Flexible Enterprise Network Management on Commodity Switches

Nanxi Kang

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

January 2016

Abstract

Enterprise networks interconnect heterogeneous hosts, requiring careful management to provide secure, reliable and high-performance network communication. Today, the operators have to manually configure individual network devices, while considering the host address assignments and devices constraints (*e.g.,* limited memory). These approaches are too complicated and inefficient for enterprise networks with growing numbers of hosts and devices.

The rise of Software Defined Networks (SDN) offers opportunities to simplify the management of enterprise networks. Leveraging the logically-centralized control plane and the programmable switch rule-tables in SDN, we design a novel network management system that supports flexible policies and reduces configuration complexity. We argue that the operators should focus on defining network-wide policies rather than grappling with low-level details, such as switch memory sizes, individual switch configurations and host addresses. It is the controller's job to compile the high-level policies into rules for individual switches while staying within the rule-table sizes.

In this thesis, we present a flexible enterprise network management system that assigns addresses based on host attributes, distributes network-wide policies across multiple switches and computes switch rules to achieve high-level load balancing policies. Specifically,

- we propose the "Attribute-Carrying IPs" (ACIPs) abstraction, where the attributes of a host are encoded in the IP addresses to enable flexible policy specification. We present Alpaca, algorithms for realizing ACIPs under practical constraints of limited-length IP addresses and constraint switch rule-tables.

- we propose the "One Big Switch" abstraction, which consists of an endpoint policy that views all hosts connected to a single switch, a hop-by-hop routing policy that defines paths between endpoints and a compiler that synthesizes switch rules that

obey policies and adhere to the resource constraints. We present efficient algorithms that distribute policies across networks while managing rule-space constraints.

- we propose the "One Big Server" abstraction, where a switch load balances incoming service requests to multiple equivalent servers based on their capacities. We present Niagara, an SDN-based traffic-splitting scheme that achieves accurate traffic splits while being extremely efficient in using the rule-table space.

Acknowledgments

I'm deeply grateful to my advisor, Jennifer Rexford. It is she that introduces me to the world of computer networking. From her, I learnt how to conduct solid research work, how to present to the public as well as how to collaborate with people. Being an advisor, Jen always encourage me to take various opportunities such as conferences and internships, make my own decisions and get to know great people in the field. One of our projects, Niagara, started from my casual chatting with Monia Ghobadi, introduced by Jen. Working with her has been one of the most amazing experiences in my life. I would like to thank for all her advices, encouragement and support through my Ph.D. years.

I would have not completed this thesis without my collaborators. I would like to thank David Walker and Zhenming Liu for their tremendous help with my very first project (One Big Switch). Their clear and vigorous thinking always inspires me. Monia Ghobadi has been a great friend and a mentor for me. I owe so much to her for the continuous patience and confidence in me during the ups and downs of working on Niagara. I also want to thank John Reumann and Alexander Shraer for their contributions and feedbacks to the work. I'm very fortunate to work with Sanjay Rao and Ori Rottenstreich on Alpaca. Thanks to them for the heated discussion, effective execution and the fun final deadline rush.

I benefit a lot from my two internships with Microsoft. The first one was in my senior year in the college, when I interned in the system group at Microsoft Research Asia. I sincerely thank my mentors, Zheng Zhang and Zhenping Qian, for guiding me into the research world. During my second internship, I worked with Ming Zhang, Lihua Yuan, Guohan Lu and Ratul Mahajan. I'm greatly inspired by their intelligence and expert skills in applying cutting-edge research to industrial products. Their professions will long serve as an example for my career.

I would like to thank David Walker, Sanjay Rao, Michael Freedman and Nick Feamster to serve as my committee and provide feedbacks on my research.

I enjoyed research discussion and fun social activities with the people in Princeton. I would like to thank the (past and present) members of cabernet: Xin Jin, Peng Sun, Kelvin Zou, Praveen Naga Katta, Srinivas Narayana, Joshua Reich, Laurent Vanbever, Jennifer Gossels, Mojgan Ghasemi, Mina Tahmasbi Arashloo and Robert MacDavid. I would like to thank my other friends in Princeton: Jingwan Lu, Jennifer Guo, Fisher Yu, Xinyi Fan, Xiaozhou Li, Haoyu Zhang, Fanglu Liu, Linpeng Tang, Feng Liu, Yichen Chen, Mengdi Wang, Yanqi Zhou and Ping Lu. I would like to give my special thanks to Loretta Bercuk, who is always considerate and encouraging. The time we spent together is unforgettable.

This dissertation work is supported by National Science Foundation grant CNS-1162112, CNS-1409056 and CNS-1162112, University of Pennsylvania award 559611, and Office of Naval Research ONR N00014-12-1-0757.

I dedicate this thesis to my parents for their enduring love and faith in me.

To my parents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Enterprise Networks

An enterprise network interconnects many hosts, such as personal mobile phones, workstations and public servers, within a university or corporation. A typical enterprise network topology consists of two layers of devices, as shown in Figure 1.1. At the perimeter of the network, edge switches form disjoint islands by connecting physically co-located hosts (*e.g.,* computers in the same building). These islands are then connected by the core routers at the center of the network.

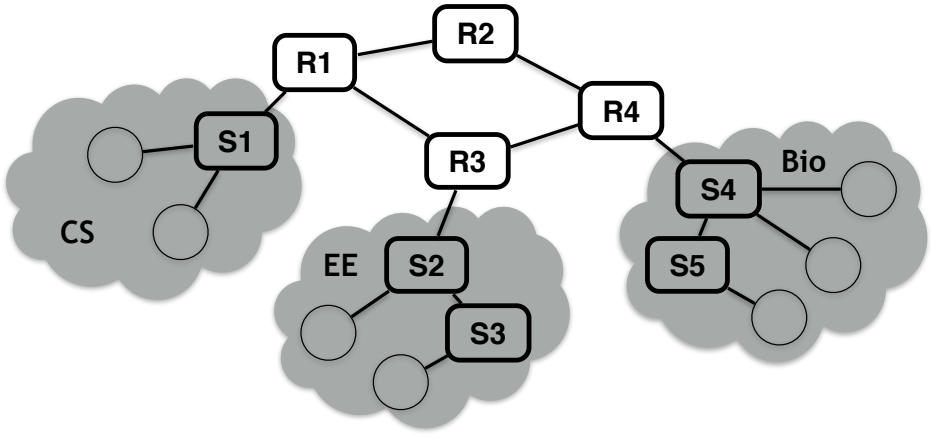


Figure 1.1: An example enterprise network.

