary to finish them quickly, as it is essential for service quality. For instance, the time to finish search index synchronization directly impacts the search quality [69]. Furthermore, some bulk transfers are associated with deadlines, e.g., timely delivery of high-definition video content to some cities by a certain time is the key for business success [69, 140]. It requires network operators to carefully schedule these transfers in order to meet their deadlines.

Existing practice performs traffic engineering (TE) in the network layer. Traditional WAN designs over-provision the network with 30–40% average network utilization, in order to handle traffic demand changes and failures [58]. Recent designs like Google B4 and Microsoft SWAN leverage software-defined networking (SDN) to directly control the network with a global view [58, 55, 69, 140]. They use a global TE to dynamically change routing and rate allocation, so that they can accommodate more traffic and meet more deadlines. They all assume a fixed network-layer topology.

(a) Internet2 physical infrastructure. (b) Internet2 IP layer topology.

Figure 4.1: WAN infrastructure example (Internet2 [57]).

In a modern WAN, the network-layer topology is constructed over an intelligent optical layer.[1] By reconfiguring the optical devices, the operator can dynamically change the network-layer topology. Figure 4.1 shows an example of a modern WAN infrastructure—the Internet2 network [57]. The network-layer link between SEA and LA in Figure 4.1(b) is implemented by an optical circuit that traverses multiple optical switches in the optical layer in Figure 4.1(a). In practice, a WAN router is connected to an optical switch called Reconfigurable Optical Add-Drop Multiplexer (ROADM) via short-reach wavelength. To connect two WAN router ports, the operator needs to properly configure the ROADMs along the path to establish an optical circuit. By changing the circuits in the optical layer, operators can change which two router ports are connected.

Traditionally, the optical layer is reconfigured on a long time scale, e.g., weeks to months, or even years. The major reason is the labor and risk involved in the reconfiguration: operators need to deal with sophisticated configurations, including IP, BGP and access control list (ACL), and they have to perform operations on many routers without consistent configuration interfaces, which is tedious and error-prone. Also, after a optical layer reconfiguration, traditional distributed routing protocols may be slow to converge.

In this chapter, we present Owan, a new traffic management system that optimizes wide-area bulk transfers with centralized joint control of the optical and network layers. We leverage two technology trends. The first is SDN that allows direct control of network

[1] A WAN network is a packet-switched network, which is usually built on top of an optical network. In this chapter, the WAN network is referred as the network layer, and the optical network is referred as the optical layer.

devices and simplifies network management; the second is modern ROADM devices that allow fast remote reconfigurations (e.g., provisioning a circuit in tens to hundreds of milliseconds [27]). Owan orchestrates bulk transfers in a centralized manner. It computes and implements the optical circuit configuration (the optical circuits that implement the network-layer topology) and the routing configuration (the paths and rate allocation for each transfer) to optimize the transfer completion time or the number of transfers that meet their deadlines.

A major technical challenge for Owan is that the optimization problem includes a large number of constraints, some of which are integral. Most TE algorithms assume a given topology and only compute the network-layer configuration [58, 55, 69, 140]. While there is research on reconfigurable optics, these projects focus on data center networks under the assumption of specific optical devices (e.g., MEMS switches) and certain topologies [35, 129, 25, 50, 26]. However, there are three unique constraints on WANs that do not present in data centers: ROADMs, regenerators and arbitrary topology. We accommodate ROADMs in our formulation which are typically used as building blocks for WANs. We take into account *regenerators*, which regenerates optical signals after certain distance. Also, we do not make any assumptions of the optical-layer topology, allowing it to be irregular.

The key idea to solve the optimization problem is to do a probabilistic search in the search space with simulated annealing. At each time slot, we use the current topology as the starting point, and use simulated annealing to find a topology with the highest total throughput. There are two major benefits. First, searching for a topology, instead of the entire optical and routing configurations, substantially reduces the search space. Second, using the current topology as the starting point in simulated annealing allows us to find a target topology that is close to the current topology but has higher throughput. This significantly reduces the number of changes we need to make in the optical layer, in order to update the topology.

We build a Owan prototype using commodity optical and electronic hardware. The prototype has nine sites and emulates the Internet2 topology in Figure 4.1. We conduct extensive evaluations through both testbed experiments with our prototype that emulates the Internet2 network and large-scale simulations with data from an ISP network and an inter-DC network. Our results show that Owan improves the transfer completion time by up to $4.45\times$ on average and $3.84\times$ at the 95th percentile, as compared to prior methods that only control the network layer. Furthermore, Owan allows up to $1.36\times$ more transfers to meet their deadlines and up to $2.03\times$ more bytes to finish before their deadlines.

4.2 Background and Motivation

We focus on bulk transfers on the WAN. Our design applies to both private WANs (e.g., inter-DC WANs) and public WANs (e.g., provided by ISPs). Large ISPs usually own both the public WAN and the underlying optical network. They can directly use Owan to manage their networks. Small ISPs and private WANs usually lease optical circuits from optical-network providers. In such case, they would need an interface with the optical-network operator to change the optical configurations together with Owan. Furthermore, Owan requires to know the traffic demand and to control the rate of each transfer, which can be assumed for inter-DC networks but not for ISP networks. To use Owan in ISP networks, ISPs can provide a bulk transfer service to their clients. This service has an interface for clients to submit transfer requests that contain traffic demand information and inform clients data rates they can use for their transfers. Before we introduce Owan, we give some background on WAN infrastructure and a motivating example to show the benefits of joint optimization of the optical and network layers.

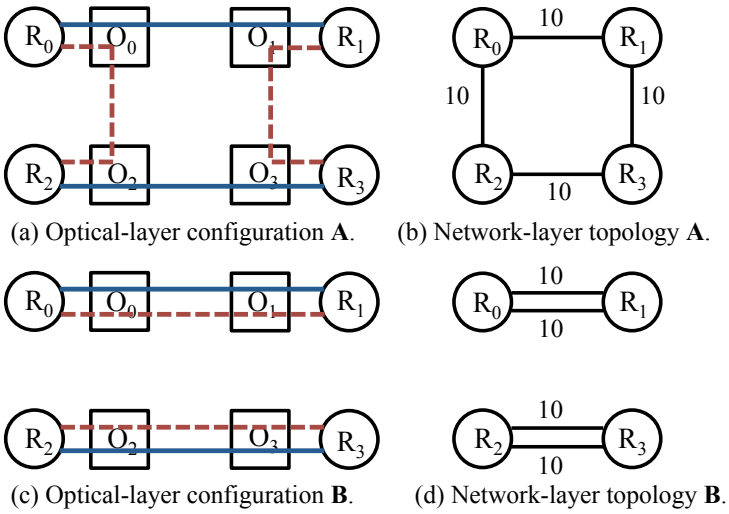


Figure 4.2: Example of topology reconfiguration. Different line types/colors in (a) and (c) denote different wavelengths. A router port or a wavelength carries 10 bandwidth units. By reconfiguring how wavelengths are switched in ROADMs (rectangle nodes), we can change how routers (circle nodes) are connected. (b) and (d) show the resulting network-layer topologies.

4.2.1 Background on WAN Infrastructure

A typical WAN infrastructure consists of network routers, optical devices, and fibers. A bulk transfer enters a WAN on a router from an access network (e.g., a data-center network or a metro network) or other autonomous systems, passes through intermediate routers to the destination router, and leaves the network. Since a WAN link is a circuit in the optical layer, packets over any WAN link actually traverse multiple optical switches in the form of optical signals.

Optical layer: A modern optical network consists of ROADMs connected by fiber pairs. Today's commercial ROADM technology is able to support 80 or more wavelengths per fiber pair and 40 Gbps (100 Gbps, and higher with high-order modulations and digital coherent receivers) per wavelength, which leads to 3.2 Tbps (8 Tbps, and even higher capacity) per fiber pair. A router port can connect to a ROADM port with a tunable optical transponder via standard short-reach wavelength. The tunable optical transponder is able to tune the standard wavelength to another specific wavelength. The ROADM can switch the

wavelength to an output port or an add/drop port connected to another router port. Commercial ROADMs can be reconfigured in hundreds of milliseconds and future ROADMs can reduce the time to tens of milliseconds and even lower [27, 56, 98].

Due to optical signal loss and some non-linear impacts on optical signals, a wavelength normally has limited transmission range, which is called *optical reach*. When an optical signal transmits beyond the optical reach, a regenerator device is required to regenerate the signal. In order to dynamically establish optical circuits on demand, operators usually pre-deploy some regenerators at certain concentration sites such that between any two ROADMs, there is at least one path using those limited regenerator concentration sites to satisfy the optical reach constraint [138, 14].

Network layer: A router is usually co-located with a ROADM. Customer-facing router ports are connected to customer equipment, such as data-center routers or metro-network routers; network-facing router ports are connected to ROADM ports via standard short-reach wavelength. A network-layer link is implemented by an optical circuit. By reconfiguring the optical layer, we can change the connectivity of router ports in the network layer, i.e., the network-layer topology.

Topology reconfiguration example: We use the example in Figure 4.2 to illustrate how the network-layer topology can be reconfigured with optical devices. In the network, we have four routers R_0-R_3 and four ROADMs O_0-O_3. Each router has two WAN-facing ports. In configuration **A**, each ROADM converts electrical packets from two router ports to different wavelengths and sends them to different ROADMs. For example, O_0 sends the solid/blue wavelength to O_1 and the dashed/red wavelength to O_2 (Figure 4.2(a)). In the resulting network-layer topology, each router is connected to two other routers (Figure 4.2(b)). In configuration **B**, a ROADM multiplexes two wavelengths on to the same fiber and is connected to only one other ROADM. For example, both wavelengths at O_0 are multiplexed to the fiber to O_1, with the fiber between O_0 and O_2 carrying no wavelengths (Figure 4.2(c)). In the network-layer topology, a router has both ports connected to an-

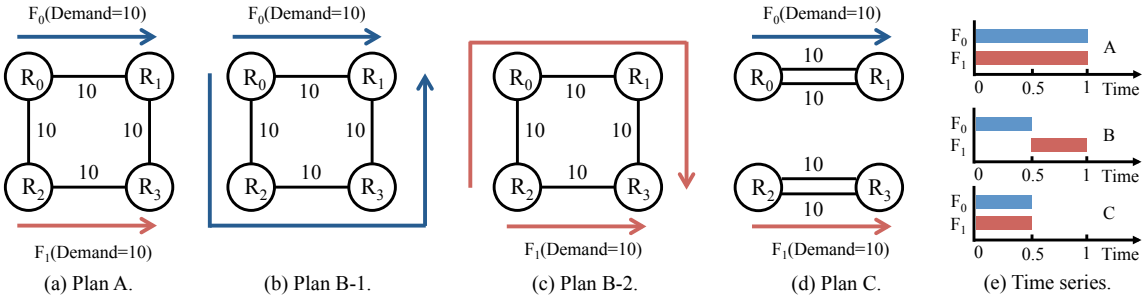


Figure 4.3: Example of optimizing bulk transfers. (a) Plan A transmits F_0 and F_1 simultaneously. (b-c) Plan B first transmits F_0 and then F_1. (d) Plan C reconfigures the topology and has the lowest average transfer completion time. (e) Time series to show the transfer completions of these three plans.

other router (Figure 4.2(d)), doubling the capacity between R_0 and R_1 from configuration **A** (Figure 4.2(b)).

4.2.2 Motivating Example

Topology reconfiguration opens a new opportunity for optimizing bulk transfers. Existing approaches assume a given and fixed network-layer topology, and optimizes bulk transfers by controlling the routing and/or the rate of each transfer [58, 55, 69, 140]. We provide a motivating example to show that by reconfiguring the topology we can significantly reduce average transfer completion time (Figure 4.3).

In the example, we have four routers R_0-R_3 similar to Figure 4.2. We only show the network-layer topology and omit the ROADMs for brevity. We have two transfers, F_0 and F_1. Each transfer has 10 units of traffic to send. Plan A only controls routing (Figure 4.3(a)). It uses the shortest paths and the two transfers are transmitted simultaneously. The average transfer completion time is 1 time unit.