To operate the network, the enterprise hires staffs (*i.e.,* operators), who manage devices (*i.e.,* switches and routers) to provide secure, reliable and high-performance network communication. The management tasks include assigning IP addresses to hosts, routing packets between hosts, setting up access control, providing Quality-of-Service to network traffic, balancing traffic loads of servers and so on. In what follows, we use the example network (Figure 1.1) to illustrate these tasks.

IP Address Assignment. A host has two addresses: the static MAC address, which is tied to the host permanently, and the dynamic IP address, which is assigned when the host joins a network. Conventionally, IP address assignment is location-dependent. The operators divide the enterprise's IP address space into subnets, each corresponding to a physical location (*e.g.,* a department building). Hosts in the same location are given IPs from the same subnet. For example, the operators could divide the address space 1.2.0.0/16 into 256 subnets, *i.e.,* 1.2.0.0/24, ..., 1.2.255.0/24, and assign 1.2.3.0/24 to the CS building and 1.2.4.0/24 to the EE building.

Routing. A network packet contains the addresses of the sender and the receiver. Switches and routers forward packets based on destination addresses. For example, within an island, switches forward a packet based on the receiver's MAC address by remembering the incoming port of packets sent by that address. Across islands, routers look up IP addresses, which are allocated based on locations. The operators configure routers so they could identify the subnet associated with each location and forward packets destined to the same subnet together. For example, router R1 can forward packets with destination IP address matching 1.2.3.0/24 (*e.g.,* 1.2.3.4), which is associated to the CS building, to S1.

Access Control. The operators use access control to block packets from untrustworthy hosts. For example, the database server should be accessible to only a small group of privileged hosts, or the communication between two unrelated departments should be forbidden. To enforce such policies, the operators install Access Control Lists (ACL) on routers, which examine packet header fields, such as source and destination IP addresses,

to permit or deny packets. For example, to allow network traffic from the CS building to the EE building, operators can install an ACL rule that permits packets with source IP addresses in 1.2.3.0/24 and destination IP addresses in 1.2.4.0/24.

Quality of Service. Some network traffic is more critical than others. For example, video streaming applications require packet delivery with very small latency, whereas packets of data backup applications are not latency-sensitive. Routers differentiate Quality-of-Service (QoS) for packets by isolating the critical traffic and the regular traffic in separated queues and prioritizing packet handling. The classification of traffic depends on certain header fields, such as the IP DSCP field, the TCP/UDP ports and so on. For example, the operators can set up a QoS rule that puts any packets with UDP port 3785, which denotes traffic of VoIP applications, into the critical queue.

Load Balancing. An enterprise network runs many public services (*e.g.,* web services, DNS services). Each service is replicated over multiple servers for greater throughput and reliability. The service requests from one client can be sent to any of these equivalent servers. To access a service, clients send requests destined to a public Virtual IP address (called VIP), which is associated with the service. A load balancer, placed in front of the servers and configured by the operators, rewrites the destination IP field in the VIP requests to the IP address (called DIP) of one of the servers. For example, server with DIP 1.1.1.1 hosts a web service, whose VIP is 1.0.1.1. The load balancer can redirect a request for VIP 1.0.1.1 to server 1.1.1.1 by rewriting its destination IP field. The goal of load balancing is to ensure that no server is overloaded.

1.2 Today's Enterprise Network Management

Management tasks are done in three layers, as shown in Figure 1.2. At the management plane, the operators decide policies for traffic handling, such as routing and access control, and specify these policies using the configuration APIs provided by the control planes.

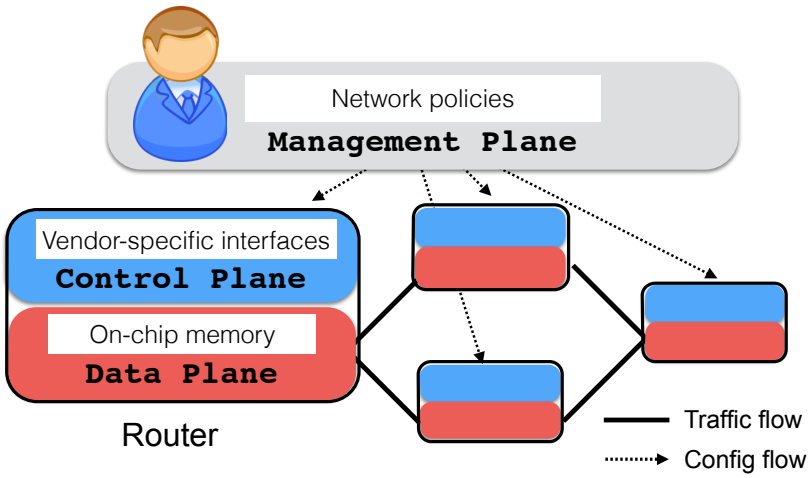


Figure 1.2: Today's enterprise network management.

The control planes then instruct the data planes to enforce these policies. The data planes can process packets at high speed using fast on-chip memory that stores packet-processing rules. For example, the operators may decide to allow traffic from the CS building to the EE building and interpretate this access control policy as "permit packets with source IP addresses in 1.2.3.0/24 and destination IP addresses in 1.2.4.0/24" using the control-plane APIs. The control plane installs the corresponding ACL rules in the on-chip memory, which is looked up by the data plane and executed on every matched packet.

Managing an enterprise network is not easy. Each device has its own data plane and control plane. Different vendors provide different control-plane configuration APIs, which support limited network policies. Today, the operators have to manually translate the network policies to control-plane configurations for individual devices, while considering the address assignment for hosts and differences in configuration APIs. In what follows, we discuss in detail the current enterprise network management approaches and their drawbacks.

1.2.1 VLAN-based IP Assignment

Virtual Local Area Network (VLAN) technology is widely used to assign hosts IP addresses in today's enterprise networks. In contrast to assigning co-located hosts addresses

from the same subnet, VLAN technology groups related hosts (*e.g.,* hosts affiliated with the same department) in the same subnet even if they are physically apart. A host is assigned a VLAN number when it joins the network, based on its MAC address or the physical switch port it connects to; related hosts are assigned the same VLAN number. The operators divide up the address space into subnets and associate each subnet with a VLAN number. Hosts are given addresses in the subnet that corresponds to its VLAN number.