We can do better if we can control the sending rates too. Plan B (Figure 4.3(b-c)) schedules F_0 first with two paths, R_0-R_1 and R_0-R_2-R_3-R_1, and let F_0 wait. It takes 0.5 time unit for F_0 to finish. Then F_1 starts and takes another 0.5 time unit to finish. The average transfer completion time is $\frac{0.5+1}{2} = 0.75$ time unit, or $1.33\times$ faster than Plan A.

Note that both Plan A and B waste available network capacity, in different ways. Plan A leaves bandwidth unused while Plan B has two-hop routing paths. We can do better if we control the network-layer topology. Plan C reconfigures the topology (Figure 4.3(d)). Two router ports on R_0 are all connected to R_1. Now both F_0 and F_1 can enjoy a bandwidth of 20 units and finish within 0.5 time unit. Plan C is $2\times$ faster than Plan A, and $1.5\times$ faster than Plan B.

4.3 Owan Design

In this section, we first provide an overview of Owan. Then, we present the algorithms to compute the optical and routing configurations to optimize bulk transfers. Finally, we describe how to deal with updates and some practical issues.

4.3.1 Owan Overview

Owan is a centralized system that orchestrates bulk transfers on the WAN. Figure 4.4 shows the system architecture. Abstractly, Owan works as follows.

1. Clients submit bulk transfer requests to the controller. A request is a tuple $(src_i, dst_i, size_i, deadline_i)$ that denotes the source, destination, size, and deadline (optional) of transfer request i.

2. The controller has a global view of the physical topology and transfer requests. It computes the optical circuits that build the network-layer topology, the paths and the sending rates for transfers.

3. The controller sends the rate allocation to each client and clients enforce rates on their applications. The controller directly programs routers and ROADMs to set up routing paths and optical circuits. On public WANs, the controller also needs to enforce rates

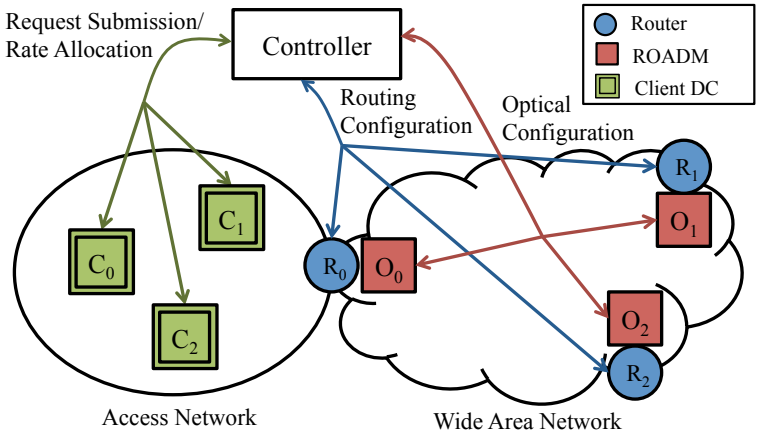


Figure 4.4: System architecture.

with rate limiters on routers in case clients do not properly enforce these rate limits on their applications.

The above process is an online process. We divide time into time slots. A time slot is much longer than the time to reconfigure the network and adjust sending rates, i.e., a few minutes vs. hundreds or thousands of milliseconds. The major job for the controller is to compute the configurations at each time slot to optimize bulk transfers.

4.3.2 Computing Network State

All the configurations are denoted as network state. We precisely define the network state and formulate the problem as follows. Table 4.1 summarizes the key notations.

Network state: A WAN is represented as a graph $G = (V, E)$ where V is the set of all sites and E is the set of links in the network-layer topology. A site v consists of one ROADM, a set of pre-deployed regenerators (could be zero), and zero or one router.

A network state NS is a configuration of the devices in the WAN. It includes the optical configuration OC and the routing configuration RC. OC is the set of optical circuits to be configured on the optical devices, which builds the network-layer topology. A network-layer link between u and v is implemented by a circuit oc_{uv} in the optical layer. RC is the set of routing configurations to be installed on routers (and end hosts if rates are enforced

Symbol	Description
V	The set of sites.
E	The set of network-layer links.
G	The network-layer topology.
F	The set of transfers.
NS	The network state $NS = (OC, RC)$.
OC	The set of optical circuits.
RC	The routing configuration $RC = \{rc_f \| f \in F\}$.
p	A routing path.
$r_{f,p}$	The rate of transfer f on routing path p.
rc_f	The routing configuration of f: $\{r_{f,p} \| p \in P_f\}$.
θ	The capacity of a wavelength.

Table 4.1: Key notations in problem formulation.

by clients). Specifically, the routing configuration of a transfer f, denoted by rc_f, includes its routing paths and rate limits for each path.

Problem formulation: The problem of finding the optimal network state is an online optimization problem. There are a stream of new transfers arriving at the system. At each time slot, we need to compute a network state NS that optimizes the average transfer completion time or the number of transfers that meet their deadlines. The problem has the following constraints.

1. The number of router ports connected to ROADM ports at each site v is limited, denoted by fp_v. This constrains the total ingress and egress capacity of the router in the network-layer topology.

2. A wavelength can traverse at most distance η before it needs to be regenerated. If an optical circuit is longer than η, it has to use regenerators on its path to regenerate the signal.

3. The number of regenerators at each site v is limited, denoted by rg_v. A regenerator can regenerate one optical circuit and transform the circuit to a different wavelength if needed.

4. The optical circuits in the same fiber have to use different wavelengths. A fiber can carry at most ϕ different wavelengths and each wavelength can support a capacity of θ.

5. The total rate of transfers on a network-layer link cannot exceed its capacity θ.

As an additional consideration, we want to keep the changes to the network *incremental*, i.e., only updating a small number of optical links when we perform an update. This minimizes the disturbance during the network update process.

Algorithm overview: The problem has a large number of constraints and variables. Some constraints, like the number of router ports at each site, the number of regenerators at each site, and the number of wavelengths on each fiber, are integral. Even if the network-layer topology is given, optimizing for average transfer completion time is NP-hard [8].

A naive approach is to separately optimize the optical layer and the network layer. However, as the routing decisions are highly coupled with the underlying optical configuration, this greedy approach does not yield good performance results, as we will show in §4.5.4.

Instead, we use simulated annealing [74] to search for an approximate solution. The motivation for using simulated annealing is that we have a huge search space with integral variables. Simulated annealing is effective in finding acceptable local optimums in a reasonable amount of time while finding the global optimum is computationally expensive. Furthermore, the potential loss of using local optimums is compensated by the fact that the traffic demand changes over time and we frequently reconfigure the network to adapt to the traffic demand changes.

At a high level, we use the network-layer topology G as the state in simulated annealing. We use the current topology as the initial state and probabilistically jump to a neighbor state in each iteration, aiming to find a topology with the highest total throughput. To minimize changes to the network, we construct neighbor states by randomly changing four links of the current state, which is the minimal number of links to change to satisfy the port number constraints.

Algorithm 9 Compute Next Network State (Main Routine)

function $ComputeNetworkState(G)$
  $s_{current} \leftarrow G$
  $e_{current} \leftarrow ComputeEnergy(s)$
  $T \leftarrow e_{current}$
  $s^* \leftarrow s_{current}$
  $e^* \leftarrow e_{current}$
  while $T > \epsilon$ **do**
    $s_{neighbor} \leftarrow ComputeNeighbor(s_{current})$
    $e_{neighbor} \leftarrow ComputeEnergy(s_{neighbor})$
    if $e_{neighbor} > e^*$ **then**
      $s^* \leftarrow s_{neighbor}$
      $e^* \leftarrow e_{neighbor}$
    if $P(e_{current}, e_{neighbor}, T) > Rand(0, 1)$ **then**
      $s_{current} \leftarrow s_{neighbor}$
      $e_{current} \leftarrow e_{neighbor}$
    $T \leftarrow T \times \alpha$
  return s^*

Algorithm 10 Generate A Random Neighbor State

function $ComputeNeighbor(s)$
  $l_{uv}, l_{pq} \leftarrow RandomlySelectTwoEdges(E_l)$
  $l_{uv}.capacity \leftarrow l_{uv}.capacity - \theta$
  $l_{pq}.capacity \leftarrow l_{pq}.capacity - \theta$
  $l_{up}.capacity \leftarrow l_{up}.capacity + \theta$
  $l_{vq}.cacacity \leftarrow l_{vq}.capacity + \theta$
  return s

Our approach has two benefits. First, using the network-layer topology G as the state in simulated annealing, instead of the entire network state NS, significantly reduces the search space. If we search for NS, we have to decide both the optical circuits, the routing paths, and the rate assignments for the network. Instead, if we search for G, we only need to decide the links in the network-layer topology. Second, by using the current topology as the initial state, we are likely to end up with a topology that is not very different from the current one. This reduces the number of changes we need to make for network updates. Now we describe the algorithms in detail.

Simulated Annealing (Algorithm 9): The algorithm uses the current topology G as the initial state and the current throughput as the initial temperature (line 2-3). s^* is used to store the topology with the highest throughput and e^* is the energy (throughput) of s^*. The algorithm searches in the search space (line 7-16) until temperature T is less than an epsilon value. T is decreased by a factor of α in every iteration. At each iteration, it uses $ComputeNeighbor$ subroutine to find a neighbor state of the current state and uses $ComputeEnergy$ to compute the energy of the neighbor state. If the neighbor state has a higher energy than s^*, it updates s^* (line 10-12). The algorithm uses a probabilistic function P to decide whether to transition from the current state to the neighbor state. The probabilistic function P is defined as follows: if the neighbor state has a higher energy than the current state, the probability is 1; otherwise, the probability is $e^{(e_{current}-e_{neighbor})/T}$.

ComputeNeighbor (Algorithm 10): This subroutine finds a neighbor state of the current state. It first randomly selects two links from E, say e_{uv}, e_{pq}. Then it decreases the capacity of the selected two links by θ while increasing the capacity of e_{up}, e_{vq} by θ. In other words, it moves the capacity from e_{pq} and e_{uv} to e_{up} and e_{vq} by reconfiguring the optical links. This procedure ensures the total number of ports used on each router remains unchanged.

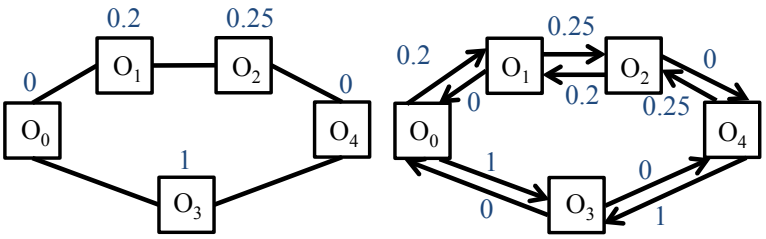
ComputeEnergy (Algorithm 11): This function computes the total throughput that can be achieved on the given state s, where s is a network-layer topology. The computation is divided into two steps. The first step is to establish multiple optical circuits for each link (line 2-14) based on its desired capacity, and the second step is to assign routing paths and rates to the flows based on the topology (line 15-25).

In the first step, we have constraints 2-4 in the problem formulation to affect whether an optical circuit can be established for a link. We use a regenerator graph to help us compute an optical circuit under these constraints. The nodes in a regenerator graph contain the source site, the destination site, and the sites that have remaining regenerators. We create an edge in the regenerator graph if the shortest paths between two sites is no longer than η. Figure 4.5(a) shows a regenerator graph. If the source and the destination are

Algorithm 11 Compute Energy

```
1: function ComputeEnergy(s)
       // build optical circuits for each link
2:     for network link l ∈ s.links do
3:         Build regenerator graph RG
4:         Build transformed graph TG
5:         P ← TG.sortedPathsByLength(l.src, l.dst)
6:         c ← l.capacity
7:         for path p in P do
8:             if p.canBeBuilt() then
9:                 Build circuit p for l
10:                c ← c − θ
11:                if c <= 0 then
12:                    break
13:        if c > 0 then
14:            Decrease the cacacity of l by c
       // assign routing paths and rates
15:    throughput ← 0
16:    Sort transfers F by policy // e.g., SJF, EDF
17:    l ← 1
18:    while (there exists unsatisfied demand
19:    and there exists free network capacity) do
20:        for transfer f ∈ F do
21:            for path p ∈ paths of f with length l do
22:                min_c ← min_{e∈p} e.remain_capacity
23:                r_{f,p} ← min(f.demand, min_c)
24:                throughput+ = r_{f,p}
25:        l ← l + 1
26:    return throughput
```

directly connected in the graph, we can directly establish an optical circuit; otherwise, they have to use regenerators in the intermediate sites. We want to balance the consumption of regenerators in different sites to improve the possibility that a later optical circuits can find an available one to use. To do this, we assign a weight to each node with *the inverse of their remaining regenerators*; the source and the destination nodes are assigned with weight zero. Then the problem is to find a path with smallest total node weight in the regenerator graph. This problem can be transformed to the shortest path problem in a directed graph. The transformation first builds a transformed graph from the regenerator



(a) Regenerator graph. (b) Transformed graph.