A single VLAN can span multiple islands of hosts; a switch may participate in multiple VLANs. For each VLAN, the participant switches store the output port for a MAC address by remembering the incoming port of packets sent by that address. The switches also cooperate to construct a spanning tree. When one switch receives a packet destined to an unknown MAC address, it floods the packet along the spanning tree. Across VLANs, routers forward packets based on IP prefixes, *i.e.,* VLAN subnets. A packet is sent to a switch participating in the VLAN of the destination. Inter-VLAN communication must traverse a router. In other words, as long as they belong to different VLANs, even if two hosts connect to the same switch, their packets must be routed through the core routers, incurring unnecessary routing overhead.

VLANs are primarily used to simplify applying access control and QoS to groups of hosts [85]. Consider an access control policy that permits traffic from the CS department to the EE department. With VLAN, the operators can group CS hosts in one subnet, which may not be co-located at the same building, and EE hosts in another. As a result, the operators only need to configure a single ACL rule that permits packets from the CS VLAN subnet to the EE VLAN subnet. Similarly, QoS rules can classify packets using prefix matches on IP addresses. However, these ACL and QoS rules may only be installed on routers and applied to inter-VLAN traffic. Another problem is that VLANs always classify hosts into *disjoint* groups. If policies depends on orthogonal dimensions of information such as the affiliated department and the role of the device owner (*e.g.,* faculty or students),

the operators can only use VLANs for either one dimension or the product of both (*e.g.,* CS faculty), increasing configuration complexity.

1.2.2 Per-device Configuration

While the operators define network-wide policies at the management plane, they have to decide the portion of the policies implemented by each device and configure them individually, as a result of the distributed control planes and distinct control-plane APIs.

Per-device configuration is complicated and error-prone due to the lack of network-wide view and complex dependencies among policies. Consider enforcing the access control policy. Access control depends on routing: to deny unwanted traffic, operators need to install ACL rules on routers traversed by those packets. As packets may traverse different paths, ACL rules can span multiple routers. The configuration complexity is very high: operators have to decide for each router what packets should be dropped and what should be allowed. In fact, how the packets are treated depends on the aggregated actions taken by all devices along the path: while one device may permit the traffic, another device may deny the same traffic. Consider packets routed from S1 to S4 using path S1→R1→R3→R3→S4. The packets are safe, if all the devices permit the traffic; otherwise, they will be dropped in the middle.

Furthermore, manual per-device configuration reacts slowly to network changes (*e.g.,* link down, router failures). When changes occur, the operators have to inquire the current status of devices, decide the new configurations and apply the configurations accordingly. For example, when a router fails, the operators need to decide how to "migrate" the ACL rules on the failed devices to other devices such that the network-wide access control remains unchanged. All these tasks are performed manually at the granularity of individual devices.

1.2.3 Limited Policy Support

The existing control planes use different point solutions for each type of network policy. Not only are these point solutions hard to configure, they also support a limited set of policy options. Operators have to get through the inflexible control-plane APIs to enforce the desired policies, and sometimes it is even impossible to implement such policies.

Most control planes run routing protocols (*e.g.,* spanning trees or link state) to compute paths to the destination IP prefixes or destination MAC addresses. These algorithms only support forwarding packets on the shortest paths or paths in a spanning tree, thus losing the opportunities of using other paths. However, forwarding along a spare longer path and even multiple paths for a pair of sender and receiver increases throughput and reliability. To maximize bandwidth utilization, operators have to carefully tune the link weights to route traffic on customized paths.

Meanwhile, existing control planes provide limited support for access control and QoS. Many devices restrict ACL rules to prefix matches rather than wildcard matches. The operators have to either compute IP prefixes which describe the group of hosts sharing the same access privilege, or modify IP address assignment to allocate these hosts addresses in the same prefix (*e.g.,* VLANs). Similarly, many devices only support QoS rules matching on a small set of header fields, such as the DSCP field. To apply QoS policies, the operators have to ensure that hosts, or edge switches when hosts cannot be trusted, set the DSCP field correctly, otherwise the installed QoS rules cannot be applied.

Finally, load balancing is poorly supported by today's switches and routers. Most devices hash packet header fields to decide where to forward the packet, when there are multiple equivalent next-hops (*e.g.,* servers). However, the hashing solution does not consider the real traffic load (of each hash bucket) and could easily overload next-hops. Today, the operators have to purchase expensive dedicated load balancer appliances to distribute service requests, rather than trust the hashing schemes on switches.

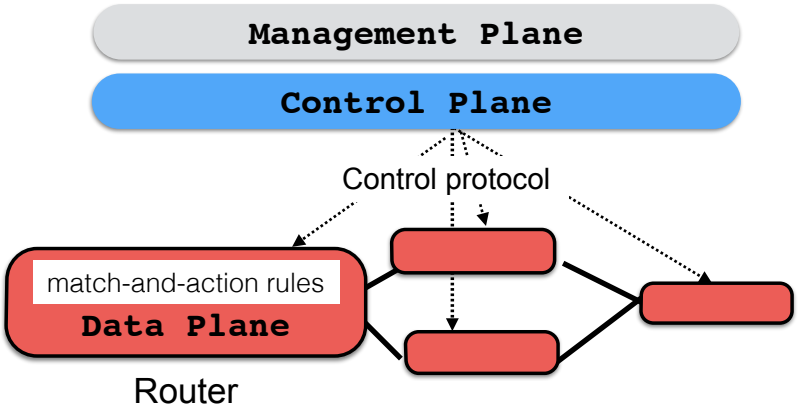


Figure 1.3: Software Defined Networks.

To summarize, the existing point solutions are inflexible, forcing operators to hack the network to (*e.g.,* tuning link weights or embedding information in a certain header field), and even purchase additional dedicated appliances to achieve the desired policies.

1.3 Flexible Network Management with SDN

1.3.1 Software Defined Networks

Software Defined Networking (SDN) proposes a new network architecture (Figure 1.3). The most important feature of SDN is the decoupled data plane and control plane, which were tightly integrated on individual devices before. SDN switches are purely data-plane devices, managed by a logically-centralized control plane (*i.e., controller*). The controller decides the behaviors of data-plane switches by installing packet-handling rules using a standard protocol (*e.g.,* OpenFlow).

The advanced architecture of SDN offers new opportunities to redesign network management. The logically-centralized control plane helps simplify policy specification. Rather than decide the configuration of each data-plane switch, the operators only need to define network-wide policies for the controller. Meanwhile, the operators do not need to worry about address assignment to enforce certain policies that depend on host information (*e.g.,*

access control policies depend on affiliated department). Given the policies, the controller can automatically compile the policies into rules for individual switches based on the current address assignment.

Another appealing benefits of SDN is the generic packet-handling rules. Abstractly, each rule contains a match and an action. The match could be exact, prefix and wildcard patterns on multiple packet header fields, allowing the controller to perform actions on a selected set of packets. The actions are diverse, including "send the packet to a port" (for routing), "drop the packet" (for access control), "put the packet in a queue" (for QoS) and so on. The "match and action" rules enable flexible and diverse policies. For example, operators could use prefix matches on destination IP to route packets to a subnet on a customized path, or they could use prefix matches on destination IP and wildcard matches on source IP to split the traffic to a subnet over multiple paths.

1.3.2 Switch Constraints

Switches have limited hardware resources. In SDN, the controller must deal with the constraints on multiple switches simultaneously. Memory limit is the one of the primary switch constraints.

Switches use three types of on-chip memory to store the packet-handling rules: Static Random Access Memory (SRAM), Content Addressable Memory (CAM) and Ternary Content Addressable Memory (TCAM). SRAM is usually used for prefix matches on a single header field; CAM only supports exact matches on header fields; TCAM is capable of wildcard matches (including prefix and exact matches) on multiple header fields. TCAM is the most popular rule-table due to its support for generic header field matches. It is widely used for access control, QoS and routing.

While SRAM and CAM have slightly bigger capacities, the rule-table size of TCAM is very small. A typical TCAM chipset can store at most a few thousands of rules. There are two reasons for the small rule-table size. On one hand, TCAM is power hungry. To perform

a single lookup, it needs to charge all the memory units and read all rules in parallel to identify the matching rule. On the other hand, a single TCAM chipset can process a limited number of lookups per second. Hence, to offer greater throughput, switches have to use multiple identical TCAMs that store the same set of rules. which further restricts the actual usable rule-table size.

Dealing with the constrained rule-table sizes has been a long-standing research topic. There has been a large body of prior work on *per-switch* optimization of limited memory through compression or caching. Given an input rule-set, the compression algorithms search for an equivalent rule-set that contains fewer rules. For prefix matches, optimal solutions are proposed to minimize the number of rules matching on one or two header fields [9, 75]. However, minimizing wildcard rules is NP-hard [75]. Algorithms are proposed to compress rules matching on multiple fields on a single rule-table [53, 54] and a pipeline of tables [52].

Besides compression, researchers propose caching to "expand" the rule-capacity of a switch [46]. The hardware rule-table is regarded as a cache, which stores a small number of popular rules matched by most packets. The control plane stores the full set of rules. When an incoming packet does not match the hardware rules, it is directed to the control plane for further processing, which is much slower. The control plane can use various caching algorithms (*e.g.,* Least-Recently-Used) to adjust the hardware rules over time. Caching allows the switch to store more packet-processing rules than what its hardware supports, by trading off the load on the control plane.

Despite the past work on rule optimization for a single switch, handling constrained rule-table sizes for multiple switches, as required in SDN, remains challenging. For example, an access control policy may need a large number of rules, which can not fit within a single switch after compression and can overwhelm the control plane if caching is used. As a result, the ACL rules must be partitioned into smaller rule-sets, which are distributed to multiple switches without exceeding the rule-table sizes. The partition problem becomes

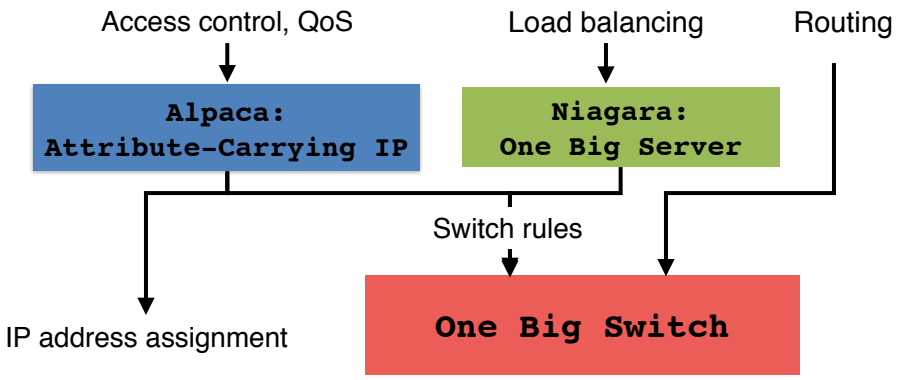


Figure 1.4: Abstractions for enterprise network management.

even harder, when routing is taken into account – each switch can only apply access control to packets that are routed through it. Per-switch rule optimization does not suffice for handling rule-table constraints in SDN.

1.3.3 Contributions

While SDN offers opportunities to simplify enterprise network management with a centralized control plane and unified "match-and-action" data-plane rules, handling the hardware constraints (*i.e.,* rule-table sizes) of multiple SDN switches simultaneously is challenging. In this thesis, we present a novel design for managing an SDN-enabled enterprise network, which supports flexible policies, reduces configuration complexity and handles hardware constraints.

Design Principles: Abstraction and Algorithms. We argue that the operators should focus on defining network-wide policies rather than grapple with low-level details, such as switch rule-table sizes, individual switch configurations and host addresses. It is the controller's job to compile the high-level policies into rules for individual switches while observing the rule-table sizes. We propose a series of *abstractions* for the operators to specify policies, which hide low-level details about switches and hosts, and *algorithms* for the controller to compute switch rules, which automate the realization of these abstractions with respect to the switch constraints.

Following the principles, we design a flexible enterprise network management system that (i) assigns addresses to reduce the number of switch rules for policies (Alpaca), (ii) distributes network-wide policies across multiple switches (One Big Switch) and (iii) computes switch rules to achieve high-level load balancing goals (Niagara). The system is shown in Figure 1.4. Given the access control and QoS policies defined for groups of hosts, Alpaca computes an efficient address assignment to minimize the switch rules that represent these policies. Meanwhile, Niagara computes a small rule-set to split traffic according to a high-level load balancing goal. The resulting switch rules of Alpaca and Niagara are distributed across multiple physical switches by "One Big Switch" based on the routing policy.

One Big Switch. Existing network architectures force operators to grapple simultaneously with end-to-end access control constraints, routing policy, switch memory limits, and the configurations of individual switches. We believe solutions to this complex problem should be factored into three distinct parts: (i) high-level endpoint policies (*e.g.,* access control, load balancing) on top of the "One Big Switch" abstraction, which hides internal topology details; (ii) a hop-by-hop routing policy, which specifies what paths traffic should follow between the ingress and egress ports; and (iii) a compiler that synthesizes switch rules that obey the user-defined policies and adhere to the resource constraints of the underlying hardware. We define and implement our proposed architecture, present efficient rule-placement algorithms that distribute policies across general SDN networks while managing rule-space constraints, and show how to support dynamic, incremental update of policies.