Figure 4.5: Example of regenerator graph.

graph. The transformed graph has the same nodes as the regenerator graph. An undirected edge in the regenerator graph is replaced by two directed edges; the weight of an edge is set to be the weight of the node the edge points to. It is easy to prove that the shortest path (the path with the smallest total edge weight) in the transformed graph corresponds to the path with smallest total node weight in the regenerator graph. Figure 4.5(b) illustrates the transformed graph of Figure 4.5(a). After we have the transformed graph, we iterate the paths based on path length to find enough number of paths we need that can be built as optical circuits (line 7-12). Line 8-12 check whether there are available wavelengths on the path to use, and build the circuit if so. If there are not enough possible optical circuits to satisfy all the desired capacity, we have to decrease the link capacity (line 13-14).

For the second step, we assign paths and rates to each transfer based on the topology to optimize their completion times or deadlines met. The problem is known to be hard. Even if the topology is non-blocking and only the ingress and egress ports are bottlenecks, it is NP-hard to compute rate allocations to achieve the minimum average transfer completion time [8]. It is also NP-hard to maximize the number of transfers that can be finished before the deadlines, when the network is fixed and three or more distinct deadlines are present [16]. A good approximation algorithm is to route transfers based on the order of the remaining transfer size or the deadline. However, in our scenario, the network is not ideal and we need multi-path routing to route some transfers. We approximate the optimal result by using the same ordering of transfers and prioritizing transfers to use shorter paths first. We order transfers with classic scheduling policies like shortest job first (SJF) and

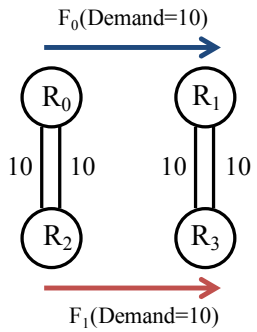


Figure 4.6: Example neighbor state.

earliest deadline first (EDF) (line 16). At each iteration, we only schedule transfers to use paths with length l (line 18-25). At each iteration, we assign rates to each transfer based on its demand and network capacity (line 22-23). Line 24 updates the total throughput. To avoid starvation, we schedule a transfer if it is not scheduled for $\hat{t}$ (configurable) time slots, which we omit in the algorithm for brevity.

Example: We use the example in Figure 4.3 to illustrate how the algorithm works. The initial state is the topology in Figure 4.3(a). The energy (i.e., the total throughput) of this state is 20 units. We use simulated annealing to find a new topology that can give a higher throughput (line 7-16 in Algorithm 9). We first find a neighbor state of the current state using the $ComputeNeighbor$ function (Algorithm 10). This function randomly select two links. Suppose it selects R_0-R_1 and R_2-R_3. We decrease 10 units of capacity from these two links, and adds 10 units of capacity to R_0-R_2 and R_1-R_3. Note that we change four links to find this neighbor state. If we only change one, two or three links, we would have a router that either does not fully use its two ports or uses more than two ports. Figure 4.6 shows the topology of the neighbor state. Since there are no paths to route F_0 and F_1, the energy of this state is 0 units, which is lower than the current state. The probability to jump to the neighbor state is $e^{(e_{current}-e_{neighbor})/T} = e^{20/T}$.

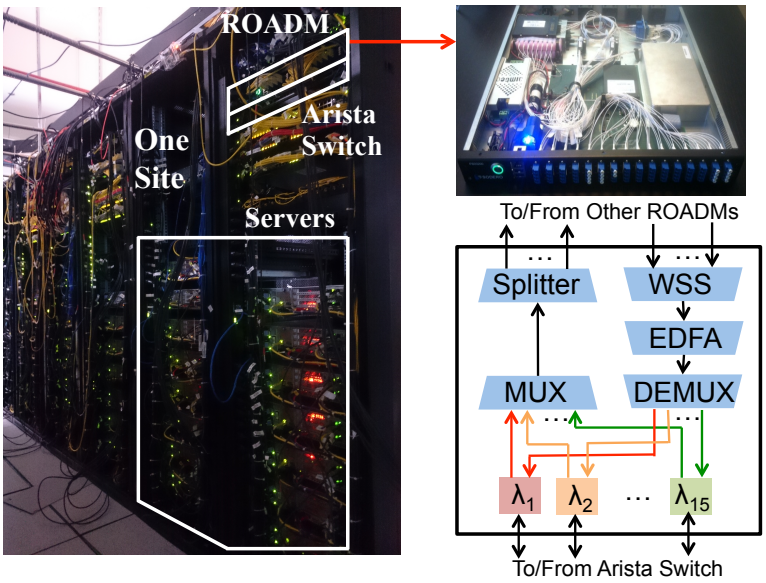
If we jump to the state in Figure 4.6, in the next round, we only have links R_0-R_2 and R_1-R_3 in the topology. The $ComputeNeighbor$ function can only select these two links and give us the original topology (the one in the Figure 4.3(a)) as the neighbor state.

Since the energy of Figure 4.3(a) is higher than that of Figure 4.6, we would jump back to Figure 4.3(a).

If we do not jump to the state in Figure 4.6, in the next round, we start again from the state in Figure 4.3(a). Now suppose the $ComputeNeighbor$ function selects R_0-R_2 and R_1-R_3. This gives us the neighbor state in Figure 4.3(d). We then compute the energy of the state. Since we add one link between R_0-R_1 and another one between R_2-R_3, we need to check whether we can establish optical circuits for these two links (line 2-14 in Algorithm 11). After we have all the links, we route F_0 and F_1 on the new topology (line 15-25). The routing first considers one-hop paths (line 17-25 in Algorithm 11). F_0 has two one-hop paths from R_0 to R_1, each of which has 10 units of free capacity. So the total rate of F_0 is 20 units. Similarly, F_1 also has two one-hop paths and has a total rate of 20 units. Then the routing considers two-hop paths (line 25 in Algorithm 11). Since F_0 and F_1 do not have other available paths, the $ComputeNeighbor$ function stops and returns 40 units as the energy of the state. Since this state has a higher energy than the current state, the algorithm jumps to this state and records this state as the best state encountered so far (line 10-12 in Algorithm 9). After a few other trials, the search stops and returns Figure 4.3(d) as the best topology it finds.

4.3.3 Updating Network State

After we compute the network state, we need to update the device configurations to the new state. Without careful update scheduling, there can be loops and routing blackholes during the update process. For example, if some packets were sent over a link with the underlying circuit being updated, these packets would be dropped. We need to be especially careful when updating the optical links as it can take several seconds. Dionysus is a recent solution on consistent network updates [65]. Dionysus builds a dependency graph to capture the dependencies between individual update operations and carefully schedules them to make the update fast and consistent. But Dionysus only handles network-layer



(a) Testbed. (b) ROADM.

Figure 4.7: Owan testbed implementation.

updates and is not sufficient to handle cross-layer updates. To solve this problem, we extend Dionysus by introducing *circuit nodes* into its dependency graph. Circuit nodes have dependencies on fibers as creating a circuit consumes a wavelength and removing a circuit frees a wavelength; circuit nodes also have dependencies on routing paths as a routing path cannot be used until circuits for all links on the path are established. After we build the dependency graph, we use the same scheduling algorithm as Dionysus to schedule the update operations.

4.3.4 Handling Practical Issues

Network failures: Link and switch failures are detected and sent to the controller. The controller removes these links and switches from the physical network, and recomputes the network state with the updated physical network. As our algorithm minimizes the amount of updates needed, it is likely to converge to a new feasible schedule with only incremental updates to avoid the failed links.

Controller Failures: Since the scheduling algorithm is stateless, we only need to store the physical network and the set of all transfers with a reliable distributed storage. When the controller fails, we spawn a new instance, which starts to compute and reconfigure the network state at the next time slot. During the controller failover, the network still carries traffic for the current time slot and is not affected.

Group of transfers: Some applications may need to send traffic to multiple locations and the important metric is the last completion time of all transfers in the group. This is similar to the coflow concept in big data applications in data centers [29, 28]. We can either treat them as single transfers or use better heuristics (like Smallest-Effective-Bottleneck-First [29]) to optimize for groups. A full exploration is our future work.

4.4 Owan Implementation

We have built a prototype of Owan. We describe the testbed hardware implementation in §4.4.1 and the controller software implementation in §4.4.2.

4.4.1 Owan Hardware Implementation

Our testbed has nine routers and ROADMs, and emulates the Internet2 topology in Figure 4.1. We use Arista 7050S-64 as the routers. Since commercial ROADMs are expensive, we use commodity optical components to emulate ROADMs that have the features needed to evaluate the system.

Figure 4.7 shows the prototype and the optical hardware design to emulate a ROADM. The optical elements for each ROADM is arranged in a 1U box. We have a Freescale i.MX53 micro controller in the box to control the optical elements. At the bottom of a ROADM, it has n(=15) ports that interface with the router. Each interface is an optical transceiver that can convert between electrical packets and optical signals at different wave-

lengths. The fifteen transceivers are at wavelengths from 1553.33nm to 1542.14nm, which are defined at standard ITU 100GHZ channel spacing.

In order to emulate any possible network-layer topology, we structure the nine ROADMs as a full mesh, i.e., each ROADM has a fiber to connect to every other ROADM. In this way, a ROADM can allocate the n wavelengths among the nine fibers arbitrarily as long as the total number of wavelengths in the nine fibers sum up to n. This means that in the network-layer topology, each router can have any number of links to any other router as long as the total number of links adjacent to a router satisfy the router port constraint. Therefore, our testbed can faithfully emulate the Internet2 network since the testbed is able to construct any network-layer topology that the Internet2 network is able to construct.

Figure 4.7 depicts the internal structure of our ROADM. For the outward direction of a ROADM, the n wavelengths from n transceivers are multiplexed by a multiplexer (MUX) on to a single fiber. Then the splitter replicates them and sends them to eight other ROADMs. For the inward direction, a Wavelength Selective Switch (WSS) receives n wavelengths from each neighbor and selects up to n different wavelengths from the input wavelengths. Then an Erbium-Doped Fiber Amplifier (EDFA) is used to amplify the wavelengths selected by the WSS, in order to compensate signal loss. Finally, a demultiplexer (DEMUX) demultiplexes the selected wavelengths and send them to corresponding ports. The MUXes and DEMUXes are the same type of device (Oplink AAWG) with different configurations.

To transmit packets from one router to another, the optical signal passes through multiple optical elements, including MUX, splitter, fiber, WSS and DEMUX. These five elements introduce typical optical power loss of 5 dB, 10.5 dB, 0.5 dB, 7 dB, and 5 dB, respectively. The total optical power loss is $\sim$28 dB, which is higher than the optical power budget ($\sim$16 dB) of the transceivers. That is the reason to put an EDFA between WSS and DEMUX. The EDFA is set to operate at fixed gain mode, and has a default setting of gain parameter of 18 dB to compensate the optical power loss.

4.4.2 Owan Software Implementation

The Owan controller is implemented with 5000+ lines of Java code and uses several third-party software and libraries. It has four modules. The core module computes the network state, and the other three modules handle interactions with routers, ROADMs, and client servers.

Core module: The core module implements the algorithms in §4.3. We have implemented the blossom algorithm [43] for maximum matching in general graphs and used JGraphT library [61] for other graph algorithms.

Router module: We configure the Arista switches to work in OpenFlow 1.0 mode. We use the Floodlight controller [37] to handle the details of the OpenFlow protocol and interface with the switches. The router module uses the RESTful API exposed by the Floodlight controller to install routing rules.