Attribute-Carrying IP Addresses. In enterprise networks, policies (*e.g.,* QoS or access control) are often defined based on the categorization of hosts along dimensions such as the organizational role of the host (faculty vs. student), and the department (engineering vs. sales). We propose Attribute-Carrying IPs (ACIPs), where the IP address allocation process in enterprises considers attributes of a host along all policy dimensions. ACIPs enable

flexible policy specification that may be infeasible owing to the limited size of switch rule-tables. Our solution, Alpaca (ALgorithms for Policies on Attribute-Carrying Addresses), realizes ACIPs under practical constraints of limited-length IP addresses and applies to different switch architectures. Alpaca computes (i) an efficient address assignment, based on the policy dimensions and the attributes of individual hosts, and (ii) a compact list of switch-level rules that realize a specific policy. Alpaca greatly simplifies enterprise management by (i) enabling administrators to specify policies on many orthogonal dimensions while achieving an order of magnitude reduction in the number of rules; (ii) allowing policies to be correctly maintained even when a host connects to the network at a new location; and (iii) simplifying federated management, where different teams manage different parts of the enterprise network.

One Big Server. A network hosts many public services (*e.g.,* websites), each replicated over multiple backend servers for greater throughput. Dedicated load-balancer appliances are expensive and do not scale with growing traffic demand. Rather than rely exclusively on special-purpose load-balancing equipment, we argue that commodity SDN switches, which offer high-speed packet processing as well as flexible interfaces for installing forwarding rules, should be programmed to perform most of the load-balancing function. We present the abstraction of "One Big Server" constituted by backend servers running the same services. We design the Niagara load balancer to distribute client requests within the "One Big Server". Niagara combines the per-packet performance of hardware switches and the large rule-space of software switches. The hardware switches approximate the load-balancing weights for each service, and the software switches correct for small errors in the approximation and ensure connection affinity during weight changes. Our main contributions are algorithms for (i) approximating the weights for each service, (ii) sharing a limited rule table across many services, and (iii) computing incremental updates to the rules when the weights change.

These three works significantly differ from the previous works that focus on optimizing the *given* rules for a single switch. All of our algorithms aim at *generating* rules for multiple switches to achieve high-level network policies with respect to the switch constraints. We (i) compute IP address assignments that would induce fewer rules, (ii) do network-wide minimization of rule tables through partitioning and placement, and (iii) generate rules that achieve a particular goal with maximum accuracy, subject to rule-table size as a constraint.

With these high-level management abstractions that enables operators to define diverse network-wide policies and efficient algorithms that realize these abstractions and deal with the switch constraints, our system achieves flexible management of enterprise networks built from commodity switches.

Chapter 2

Optimize "One Big Switch" Abstraction

Software Defined Networks (SDNs) offers direct, network-wide control over how switches handle traffic. Many SDN controller platforms [2, 24, 27, 31, 80] are developed, with which users can write control *applications* for diverse network policies. Unfortunately, these controller platforms force applications to grapple simultaneously with end-to-end connectivity constraints, routing policy, switch memory limits, and the hop-by-hop interactions between rules installed on different switches. We believe solutions to this complex problem should be factored in to three distinct parts: (1) an endpoint connectivity policy defined on top of "one big switch"; (2) a hop-by-hop routing policy; and (3) a compiler that synthesizes switch rules that obey the policies and adhere to the resource constraints of the underlying hardware.

In this chapter, we define and implement our proposed architecture, present efficient rule-placement algorithms that distribute forwarding policies across SDN networks while managing rule-space constraints, and show how to support dynamic, incremental update of policies. We evaluate the effectiveness of our algorithms analytically by providing complexity bounds on their running time and rule space, as well as empirically, using both synthetic benchmarks, and real-world firewall and routing policies.

2.1 Introduction

Software-Defined Networking (SDN) enables flexible network policies by allowing controller applications to install packet-handling rules on a distributed collection of switches. Over the past few years, many applications (*e.g.,* server load balancing, virtual-machine migration, and access control) have been built using the popular OpenFlow protocol [51]. However, many controller platforms [2, 24, 27, 31, 80] force applications to manage the network at the level of individual switches by representing a high-level policy in terms of the rules installed in each switch. This forces users to reason about many low-level details, all at the same time, including the choice of path, the rule-space limits on each switch, and the hop-by-hop interaction of rules for forwarding, dropping, modifying, and monitoring packets. Rule space, in particular, is a scarce commodity on current SDN hardware. Many applications require rules that match on multiple header fields, with "wildcards" for some bits. For example, access-control policies match on the "five tuple" of source and destination IP addresses and port numbers and the protocol [19], whereas a load balancing policy may match on source and destination IP prefixes [81]. These rules are naturally supported using TCAM, which can read all rules in parallel to identify the matching entries for each packet. However, TCAM is expensive and power hungry. The merchant-silicon chipsets in commodity switches typically support just a few thousand or tens of thousands of entries [73].

Rather than grappling with TCAM sizes, we argue that the SDN applications should define high-level policies and have the controller platform manage the placement of rules on switches. We, and more broadly the community at large [20, 56, 71, 74], have observed that such high-level policies may be specified in two parts, as shown in Figure 2.1:

- **An endpoint policy:** Endpoint policies, like access control and load balancing, view the network as *one big switch* that hides internal topology details. The policy spec-

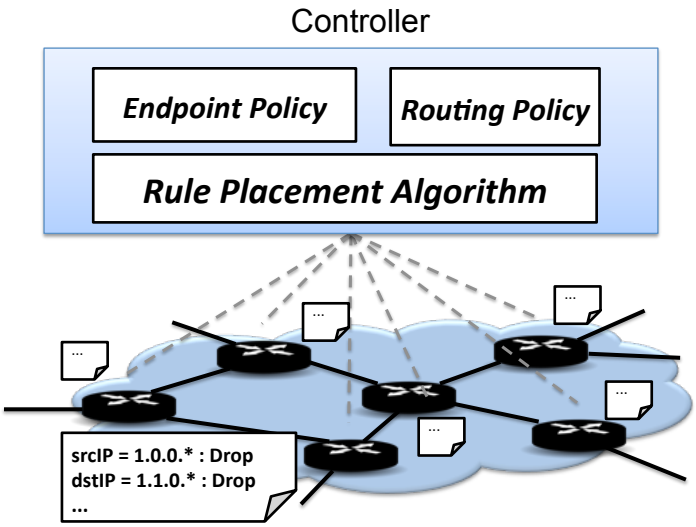


Figure 2.1: High-level policy and low-level rule placement

ifies which packets to drop, or to forward to specific egress ports, as well as any modifications of header fields.