ROADM module: The Freescale i.MX53 micro controller in each ROADM handles the low level configurations, monitors the optical elements, and exposes a simple API for remote configuration. The ROADM module uses this API to configure each ROADM.

Client module: The client module sends the rate allocation of each transfer to the end hosts. Since a transfer may use multiple paths, we break a transfer into multiple flows and use prefix splitting to implement multi-path routing. The client module configures Linux Traffic Control on each end host to enforce rates.

4.5 Evaluation

In this section, we present the evaluation results. Besides a testbed that emulates the Internet2 topology, we have also built a flow-based simulator to evaluate Owan in a large scale with topologies and traffic from an ISP network as well as an inter-DC network from an Internet service company.

4.5.1 Methodology

Topologies: The testbed topology has nine sites as described in §4.4. We use Figure 4.1(b) as the network-layer topology to evaluate TE methods with only network-layer control. The simulations use a topology from an ISP backbone that contains about 40 sites. These sites are connected into an irregular mesh. The inter-DC network has about 25 sites. There are several sites called "super cores" that are connected to many smaller sites, and the super cores are connected in a ring topology.

Workloads: We obtain traces from both the ISP network and the inter-DC network. The traces are traffic counters collected from routers. From the traces, we can get site-to-site traffic demand, but not transfer-level details like transfer sizes and deadlines. Similar to previous work [69, 140], we use synthetic models to generate transfer-level information as follows. First, we sum up all the incoming and outgoing traffic demand for each site. Then we generate transfers for two hours. The transfers for the ISP network follow an exponential distribution with a mean of 500GB/5TB for testbed/simulation experiments. For each transfer, we randomly select a pair of sites whose total traffic demand has not exceeded the sum obtained from the traces. We multiply the sum of traffic demand at each site by a traffic load factor λ to evaluate the system under different loads. For deadline-constrained traffic, we choose deadlines from a uniform distribution between $[T, \sigma T]$ where T is the time slot length and σ is deadline factor that is used to change the tightness of deadlines. The inter-DC traffic follows roughly a similar distribution (with different λ values), except for that it has some "hotspots" in the network that generate lots of transfers for a period of time, and these hotspots can move from site to site.

Traffic engineering approaches: We compare the following approaches. Only Owan has optical-layer control. Tempus [69] and Amoeba [140] only work with transfers with deadlines, so we only compare them on deadline-constrained traffic in §4.5.3.

- Owan: The approach described in this chapter. It jointly controls the optical layer and the network layer.
- MaxFlow: This approach uses linear programming to maximize the total throughput for each time slot.
- MaxMinFract: This approach uses linear programming to maximize the minimal fraction that a transfer can be served at each time slot.
- SWAN [55]: It uses linear programming to maximize the throughput while achieving approximate max-min fairness for each time slot.
- Tempus [69]: It deals with deadline-constrained traffic. It first maximizes the minimal fraction a transfer can be served across all time slots and then maximizes the total number of bytes that can be satisfied.
- Amoeba [140]: This is another approach that deals with deadline-constrained traffic. It uses graph algorithms to compute routing and rate allocation for multiple time slots and adjust previous allocation when new transfers arrive.

Performance metrics: For deadline-unconstrained traffic, the primary metric is *transfer completion time*. We use *factor of improvement* to show the improvements of Owan over other approaches, which is the transfer completion time of the alternative approach divided by that of Owan. We also show *makespan*, which is the total time to complete a series of transfers.

For deadline-constrained traffic, we use *the percentage of transfers that meet deadlines* and *the amount of bytes that finish before deadlines*.

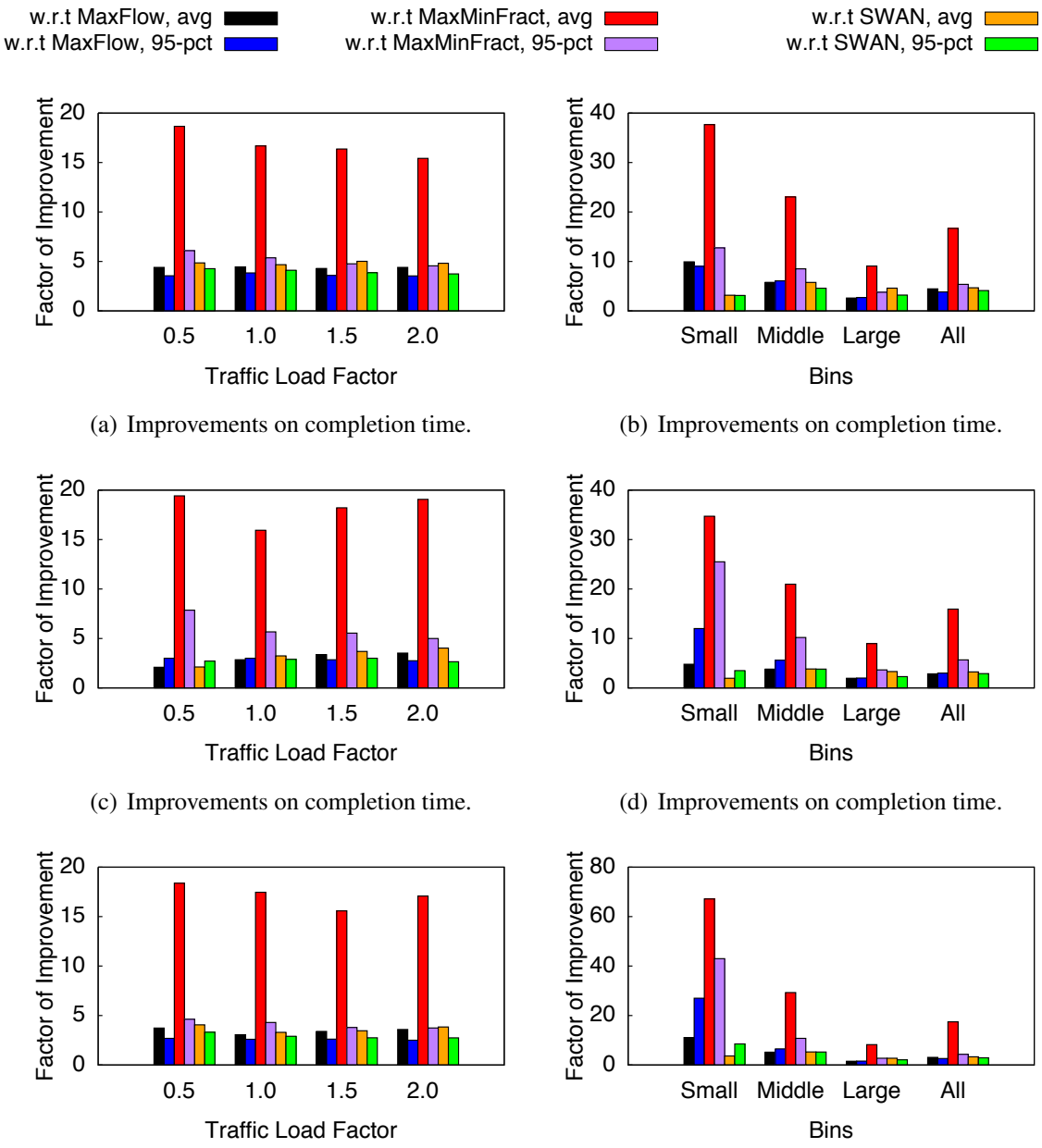
Performance validation: We have validated the results of our flow-based simulator with our testbed results on the Internet2 topology. The difference on the performance metrics is within 10%, which is mainly from the imperfect rate limiting and prefix splitting for multi-path routing on the testbed.

Testbed configurations: We run the controller on a commodity 2U server with two six-core Intel Xeon E5-2620v2 processors running at 2GHz. As we will show later, even this modest configuration is sufficient to run the Owan core module. All the test clients run on servers with the same configuration, and they connect to the network with 10GE. We use both *iperf* and a custom multi-threaded traffic generator to send emulated traffic, and we have verified that each client is able to saturate a 10Gbps link. Both generators have TCP enabled. We perform reconfigurations every five minutes.

4.5.2 Deadline-Unconstrained Traffic

In this experiment, the transfer requests submitted to the system do not have deadlines. The key metric is to optimize the transfer completion time. Figure 4.8(a-b) shows the results of the testbed experiments on the Internet2 topology. Figure 4.8(a) shows the factor of improvement on the average and the 95th percentile transfer completion time under different traffic loads. Compared to MaxFlow, Owan improves the average (95th percentile) transfer completion time by up to $4.45\times$ ($3.84\times$); compared to MaxMinFract, Owan improves the average (95th percentile) transfer completion time by up to $18.66\times$ ($6.09\times$); compared to SWAN, Owan improves the average (95th percentile) transfer completion time by $5.01\times$ ($4.27\times$). The results show that by dynamically reconfiguring the optical layer, Owan can significantly improve the transfer completion time for bulk transfers on the WAN. Moreover, we observe that Owan has bigger improvements over MaxMinFract than MaxFlow and SWAN. This is because MaxMinFract optimizes for the minimal fraction served by each transfer for each time slot, which causes most transfers to take several time slots to finish.

To further zoom in on the results, we divide the transfers into three bins (small, middle, large) based on transfer size, i.e., the smallest $1/3$ transfers are in the small bin, the middle $1/3$ in the middle bin, and the largest $1/3$ in the large bin. Figure 4.8(b) shows the factor of improvement in different bins when the traffic load factor is 1. Owan consistently improves



(a) Improvements on completion time.

(b) Improvements on completion time.

(c) Improvements on completion time.

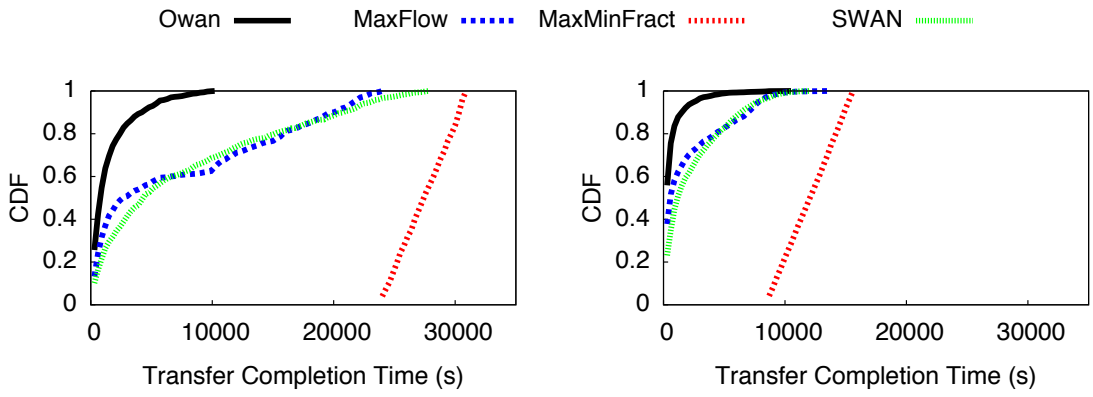
(d) Improvements on completion time.

(e) Improvements on completion time.

(f) Improvements on completion time.

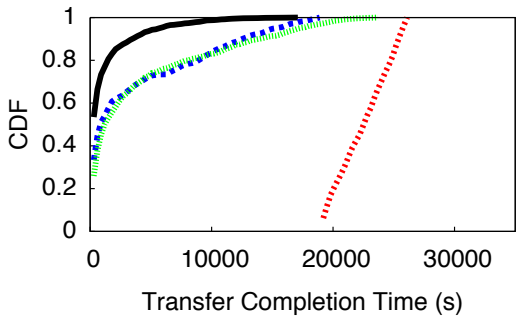
Figure 4.8: Results for deadline-unconstrained traffic. (a-b), (c-d), and (e-f) are results of the Internet2 network, ISP network, and inter-DC network, respectively.

the average and 95th percentile transfer completion time over MaxFlow, MaxMinFract and SWAN in different bins. We observe the most improvement is in the small bin. This is because Owan adjusts the network-layer topology based on traffic demand and small transfers are prioritized to take the most benefits of the topology.



(a) CDF of completion time.

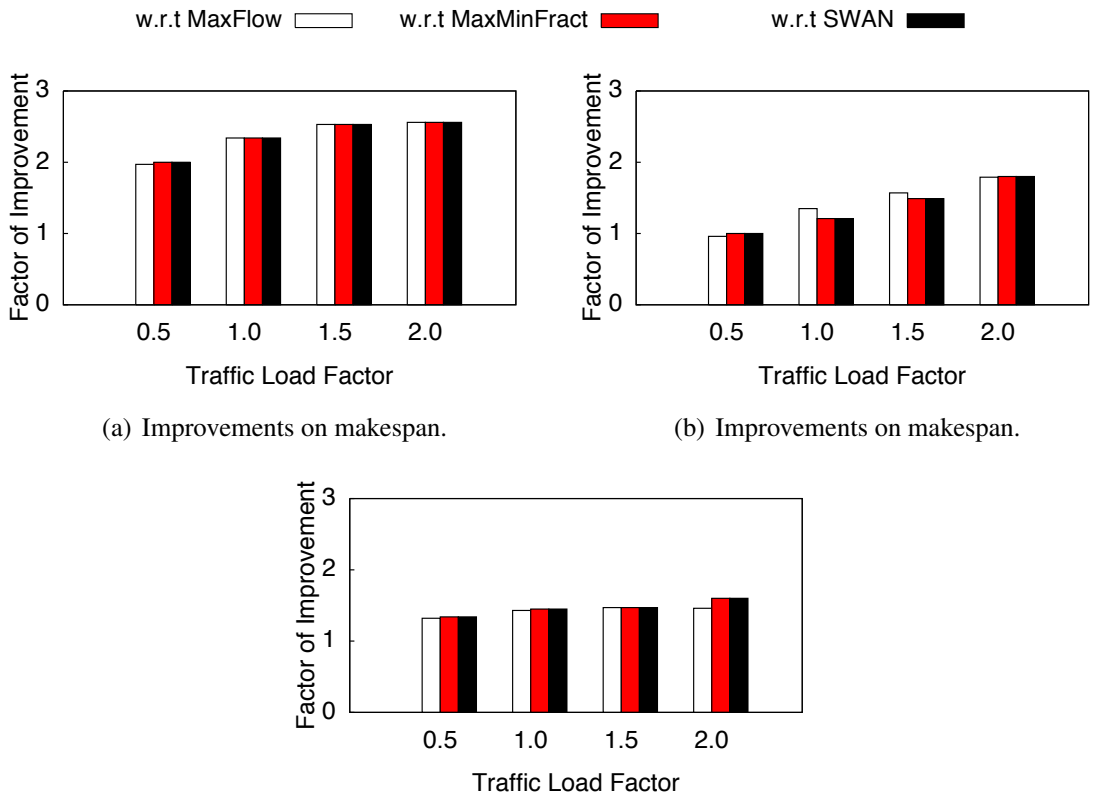
(b) CDF of completion time.



(c) CDF of completion time.

Figure 4.9: Results for deadline-unconstrained traffic. (a), (b), and (c) are results of the Internet2 network, ISP network, and inter-DC network, respectively.

To show the performance of Owan from another angle, we also plot the CDF of the transfer completion time. Figure 4.9(a) shows the CDF of the transfer completion time when the traffic load is 1. In the figure, the line of Owan stays at the leftmost, which means Owan achieves the smallest transfer completion time across all percentiles. MaxFlow, MaxMinFract and SWAN have longer tails than Owan. This means some transfers can have longer completion time than other transfers if MaxFlow, MaxMinFract or SWAN is used. The reason is also due to the fixed network-layer topology used by these approaches. The fixed topology causes many transfers to use multiple hops to reach their destinations and the total throughput of the network is lower than that in Owan. Overtime, some transfers are accumulated in the scheduling queue because of the limited total throughput and need to take a long time to complete.



(a) Improvements on makespan.

(b) Improvements on makespan.

(c) Improvements on makespan.

Figure 4.10: Results for deadline-unconstrained traffic. (a), (b), and (c) are results of the Internet2 network, ISP network, and inter-DC network, respectively.

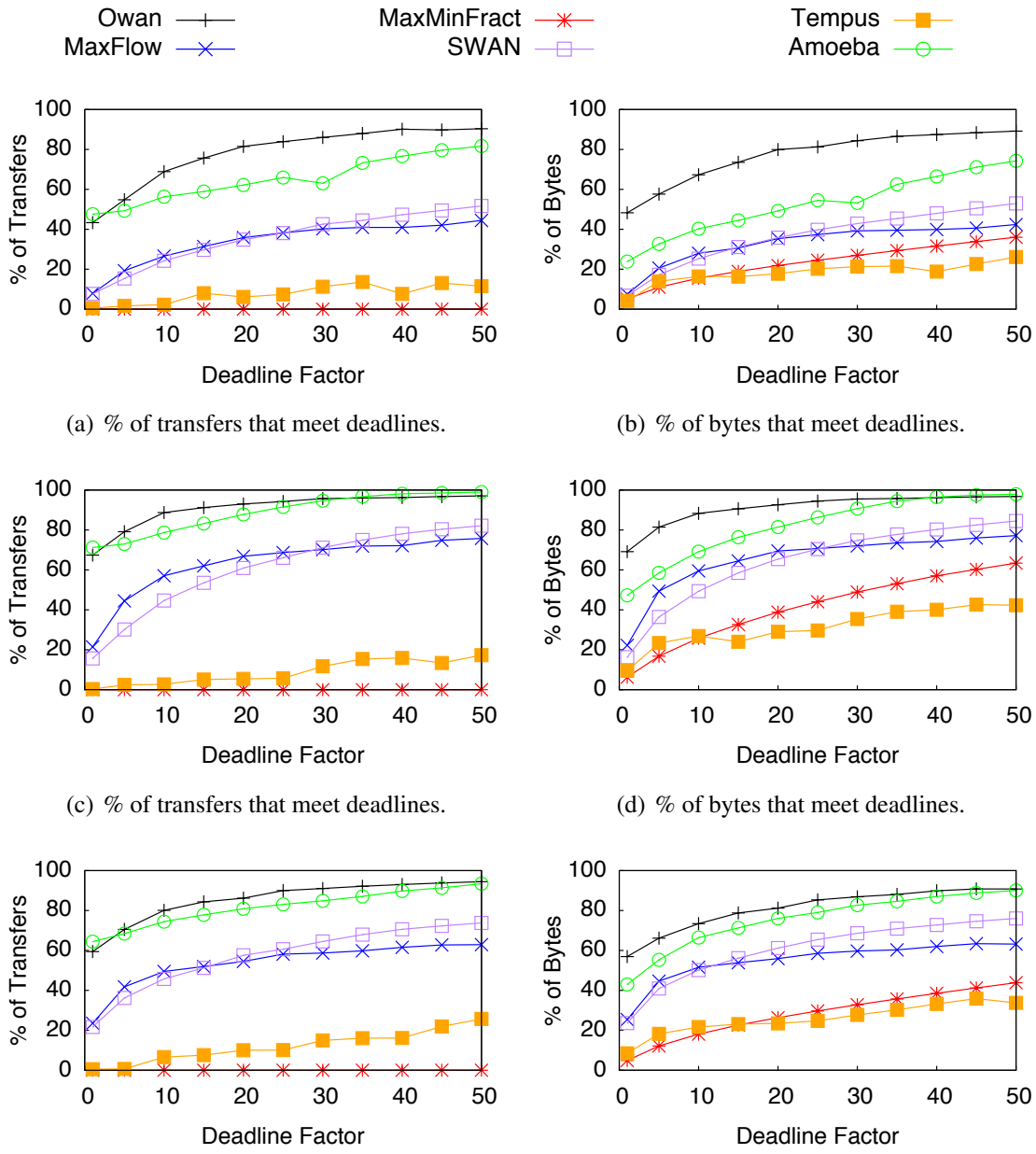
To evaluate Owan on a topology larger than our 9-site testbed, we also perform simulations using the ISP topology and inter-DC topology. Figure 4.8 and Figure 4.9 show the respective results. Similar to the Internet2 results, Owan significantly improves the transfer completion time. Specifically, Figure 4.8(c) shows that Owan improves the average (95th percentile) transfer completion by up to $3.52\times$ ($3.00\times$) as compared to MaxFlow, $19.42\times$ ($7.86\times$) as compared to MaxMinFract, and $4.03\times$ ($3.00\times$) as compared to SWAN. Also, Owan is better than the other three approaches across different bins (Figure 4.8(d)) and different percentiles (Figure 4.9(b)). Figure 4.8(e), Figure 4.8(f) and Figure 4.9(c) also show improvement factors on the inter-DC topology.

Finally, we show the improvement on *makespan*. Makespan is the total time to finish a given number of requests. We inject traffic requests for two hours and measure the

makespan of different approaches under different traffic loads. Figure 4.10 shows the improvement of Owan on makespan over the other three approaches. From the figure, we can see that Owan improves the makespan by up to $2.56\times$, $1.80\times$ and $1.60\times$ in the three topologies respectively. The improvement increases with the traffic load. This is because by reconfiguring the topology Owan can achieve higher total throughput and thus finish more requests in a certain time. When the load is higher, MaxFlow, MaxMinFract and SWAN have more transfers accumulated over time than Owan.

4.5.3 Deadline-Constrained Traffic

This experiment evaluates the performance of Owan for deadline-constrained traffic. In addition to MaxFlow, MaxMinFract and SWAN, we also compare Owan with another two approaches, Tempus and Amoeba, which are specifically designed for deadline-constrained traffic on the WAN. Figure 4.11(a-b) shows the results of testbed experiments on the Internet2 topology. Figure 4.11(a) shows the percentage of transfers that meet deadlines under different deadline factors. Owan enables the most number of transfers to meet deadlines. Amoeba is particularly designed for transfers to meet deadlines and performs the second best. The objective of Tempus is to maximize the minimal fraction served for each transfer across all time slots and then maximize the total bytes that finish before their deadlines. Thus it has relative poor performance to enable transfers to meet their deadlines. Overall, Owan increases the number of transfers that meet their deadlines by up to $1.36\times$, as compared to Amoeba, which performs the second best. The improvement is relatively small when the deadline factor is too small or too large. This is because when the deadline factor is too small, all the transfers have tight deadlines and there is little room for Owan to further increase the number of transfers that can meet their deadlines. When the deadline factor is too large, many transfers can easily meet their deadlines, and the absolute value of the percentage is already high. The benefit of reconfiguring the optical layer is most significant when the deadline factor is not at extreme values.



(a) % of transfers that meet deadlines.

(b) % of bytes that meet deadlines.

(c) % of transfers that meet deadlines.

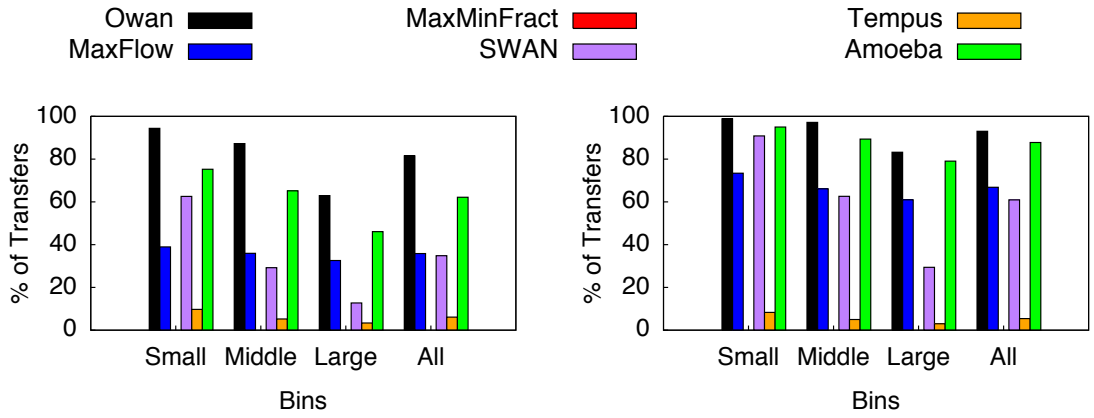
(d) % of bytes that meet deadlines.

(e) % of transfers that meet deadlines.

(f) % of bytes that meet deadlines.