- **A routing policy:** The routing policy specifies what paths traffic should follow between the ingress and egress ports. The routing policy is driven by traffic-engineering goals, such as minimizing congestion and end-to-end latency.

Expressing these two parts of the policy separately modularizes the problem and allows, for example, the security application to express an endpoint policy at the highest level of abstraction while the traffic engineering application plans the lower-level routing policy. Given two such specifications, the controller platform (a compiler) can apply a *rule-placement algorithm* to generate switch-level rules that realize both parts of the policy correctly, while adhering to switch table-size constraints.

Optimizing the placement of rules is challenging. Minimizing the number of rules for a *single* switch is computationally difficult [9], though effective heuristics exist [53]. Solving the rule-placement problem for an entire *network* of switches, given independent endpoint and routing policies, is even harder:

- Given the rule list for the traffic between two endpoints, we must generate an efficient and correct placement of rules along the path between them.
- The placement must carefully manage the interactions between packet modification and packet forwarding (which may depend upon the fields modified).
- A network consists of multiple paths that traverse the same switch and share the switch rule space. Consequently, we must consider the joint optimization problem across all paths.
- Since network policies evolve dynamically, we must be able to process changes efficiently, without recomputing rule placement from scratch.
- We must handle the fact that in each of the previous tasks, forwarding depends upon packet analysis over multiple dimensions of header fields.

In this paper, we take on the general challenge of solving the rule-placement problem. In doing so, we make a number of important contributions that range from new algorithm design, to complexity analysis, to implementation and empirical analysis on both synthetic and real-world data sets. More specifically, our central contributions include:

- The design of a novel rule-placement algorithm. The algorithm has as a key building block an elegant and provably efficient new technique for distributing rules along a linear series of switches.
- The design and analysis of *principled heuristics* for controlling the time complexity of our algorithm. These heuristics bring, among other things, the concept of *cost-effective covers* from the broader algorithms literature to bear on rule placement.
- The design of new *algorithms for incremental rule update* when either the endpoint or routing policy changes. Such algorithms are a crucial practical component of any SDN system that requires rapid response to a changing environment.

- An *evaluation* of our algorithms in terms of rule space and running time on both synthetic and real-world data that validates our algorithm.

In the next section, we formally introduce the optimization problem for the *one big switch* abstraction. Next, Section 2.3 presents related work on optimizing rule space. Section 2.4 gives an overview of our algorithm, followed by the detailed algorithm components (Section 2.5 and 2.6). Section 2.7 addresses the incremental update issues. Section 2.8 presents experiments involving both synthetic benchmarks and real-world policies. We conclude in Section 2.9.

2.2 Rule Placement in One Big Switch

A key problem in implementing the *one big switch* abstraction is mapping global, high-level policies to an equivalent, low-level set of rules for each switch in the network. We call this problem the *one big switch problem*, and introduce a precise formulation in this section.

Network topology: The network consists of n switches, each with a set of ports. We refer a port at a switch as a *location* (loc). The locations connected to the outside world are *exposed locations*. A packet enters the network from an exposed location called *ingress* and leaves at an exposed location called *egress*.

Packets: A packet (pkt) includes multiple header fields. Examples of header fields include source IP (src_ip) and destination IP (dst_ip). Switches decide how to handle traffic based on the header fields, and do not modify any other part of the packet; hence, we equate a packet with its header fields.

Switches: Each switch has a single, prioritized list of rules $[r_1, \ldots, r_k]$, where rule r_i has a predicate $r_i.p$ and an action $r_i.a$. A *predicate* is a boolean function that maps a packet header and a location (pkt, loc) into $\{\text{true}, \text{false}\}$. A predicate can be represented as a conjunction of clauses, each of which does *prefix* or *exact* matching on a single field

or location. An action could be either "drop" or "modify and forward". The "modify and forward" action specifies how the packet is modified (if at all) and where the packet is forwarded. Upon receiving a packet, the switch identifies the *highest-priority* rule with a matching predicate, and performs the associated action. A packet that matches no rules is dropped by default.

Endpoint policy (E): The endpoint policy operates over the set of exposed locations as if they were ports on one big abstract switch. An endpoint policy is a prioritized list of rules $E \triangleq [r_1, ..., r_m]$, where $m = \|E\|$ is the number of rules. We assume that at the time a given policy E is in effect, each packet can enter the network through at most one ingress point (*e.g.,* the port connected to the sending host, or an Internet gateway). An example of endpoint policy is shown in Figure 2.3(a).

Routing policy (R): A routing policy R is a function $R(\text{loc}_1, \text{loc}_2, \text{pkt}) = s_{i_1} s_{i_2} ... s_{i_k}$, where loc_1 denotes packet ingress, loc_2 denotes packet egress. The sequence $s_{i_1} s_{i_2} ... s_{i_k}$ is the path through the network. The routing policy may direct all traffic from loc_1 to loc_2 over the same path, or split the traffic over multiple paths based on packet-header fields. An example endpoint policy is shown in Figure 2.3(b).

Rule-placement problem: The inputs to the rule-placement problem are the network topology, the endpoint policy E, the routing policy R, and the maximum number of rules each physical switch can hold. The output is a list of rules on each switch such that the network (i) obeys the endpoint policy E, (ii) forwards the packets over the paths specified by R, and (iii) does not exceed the rule space on each switch.

2.3 Related Work

Prior work on rule-space compression falls into four main categories, as summarized in Table 2.1.

Types and examples	Switches	Optimize rule-space	Respect routing
Compressing the policy on a single switch [9, 53, 54]	Single	Yes	N/A
Distributing the policy at the network perimeter [19, 39, 86]	Edge	No	Yes
Distributing the policy while changing routing [58, 84]	All	Yes	No
Distributing the policy while respecting routing [45]	Most	Yes	Yes
Our work	All	Yes	Yes

Table 2.1: Prior work on rule-space compression.

Compressing the policy on a single switch: These algorithms reduce the number of rules needed to realize a policy on *a single switch*. While orthogonal to our work, we can leverage these techniques to (i) reduce the size of the endpoint policy which is the input to our rule-placement algorithm and (ii) further optimize the per-switch rule-lists output by our algorithm.

Distributing the policy at the network perimeter: These works distribute a centralized firewall policy by placing rules for packets at their ingress switches [19, 39], or verify that the edge switch configurations realize the firewall policy [86]. These algorithms do not consider rule-table constraints on the edge switches, or place rules on the internal switches; thus, we cannot directly adopt them to solve our problem.

Distributing the policy while changing routing: DIFANE [84] and vCRIB [58] leverage all switches in the network to enforce an endpoint policy. They both direct traffic through intermediate switches that enforce portions of the policy, deviating from the routing policy given by users. DIFANE takes a "rule split and caching" approach that increases the path length for the first packet of a flow, whereas vCRIB directs all packets of some flows over longer paths. Instead, we view the routing policy as something the SDN *application* should control, to achieve higher-level goals like traffic engineering. As such, our algorithms must grapple with optimizing rule placement while respecting the routing policy.

Distributing the policy while respecting the routing: Similar to our solution, Palette [45] takes both an endpoint policy and a routing policy as input, and outputs a rule placement. However, Palette leads to suboptimal solutions for two main reasons. First, Palette has *all* network paths fully implement the endpoint policy. Instead, we only enforce the portion of the endpoint policy affecting the packets on each path. Second, the performance of their algorithm depends on the length of the shortest path (with non-zero traffic) in the network. The algorithm cannot use all available switches when the shortest path's length is small, as is the case for many real networks. Section 2.8 experimentally compares Palette with our algorithm. We remark here that existing packet-classification algorithms [32, 72] could be viewed as a special case of Palette's partitioning algorithm. These techniques are related to a module in our algorithm, but they cannot directly solve the rule-placement problem. We make further comparisons with these techniques when we present our algorithm.

2.4 Algorithm Overview

This section gives an overview of our algorithm, which leverages the following two important observations:

The "path problem" is a building block: Given a packet pkt entering the network, the routing policy R tells the path $s_1, ..., s_\ell$ for the packet. In addition to forwarding pkt along this path, we must ensure that the endpoint policy E is correctly applied to pkt. In other words, we need to decide the rules for $s_1, ..., s_\ell$ so that these switches collectively apply E on all the traffic going through this path. Therefore, deciding the rule placement for path $s_1, ..., s_\ell$ to implement E is a basic building block in our algorithm.

The "path problem" is an easier special case of our problem: We may also interpret the path problem as a special case of our general problem, where the network topology

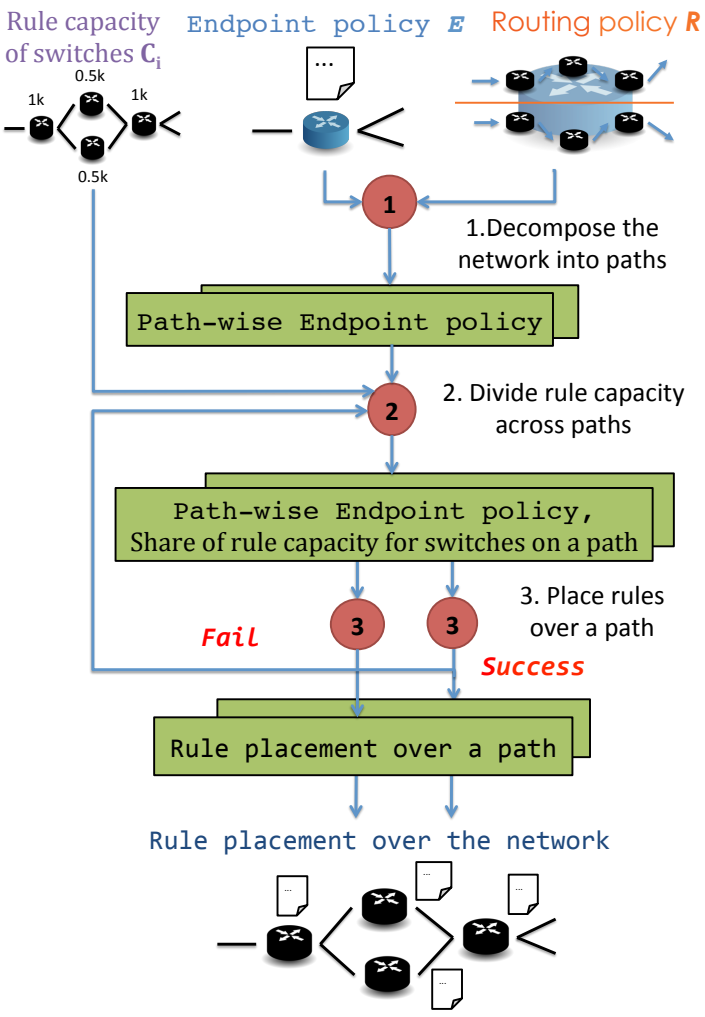


Figure 2.2: Overview of the rule placement algorithm

degenerates to a path. Thus, algorithmically understanding this special case is an important step towards tackling the general problem.

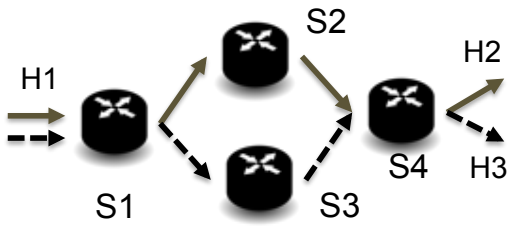
The high-level idea of our algorithm is to find an effective way to decompose the general problem into smaller problems over paths and design efficient heuristics to solve the path problems.

Figure 2.2 shows the three main components of our algorithm:

Decomposition (component 1): Our first step is to interpret the problem of implementing $\{E, R\}$ as implementing the routing policy R and a *union* of endpoint policies over

$$r_1 : (\text{dst\_ip} = 00*, \text{ingress} = H_1 : \text{Permit}, \text{egress} = H_2)$$
$$r_2 : (\text{dst\_ip} = 01*, \text{ingress} = H_1 : \text{Permit}, \text{egress} = H_3)$$

(a) An example endpoint policy E



(b) An example routing policy R

$$P_1 = s_1 s_2 s_4, D_1 = \{\text{dst\_ip} = 00*\}$$
$$P_2 = s_1 s_3 s_4, D_2 = \{\text{dst\_ip} = 01*\}$$

(c) Paths and flow spaces computed from E and R

Figure 2.3: An example decomposition

the *paths*. We give an example for the routing policy in Figure 2.3(a) and the endpoint policy in Figure 2.3(b). From these policies, we can infer that the packets can be partitioned into two groups (see Figure 2.3(c)): those in D_1 (using the path $P_1 = s_1 s_2 s_4$) and those in D_2 (using the path $P_2 = s_1 s_3 s_4$). We may separately implement two path-wise endpoint policies on P_1 and P_2. By doing so, we decompose the general rule-placement problem into smaller sub-problems. More formally, we can associate path P_i with a *flow space* D_i (*i.e.,* all packets that use P_i belong to D_i)[1] and the *projection* E_i of the endpoint policy on D_i:

$$E_i(\text{pkt}) = \begin{cases} E(\text{pkt}) & \text{if pkt} \in D_i \\ \perp & \text{otherwise,} \end{cases} \tag{2.1}$$

where $\perp$ means no operation is performed.