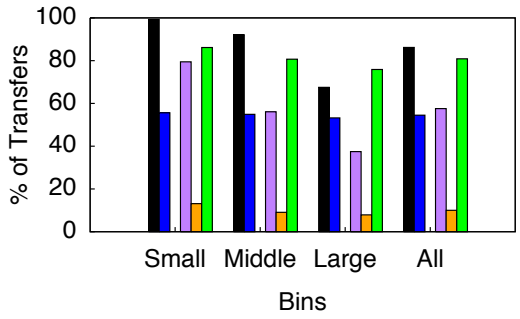
Figure 4.11: Results for deadline-constrained traffic. (a-b), (c-d), and (e-f) are results of the Internet2 network, ISP network, and inter-DC network, respectively.

Besides the percentage of transfers that meet their deadlines, we also show the percentage of bytes that finish before the deadlines in Figure 4.11(b). Owan outperforms other approaches more significantly on this metric. It improves the bytes that finish before the deadlines by up to $2.03\times$ than the second best one (Amoeba). Also we can see that



(a) % of transfers that meet deadlines.

(b) % of transfers that meet deadlines.



(c) % of transfers that meet deadlines.

Figure 4.12: Results for deadline-constrained traffic. (a), (b), and (c) are results of the Internet2 network, ISP network, and inter-DC network, respectively.

MaxMinFract and Tempus perform better on this metric than the percentage of transfers that meet their deadlines. This means they finish many bytes of a transfer though the entire transfer does not meet the deadline. This metric is important to applications that can use the available bytes as they arrive before the deadlines.

Similar to deadline-unconstrained traffic, we also show the breakdown of the percentage of transfers that meet deadlines in different bins with regard to transfer size. Figure 4.12(a) shows the result when the deadline factor is 20. Owan performs better than the other approaches across different bins.

Figure 4.11 and Figure 4.12 also show the results of the simulation results on the ISP and inter-DC topology. Similarly, Owan consistently outperforms other approaches. It improves the number of transfers that meet their deadlines by up to $1.13\times$ and $1.08\times$

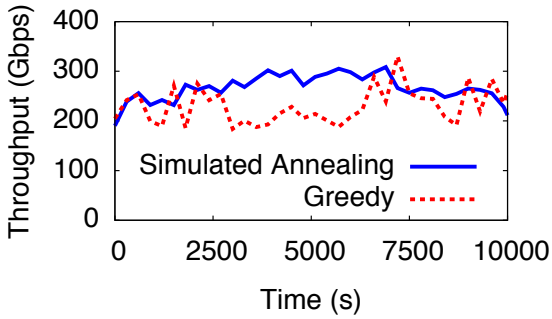
respectively, and the number of bytes that finish before their deadlines by up to $1.46\times$ and $1.33\times$ respectively, as compared to the second best alternative. Owan also performs well across different transfer sizes.

4.5.4 Microbenchmarks

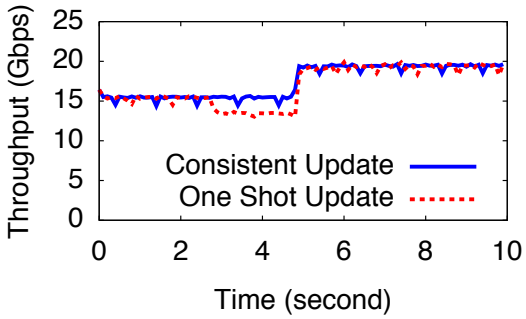
We now show some microbenchmarks. All the experiments are performed on the inter-DC topology with deadline-unconstrained traffic and the load factor being 1 if not otherwise specified.

Joint optimization of the optical and network layers: We show the benefit of jointly optimizing the optical and network layers. For comparison, we develop a greedy algorithm, which first builds a network-layer topology based on traffic demand between every two sites, and then it tries to find a routing configuration that maximizes total throughput using a similar routine as described in Algorithm 11. In other words, the greedy algorithm optimizes the optical layer and the network layer *separately*. The greediness simplifies the computation by limiting the search space. Unfortunately as Figure 4.13(a) shows, the total throughput is 21% less than the joint optimization, even if the joint optimization is only an approximation using simulated annealing. This performance difference is not incidental: as we have multiple paths for each flow, the routing configuration is tightly coupled with the optical configuration. Also, the greedy algorithm does not try to minimize the number of optical links to change while the simulated annealing algorithm does.

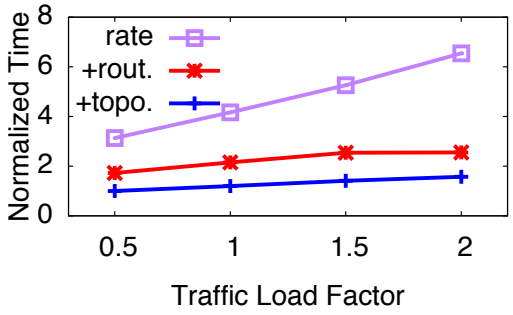
Consistent network updates: It takes about three to five seconds on our testbed to reconfigure an optical circuit. During the update of an optical circuit, the circuit goes dark and cannot carry any traffic. To avoid traffic disruptions, we use a consistent update scheme in §4.3.3. To demonstrate its effectiveness, Figure 4.13(b) shows the comparison of with and without the consistent update scheme. Without consistent update, all links are updated simultaneously in one shot to minimize update completion time. The total throughput drops 10% during the update, as packets get lost on these links, affecting the overall TCP perfor-



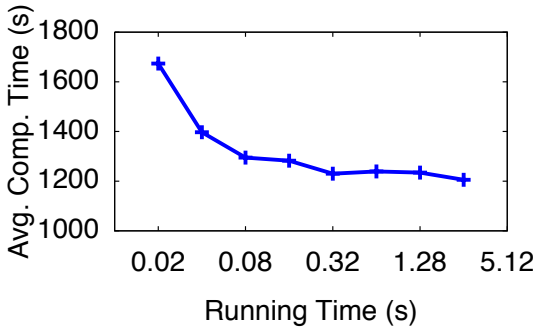
(a) Simulated annealing vs. the greedy algorithm.



(b) With and without consistent update scheme.



(c) Breakdown of gains.



(d) Impact of running time of simulated annealing.

Figure 4.13: Microbenchmark results.

mance. With consistent update, we observe no throughput drop during the update process, and we do not observe changes in end-to-end packet drop rate either.

Breakdown of gains: Owan jointly optimizes network-layer topology, routing and rate allocation. We use an experiment to show a breakdown of gains from controlling the three parts. Figure 4.13(c) shows the result of the experiment. In the experiment, we compare the average transfer completion time when the system has different levels of control of the network. All times are normalized by the average transfer completion time when the traffic load factor is 0.5 and the system has controls of all three parts. In the most basic scheme, the "rate" line in the figure, the system only controls rate allocation. The system cannot reconfigure the network-layer topology, nor can it change routing. It can only adjusts the sending rates of the transfers. In the second scheme, the "+rout." line in the figure, the system has controls of both routing and rate allocation. It assigns routing paths and rates to transfers similar to line 15-25 in Algorithm 11. The third scheme, the "+topo." line in the

figure, has controls of all three parts. As we can see from the figure, we have lower average transfer completion time when we have more control of the network.

Running time and convergence: We use simulated annealing to find a good topology. Since simulated annealing is an approximation algorithm that performs probabilistic search for the optimum, the quality of the result is related to its running time. The longer the algorithm runs, the more states it can search in the search space and the better the result can be. In our solution, since we use the current topology as the initial state of the algorithm, instead of a random topology, the algorithm starts its search with a reasonable good state. Since our system runs the algorithm and reconfigures the network every a few minutes, the traffic on the network is unlikely to change dramatically. Therefore, the algorithm can quickly find a good new topology by starting from the current topology and only changing a few links, as compared to starting from a random topology and spending a lot of time on finding a reasonably good topology. Figure 4.13(d) shows the performance of our algorithm when we run simulated annealing for different amounts of time. The performance is measured by the average transfer completion time. From the figure, we can see that the algorithm performs very bad when the simulated annealing only runs for 20 ms. However, the algorithm converges quickly, and it only requires about 320 ms to find a good topology to significantly reduce the average transfer completion time.

4.6 Related Work

WAN Traffic Engineering: Traffic engineering is a classic topic in networking research. Early work focuses on avoiding congestions. Many algorithms are developed to minimize the maximum link utilization under different conditions, such as changing traffic demands and network failures [10, 38, 68]. There are also efforts on achieving different fairness metrics theoretically and practically [32, 33]. With the emergence of SDN and the ability to program switches directly, researchers develop new centralized con-

trol systems, like Google B4 and Microsoft SWAN, to improve network utilization and its robustness in face of control plane and data plane failures [58, 55, 83, 76, 52]. Recent work goes beyond network-wide objectives like network utilization, to more fine-grained transfer-level objectives, like minimizing transfer completion time and meeting deadlines [69, 140, 24, 109, 78], and controls not only switches, but also proxies, load balancers, and DNS servers [84]. Owan follows the trend of centralized control for WANs. The key feature that differentiates Owan from previous solutions is the joint management of the optical and network layers, and we show that dynamically reconfiguring the optical layer can significantly.

The routing problem in overlay networks also concerns two layers [9, 81, 87]. The routing in the underlay network (the network layer in this chapter) builds the topology for the overlay network. However, the overlay and underlay networks are usually managed by different parties, and an overlay network usually traverses multiple ASes and has unstable end-to-end network performance.

Data-Center Traffic Engineering: Data-center networks have massive scale in terms of number of switches and hosts. Most traffic engineering work in data-center networks focuses on routing elephant flows as it is impractical to deal with all flows in a centralized manner [4, 15, 31, 118, 136, 107]. To cope with the scalability problem, CONGA designs a distributed load balancing mechanism and implement it in switch hardware [6]. Most of these solutions tackle the routing problem, i.e., choosing a path for a flow or flowlet. To solve the rate allocation problem, i.e., deciding the rate for each flow to optimize the flow completion time or the number of flows that meet deadlines, researchers have developed a wide range of new flow scheduling and congestion control algorithms [8, 7, 135, 126, 139, 54, 102, 12]. Some of them are entirely host-based; others leverage both host and switch features. There are also systems that optimize for a group of flows, which are important for many big data applications [29, 28]. Owan has similar objectives as these works, but the target of Owan is WANs. WANs do not have a structured topology as FatTree or CLOS

in data center networks (which many algorithms for data-center networks rely on), and the topology can be changed by reconfiguring the underlying optical layer.

Besides these works, some solutions propose to provide bandwidth guarantee to cloud applications and tenants, in order to provide predictable performance and enforce isolation [13, 60, 103, 79]. In these solutions, requests are formulated as bandwidth reservations between ingress and egress points. For bulk data transfers, it is more appropriate to formulate requests as volumes of data as in Owan. On the other hand, bandwidth reservations are also a useful abstraction for some use cases on the WAN. It is an interesting area of future work to explore how the reconfigurability in the optical layer can improve bandwidth reservations.

Optical Networks: With the advancements in optical technology and centralized control, researchers have started to build centralized production systems to manage the optical layer on the WAN [138, 14]. Xu *etal.* [138] present an on-line system to reconfigure the optical circuits given a set of circuit demands with constraints. Bathula *etal.* [14] develop algorithms to compute the minimal set of regenerator concentration sites such that any two optical ROADMs have at least one path available by using the selected sites. In terms of cross-layer control, early studies present algorithms and analysis for the joint optimization of the optical and network layers [47, 18, 110]. They mainly focus on admissible traffic demand and attempt to optimize for objectives like network cost and routing hops. Recent work begins to explore building systems to jointly control the optical and network layers, such as the DARPA CORONET program [27]. Our work is built up these efforts and presents the design and implementation of Owan to jointly control the optical and network layers and optimize bulk transfers for transfer completion time and deadlines met.

In terms of data centers, many researchers have proposed to use optics to boost the network performance. For example, Helios, cThrough and OSA use MEMS switches [35, 129, 25]; FireFly uses free-space optics [50]; WaveCube uses WSS switches [26]. The major objective in these works is to improve the *network throughput*. By reconfigur-

ing the topology, they can make the network be comparable to a non-blocking network, while saving on power, cost, and wiring complexity. Other works use optics to reduce latency [88]; use optics to support multicast [130, 115, 116, 137]; and design new optical hardware [104, 82, 86]. Differently, Owan reconfigures topologies in the WAN scenario, which uses ROADMs, regenerators and has the optical reach constraint, and Owan combines topology reconfiguration with routing and rate allocation to optimize transfer-level objectives.

4.7 Conclusion

We present Owan, a new traffic management system that optimizes bulk transfers on the WAN. Besides controlling routing and rate allocation, Owan goes one important step further than prior solutions into the optical layer. It reconfigures the optical layer in the same time scale as routing and rate allocation in a centralized manner. We develop efficient algorithms to compute the optical and routing configurations to optimize bulk transfers. Testbed experiments and large-scale simulations show that Owan completes data transfers up to $4.45\times$ faster in average and up to $1.36\times$ more flows meet their deadlines than methods with only network-layer control. Owan is the first step towards software-defined optical WANs. We believe centralized control of the optical and network layers would have a far-reaching impact on the theory and practice of network management for WANs.

Chapter 5

Conclusion

Network management is challenging as operators need to deploy multiple management applications for various management tasks, efficiently handle network events, and control network devices in different layers. This thesis presents a new network control platform to address these challenges. In this chapter, we summarize our contributions in §5.1, discuss open issues and future work in §5.2, and conclude in §5.3.

5.1 Summary of Contributions

Our control platform consists of three components: CoVisor, Dionysus and Owan. Our contributions include both efficient algorithms as well as realistic system implementation and evaluation.

Efficient algorithms: We design efficient algorithms to optimize the three components of our control platform. In CoVisor, we design efficient algorithms to compose policies from multiple applications into a single policy for the network, to compile policies from the virtual topology to the physical topology, and to handle policy updates from applications. In Dionysus, we design a dependency graph to capture dependencies between update operations, and develop adaptive scheduling algorithms to dynamically schedule updates according to runtime conditions. In Owan, we design an optimization algorithm that jointly

considers optical circuit setup, routing, and rate allocation to optimize bulk transfers. We also extend Dionysus for Owan to coordinate the updates in the optical layer and the network layer.

System implementation and evaluation: We build software prototypes and hardware testbeds to implement our platform and perform extensive evaluations. In CoVisor, we build a hypervisor prototype by extending and modifying OpenVirteX [5], and use a few example application compositions to show that our prototype is several orders of magnitude faster than a naive implementation. In Dionysus, we implement a scheduler prototype and build a hardware testbed that consists eight Arista switches. We evaluate Dionysus using both testbed experiments and large-scale simulations based on topology and traffic traces from a production data center network and a production WAN. Evaluation results show that Dionysus improves the median update speed by 53–88%. In Owan, we implement a controller prototype and build a hardware testbed with commodity optical and electrical hardware that emulates the Internet2 network. Besides testbed experiments, we also conduct large-scale simulations with data from an ISP WAN and an inter-DC WAN. Evaluation results show that Owan completes bulk transfers up to $4.45\times$ faster on average, and up to $1.36\times$ more transfers meet their deadlines, as compared to prior methods that only control the network layer.

5.2 Open Issues and Future Work

SDN is still at an early age. Our work is an attempt to improve the network control platform. It has the following open issues and directions for future work.

5.2.1 System Integration and Deployment

CoVisor and Dionysus can be naturally integrated into a single system by having CoVisor layered on top of Dionysus. CoVisor is used to host several applications and merge policy

updates from these applications into a single update for the network. Then Dionysus takes this single update as input and distribute it to multiple switches quickly and consistently. To integrate Owan to the system, we need to divide Owan into two parts. One part is the traffic management part, which computes the optical-layer configuration and the network-layer configuration. This part can be integrated to the system as an application that runs on top of CoVisor. The other part is the network update part, which coordinates the updates between the optical layer and the network layer. This part can be integrated to Dionysus to have the update scheduler handle both layers.

The overall system is a general network control system that can be deployed to networks under a single administrative domain, like a data center network, an ISP WAN, an inter-DC WAN, and an enterprise network. Some features of the system only apply to certain networks. Specifically, the dependency graph in Dionysus currently supports tunnel-based forwarding policies which are prevalent in WANs and WCMP-based forwarding policies which are prevalent in data center networks. So all the features of Dionysus are available for WANs and data center networks. The dependency graph is a general framework to capture dependencies in network updates. To use Dionysus in other networks, we need to extend the dependency graph to support policies in those networks. Furthermore, since Owan jointly optimizes the optical and network layers, its deployment requires operators to have control over both layers of the network, e.g., ISPs that control both layers of a public WAN and data center operators that have control of both layers of a private WAN.

5.2.2 Multi-Table Support in CoVisor and Dionysus

OpenFlow 1.0 [1] models the packet processing in switches as a single match-action tables. Later OpenFlow specifications [2] and recent initiatives on programmable data planes like P4 [17] generalize this model to multiple match-action tables. Extending existing and exploring new compilation techniques for multi-table support in CoVisor is a very promising direction [67, 117]. This would allow us to make efficient use of hardware capabilities and

reduce the size of the final policies for composition and devirtualization. Furthermore, it also introduces an incremental deployment path for new hardware as legacy applications written in OpenFlow 1.0 can run on top of CoVisor with CoVisor compiling them to multiple match-action tables that are supported in new hardware.

We also need to extend Dionysus to support multiple tables in a switch. In the multi-table setting, updates to multiple tables in the same switch are also dependent. We not only need to model dependencies between updates to different switches, but also updates to different tables in each single switch. Conceptually, we can view each table in the multi-table setting as a *switch* in the single-table setting, and we can extend the dependency graph to capture dependencies in the multi-table setting. A full exploration of this idea is an interesting direction of future work.

5.2.3 Bridging the Gap between Optics and Networking

Traditionally, the optical communications and the computer networking communities are separate, with one in EE and the other in CS. It requires different expertise to manage optical devices in the optical layer and electrical switches in the network layer. Operators have separate teams to manage these two layers. The emergence of SDN and the commoditization of optical devices and electrical switches greatly simplify the control of the two layers. It is an exciting time to build new management applications and control platforms to jointly manage these two layers and bridge the gap between the two communities. Our work on Owan is just the first step in this direction. There are many unsolved research problems in this direction, e.g., dynamically provisioning the optical layer to accept more bandwidth reservation requests from customers. Furthermore, as most optical devices today are not standard and use vendor-specific interfaces, there are also great opportunities in designing open platforms and interfaces for optical networks, i.e., the "OpenFlow" for optics.

5.3 Concluding Remarks

This thesis has (1) presented a new network hypervisor that can compose multiple network management applications and can efficiently merge policy updates from these applications; (2) designed a new network update scheduler that can quickly and consistently distribute policy updates to a distributed collection of switches; (3) developed a traffic management system that can jointly control the optical and network layers to optimize bulk data transfers over the wide area network.

At a high level, the goal of this thesis is to design and build network control systems to simplify network management, which is notoriously complicated today because of a diverse set of management tasks, prevalent network events, and complex configurations of devices in different layers. We leverage the emerging SDN technology to design a new centralized control platform to solve these challenges with efficient algorithms and solid system engineering. We believe this is a fruitful research area, and are excited about future research on designing new network control platforms with next-generation programmable switches and high-performance optical devices.

Bibliography

[1] OpenFlow Switch Specification 1.0.0. http://tinyurl.com/md8gge7.

[2] OpenFlow Switch Specification 1.5.0. http://tinyurl.com/qcz3bow.

[3] OpenFlow Table Type Patterns 1.0. http://tinyurl.com/oebjbsf.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI*, April 2010.

[5] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. OpenVirteX: Make your virtual SDNs programmable. In *ACM SIGCOMM HotSDN Workshop*, August 2014.

[6] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM*, August 2014.

[7] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, August 2011.

[8] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM*, August 2013.

[9] David G. Andersen, Alex C. Snoeren, and Hari Balakrishnan. Best-path vs. multi-path overlay routing. In *ACM SIGCOMM Conference on Internet Measurement Conference*, October 2003.

[10] David Applegate and Edith Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs. In *ACM SIGCOMM*, August 2003.

[11] Arista. Arista 7500 series technical specifications. http://tinyurl.com/lene8sw.

[12] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *USENIX NSDI*, May 2015.

[13] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM*, August 2011.

[14] Balagangadhar G Bathula, Rakesh K Sinha, Angela L Chiu, Mark D Feuer, Guangzhi Li, Sheryl L Woodward, Weiyi Zhang, Robert Doverspike, Peter Magill, and Keren Bergman. Constraint routing and regenerator site concentration in roadm networks. *IEEE/OSA Journal of Optical Communications and Networking*, 5(11):1202–1214, November 2013.

[15] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine grained traffic engineering for data centers. In *ACM CoNEXT*, December 2011.

[16] Maurizio A. Bonuccelli and M. Claudia Clò. Scheduling of real-time messages in optical broadcast-and-select networks. *IEEE/ACM Transactions on Networking*, 9(5):541–552, October 2001.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 44(3):87–95, July 2014.

[18] Andrew Brzezinski and Eytan Modiano. Dynamic reconfiguration and routing algorithms for IP-over-WDM networks with stochastic traffic. *Journal of Lightwave Technology*, 23(10):3188–3205, October 2005.

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

[20] Marco Canini, Daniele De Cicco, Petr Kuznetsov, Dan Levin, Stefan Schmid, and Stefano Vissicchio. STN: A robust and distributed SDN control plane. In *Open Networking Summit*, March 2014.

[21] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking*, 17(4), August 2009.

[22] Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *Workshop on Programmable Routers for Extensible Services of Tomorrow*, November 2010.

[23] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A retrospective on evolving SDN. In *ACM SIGCOMM HotSDN Workshop*, August 2012.

[24] Bin Bin Chen and Pascale Vicat-Blanc Primet. Scheduling deadline-constrained bulk data transfers to minimize network congestion. In *IEEE CCGRID*, May 2007.

[25] Kai Chen, Anubhav Singla, Ashutosh Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. OSA: An optical switching architecture for data center networks with unprecedented flexibility. In *USENIX NSDI*, April 2012.

[26] Kai Chen, Xitao Wen, Xingyu Ma, Yan Chen, Yong Xia, Chengchen Hu, Qunfeng Dong, and Yongqiang Liu. WaveCube: A scalable, fault-tolerant, high-performance optical data center architecture. In *IEEE INFOCOM*, April 2015.

[27] Angela L. Chiu, Gagan Choudhury, George Clapp, Robert Doverspike, Mark Feuer, Joel W Gannett, Janet Jackel, Gi Tae Kim, John G Klincewicz, Taek Jin Kwon, et al. Architectures and protocols for capacity efficient, highly dynamic and highly resilient core networks. *IEEE/OSA Journal of Optical Communications and Networking*, 4(1):1–14, January 2012.

[28] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM*, August 2015.

[29] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with Varys. In *ACM SIGCOMM*, August 2014.

[30] Concurrent-trees Library. `https://code.google.com/p/concurrent-trees/`.

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

[32] Emilie Danna, Avinatan Hassidim, Haim Kaplan, Alok Kumar, Yishay Mansour, Danny Raz, and Michal Segalov. Upward max min fairness. In *IEEE INFOCOM*, March 2012.

[33] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *IEEE INFOCOM*, March 2012.

[34] Roberto Doriguzzi Corin, Matteo Gerola, Roberto Riggio, Francesco De Pellegrini, and Elio Salvadori. VeRTIGO: Network virtualization and beyond. In *European Workshop on Software Defined Networks (EWSDN)*, October 2012.

[35] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A hybrid electrical/optical switch architecture for modular data centers. In *ACM SIGCOMM*, August 2010.

[36] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shiriam Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM*, August 2013.

[37] Floodlight OpenFlow Controller. `http://floodlight.openflowhub.org/`.

[38] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing OSPF weights. In *IEEE INFOCOM*, March 2000.

[39] Nate Foster, Michael J. Freedman, Arjun Guha, Rob Harrison, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Mark Reitblatt, Jennifer Rexford, Cole Schlesinger, Alec Story, and David Walker. Languages for software-defined networks. *IEEE Communications*, 51(2):128–134, February 2013.

[40] Pierre Francois and Olivier Bonaventure. Avoiding transient loops during the convergence of link-state routing protocols. *IEEE/ACM Transactions on Networking*, 15(6), December 2007.

[41] Pierre Francois, Olivier Bonaventure, Bruno Decraene, and P-A Coste. Avoiding disruptions during maintenance operations on BGP sessions. *IEEE Transactions on Network and Service Management*, 4(3), December 2007.

[42] Pierre Francois, Mike Shand, and Olivier Bonaventure. Disruption free topology reconfiguration in OSPF networks. In *IEEE INFOCOM*, May 2007.

[43] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys*, 18(1):23–38, March 1986.

[44] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *ACM SIGCOMM*, August 2015.

[45] Soudeh Ghorbani and Matthew Caesar. Walk the line: Consistent network updates with bandwidth guarantees. In *ACM SIGCOMM HotSDN Workshop*, August 2012.

[46] Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. *SIGCOMM CCR*, 35(5), October 2005.

[47] Kyle Chi Guan. *Cost-effective optical network architecture: A joint optimization of topology, switching, routing and wavelength assignment*. PhD thesis, Massachusetts Institute of Technology, February 2007.

[48] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean Patrick Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A software defined internet exchange. In *ACM SIGCOMM*, August 2014.

[49] Pankaj Gupta and Nick McKeown. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects*, August 1999.

[50] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R Das, Jon P Longtin, Himanshu Shah, and Ashish Tanwer. FireFly: A reconfigurable wireless data center fabric using free-space optics. In *ACM SIGCOMM*, August 2014.

[51] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *ACM CoNEXT*, December 2012.

[52] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *ACM SIGCOMM*, August 2015.

[53] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving energy in data center networks. In *USENIX NSDI*, April 2010.

[54] Chi-Yao Hong, Matthew Caesar, and P Godfrey. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM*, August 2012.

[55] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, August 2013.

[56] Infinera ROADM Specification. `http://tinyurl.com/jjog6no`.

[57] Internet2. `http://www.internet2.edu`.

[58] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, August 2013.

[59] Java Standard Library. `http://docs.oracle.com/javase/7/docs/api/`.

[60] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical network performance isolation at the edge. In *USENIX NSDI*, April 2013.

[61] JGraphT Graph Library. `http://jgrapht.org`.

[62] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. CoVisor: A compositional hypervisor for software-defined networks. In *USENIX NSDI*, May 2015.

[63] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. SoftCell: Scalable and flexible cellular core network architecture. In *ACM CoNEXT*, December 2013.

[64] Xin Jin, Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu, Guangzhi Li, Wei Xu, and Jennifer Rexford. Optimizing bulk transfers with software-defined optical WAN. In *ACM SIGCOMM*, August 2016.

[65] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, August 2014.

[66] John P John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus routing: The Internet as a distributed system. In *USENIX NSDI*, April 2008.

[67] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to recongurable switches. In *USENIX NSDI*, May 2015.

[68] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *ACM SIGCOMM*, August 2005.

[69] Srikanth Kandula, Ishai Menache, Roy Schwartz, and Spandana Raj Babbula. Calendaring for wide area networks. In *ACM SIGCOMM*, August 2014.

[70] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the one big switch abstraction in software-defined networks. In *ACM CoNEXT*, December 2013.

[71] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in software-defined networks. In *IEEE INFOCOM*, April 2013.

[72] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *ACM SIGCOMM HotSDN Workshop*, August 2013.

[73] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *USENIX NSDI*, April 2013.

[74] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[75] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, April 2014.

[76] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Björn Carlin, Mihai Amarandei-Stavila, Mathieu Robin, Aspi Siganporia, Stephen Stuart, and Amin Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *ACM SIGCOMM*, August 2015.

[77] Nate Kushman, Srikanth Kandula, Dina Katabi, and Bruce M Maggs. R-BGP: Staying connected in a connected world. In *USENIX NSDI*, April 2007.

[78] Nikolaos Laoutaris, Michael Sirivianos, Xiaoyuan Yang, and Pablo Rodriguez. Inter-datacenter bulk transfers with NetStitcher. In *ACM SIGCOMM*, August 2011.

[79] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM*, August 2014.

[80] Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to Algorithms*. The MIT press, 2001.

[81] Zhi Li and Prasant Mohapatra. QRON: QoS-aware routing in overlay networks. 22(1):29–40, 2004.

[82] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M Voelker, George Papen, Alex C Snoeren, and George Porter. Circuit switching under the radar with REACToR. In *USENIX NSDI*, April 2014.

[83] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *ACM SIGCOMM*, 2014.

[84] Hongqiang Harry Liu, Raajay Viswanathan, Matt Calder, Aditya Akella, Ratul Mahajan, Jitendra Padhye, and Ming Zhang. Efficiently delivering online services over integrated infrastructure. In *USENIX NSDI*, March 2016.

[85] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David A Maltz. zUpdate: Updating data center networks with zero loss. August 2013.

[86] Shiyun Liu, Qixiang Cheng, Adrian Wonfor, Richard V. Penty, Ian White, and Philip M. Watts. A low latency optical top of rack switch for data centre networks with minimized processor energy load. In *Optical Fiber Communication Conference*, March 2014.

[87] Yong Liu, Honggang Zhang, Wenyu Gong, and Don Towsley. On the interaction between overlay routing and underlay routing. In *IEEE INFOCOM*, March 2005.

[88] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. Quartz: A new design element for low-latency dcns. In *ACM SIGCOMM*, August 2014.

[89] Ratul Mahajan and Roger Wattenhofer. On consistent updates in software defined networks. In *ACM SIGCOMM HotNets Workshop*, November 2013.

[90] Rick McGeer. A safe, efficient update protocol for OpenFlow networks. In *ACM SIGCOMM HotSDN Workshop*, August 2012.

[91] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2), April 2008.

[92] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *USENIX NSDI*, April 2013.

[93] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: Dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM*, August 2014.

[94] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *USENIX NSDI*, April 2013.

[95] Mark Newman. *Networks: An Introduction*. Oxford University Press, 2009.

[96] Nicira. Network virtualization for cloud data centers. `http://tinyurl.com/c9jbkuu`.

[97] Andrew Noyes, Todd Warszawski, and Nate Foster. Toward synthesis of network updates. In *Workshop on Synthesis (SYNT)*, July 2013.

[98] Oclaro WSS. `http://tinyurl.com/hotq4s3`.

[99] Open Network Operating System (ONOS). `http://www.opendaylight.org/`.

[100] OpenDaylight Platform. `http://onosproject.org`.

[101] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *ACM SIGCOMM*, August 2013.

[102] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM*, August 2014.

[103] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical work-conserving bandwidth guarantees for cloud computing. In *ACM SIGCOMM*, August 2013.

[104] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *ACM SIGCOMM*, August 2013.

[105] POX OpenFlow Controller. `http://www.noxrepo.org/pox/about-pox/`.

[106] Zafar Qazi, Cheng-chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *ACM SIGCOMM*, August 2013.

[107] Sivasankar Radhakrishnan, Malveeka Tewari, Rishi Kapoor, George Porter, and Amin Vahdat. Dahu: Commodity switches for direct connect data center networks. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, October 2013.

[108] Barath Raghavan, Martín Casado, Teemu Koponen, Sylvia Ratnasamy, Ali Ghodsi, and Scott Shenker. Software-defined Internet architecture: Decoupling architecture from infrastructure. In *ACM SIGCOMM HotNets Workshop*, October 2012.

[109] Kannan Rajah, Sanjay Ranka, and Ye Xia. Advance reservations and scheduling for bulk transfers in research networks. *IEEE Transactions on Parallel and Distributed Systems*, 20(11):1682–1697, November 2009.

[110] Byrav Ramamurthy and Ashok Ramakrishnan. Virtual topology reconfiguration of wavelength-routed optical WDM networks. In *IEEE GLOBECOM*, November 2000.

[111] Saqib Raza, Yuanbo Zhu, and Chen-Nee Chuah. Graceful network state migrations. *IEEE/ACM Transactions on Networking*, 19(4), August 2011.

[112] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *ACM SIGCOMM*, August 2012.

[113] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: An open framework for OpenFlow switch evaluation. In *Passive and Active Measurement Conference*, March 2012.

[114] Ryu OpenFlow Controller. `http://osrg.github.io/ryu/`.

[115] P Samadi, D Calhoun, H Wang, and K Bergman. Accelerating cast traffic delivery in data centers leveraging physical layer optics and SDN. In *International Conference on Optical Network Design and Modeling*, May 2014.

[116] P Samadi, H Wang, D Calhoun, Y Xia, K Sripanidkulchai, TS Ng, and K Bergman. An optical programmable network architecture supporting iterative multicast for data-intensive applications. In *IEEE Optical Interconnects Conference*, May 2014.

[117] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent NetCore: From policies to pipelines. In *ACM ICFP*, September 2014.

[118] Ziyu Shao, Xin Jin, Wenjie Jiang, Minghua Chen, and Mung Chiang. Intra-data-center traffic engineering with ensemble routing. In *IEEE INFOCOM*, April 2013.

[119] Nick Shelly, Ethan J. Jackson, Teemu Koponen, Nick McKeown, and Jarno Rajahalme. Flow caching for high entropy packet fields. In *ACM SIGCOMM HotSDN Workshop*, August 2014.

[120] R. Sherwood, G. Gibb, K.K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *USENIX OSDI*, October 2010.

[121] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. In *ACM SIGCOMM*, August 2015.

[122] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, August 2003.

[123] Peng Sun, Minlan Yu, Michael J Freedman, Jennifer Rexford, and David Walker. HONE: Joint host-network traffic management in software-defined networks. *Journal of Network and Systems Management*, July 2014.

[124] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), June 1972.

[125] David E Taylor and Jonathan S Turner. ClassBench: A packet classification benchmark. In *IEEE INFOCOM*, March 2005.

[126] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter TCP (D2TCP). In *ACM SIGCOMM*, August 2012.

[127] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing packet classification for memory and throughput. In *ACM SIGCOMM*, August 2010.

[128] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. Lossless migrations of link-state IGPs. *IEEE/ACM Transactions on Networking*, 20(6), December 2012.

[129] Guohui Wang, David G Andersen, Michael Kaminsky, Konstantina Papagiannaki, TS Ng, Michael Kozuch, and Michael Ryan. c-Through: Part-time optics in data centers. In *ACM SIGCOMM*, August 2010.

[130] Howard Wang, Yiting Xia, Keren Bergman, TS Ng, Sambit Sahu, and Kunwadee Sripanidkulchai. Rethinking the physical layer of data center networks of the next decade: Using optics to enable efficient*-cast connectivity. *ACM SIGCOMM Computer Communication Review*, 43(3):52–58, July 2013.

[131] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Hot-ICE Workshop*, March 2011.

[132] Ye Wang, Hao Wang, Ajay Mahimkar, Richard Alimi, Yin Zhang, Lili Qiu, and Yang Richard Yang. R3: Resilient routing reconfiguration. In *ACM SIGCOMM*, August 2010.

[133] Xitao Wen, LE Li, C Diao, X Zhao, and Y Chen. Compiling minimum incremental update for modular SDN languages. In *ACM SIGCOMM HotSDN Workshop*, August 2014.

[134] Wikipedia. Petri net. `https://en.wikipedia.org/wiki/Petri_net`.

[135] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *ACM SIGCOMM*, August 2011.

[136] Xin Wu and Xiaowei Yang. Dard: Distributed adaptive routing for datacenter networks. In *IEEE ICDCS*, June 2012.

[137] Yiting Xia and TS Ng. A cross-layer sdn control plane for optical multicast-featured datacenters. In *ACM SIGCOMM HotSDN Workshop*, August 2014.

[138] Dahai Xu, Guangzhi Li, Byrav Ramamurthy, Angela Chiu, Dongmei Wang, and Robert Doverspike. On provisioning diverse circuits in heterogeneous multi-layer optical networks. *Computer Communications*, 36(6):689–697, 2013.

[139] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the flow completion time tail in datacenter networks. In *ACM SIGCOMM*, August 2012.

[140] Hong Zhang, Kai Chen, Wei Bai, Dongsu Han, Chen Tian, Hao Wang, Haibing Guan, and Ming Zhang. Guaranteeing deadlines for inter-datacenter transfers. In *EuroSys*, April 2015.