Resource allocation (component 2): The per-path problems are not independent, since one switch could be shared by multiple paths (*e.g.,* s_4 in Figure 2.3). Thus, we must divide the rule space in each switch across the paths, so that each path has enough space to

[1]We can associate a dropped packet with the most natural path it belongs to (*e.g.,* the path taken by other packets with the same destination address).

implement its part of the endpoint policy. Our algorithm *estimates* the resources needed for each path, based on analysis of the policy's structure. Then the algorithm translates the resource demands into linear programming (LP) instances and invokes a standard LP-solver. At the end of this step, each path-wise endpoint policy knows how many rules it can use at each switch along its path.

Path algorithm (component 3): Given the rule-space allocation for each path, the last component generates a rule placement for each path-wise policy. For each path, the algorithm efficiently searches for an efficient "cover" of a portion of the rules and "packs" them into the switch, before moving on to the next switch in the path. If the estimation of the rule-space requirements in step 2 was not accurate, the path algorithm may fail to find a feasible rule placement, requiring us to repeat the second and third steps in Figure 2.2.

In what follows, we present these three components from the bottom up. We start with the path algorithm (component 3), followed by the solution for general network topologies (components 1 and 2). We also discuss extensions to the algorithm to enforce endpoint policies as early in a path as possible, to minimize the overhead of carrying unwanted traffic.

2.5 Placing Rules Along a Path

Along a path, every switch allocates a fixed rule capacity to the path, as in Figure 2.4. For a single path P_i, the routing policy is simple—all packets in the flow space D_i are forwarded along the path. We can enforce this policy by installing fixed forwarding rules on each switch to pass packets to the next hop. The endpoint policy is more complex, specifying different actions for packets in the flow space. *Where* a packet is processed (or where the rules are placed), is constrained by the rule capacity of the switches. However, performing the defined action (such as drop or modification) for each packet *once* is enough

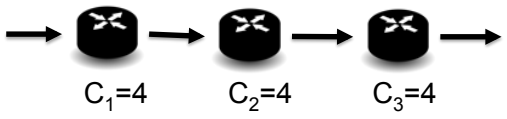


Figure 2.4: A 3-hop path with rule capacities (C)

along the path. Therefore, the endpoint policy gives us flexibility to moves rules among multiple switches.

Our goal is to minimize the number of rules needed to realize the endpoint policy[2], while respecting the rule-capacity constraints. We remark that there exists a standard reduction between decision problems and optimization problems [10]. So we will switch between these two formulations whenever needed. In what follows, we present a heuristic that recursively *covers* the rules and *packs* groups of rules into switches along the path. This algorithm is computationally efficient and offers good performance. For ease of exposition, we assume the flow space associated with the path is the full space (containing all packets) and the endpoint policy matches on the source and destination IP prefixes (two-dimensional). A fully general version of the algorithm is presented in the Section 2.5.3.

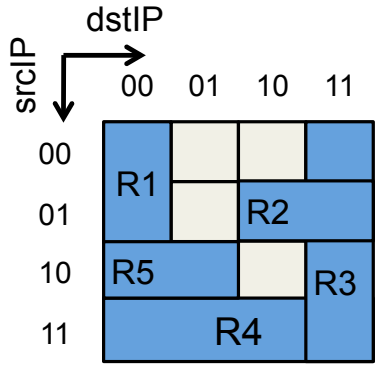
2.5.1 Cover-Pack-and-Replace

The endpoint policy E can be visualized using a two-dimensional space where each rule is mapped to a rectangle based on the predicate (Figure 2.5). Higher-priority rectangles lie on top of lower-priority rectangles, as packets are always handled by the matching rule with the highest priority. Rule $R6$ is intentionally omitted in the figure since it covers the whole rectangle. Let us consider enforcing the policy on the path shown in Figure 2.4. Since a single switch cannot store all six rules from Figure 2.5(a), we must divide the rules across

[2]Since optimizing the size of a rule list is NP-hard [75], we cannot assume an optimal representation of E is provided as input to our algorithm. Instead, we accept *any* prioritized list of rules. In practice, the application module generating the endpoint policy may optimize the representation of E.

$R1 : (\text{src\_ip} = 0*, \text{dst\_ip} = 00 : \text{Permit})$
$R2 : (\text{src\_ip} = 01, \text{dst\_ip} = 1* : \text{Permit})$
$R3 : (\text{src\_ip} = *, \text{dst\_ip} = 11 : \text{Drop})$
$R4 : (\text{src\_ip} = 11, \text{dst\_ip} = * : \text{Permit})$
$R5 : (\text{src\_ip} = 10, \text{dst\_ip} = 0* : \text{Permit})$
$R6 : (\text{src\_ip} = *, \text{dst\_ip} = * : \text{Drop})$

(a) Prioritized rule list of an access-control policy



(b) Rectangular representation of the policy

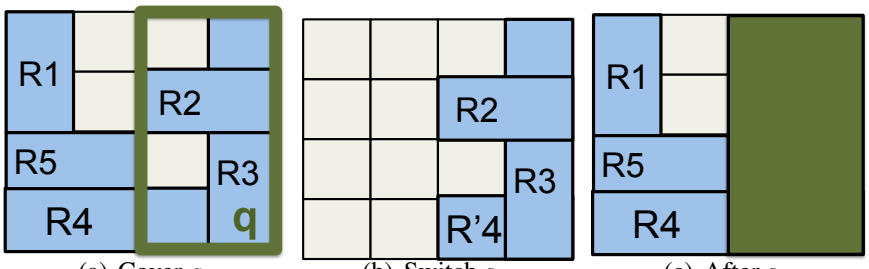
Figure 2.5: An example two-dimensional policy

multiple switches. Our algorithm recursively *covers* a rectangle, *packs* the overlapping rules into a switch, and *replaces* the rectangle region with a single rule, as shown in Figure 2.6.

Cover: The "cover" phase selects a rectangle q as shown in Figure 2.6(a). The rectangle q overlaps with rules $R2$, $R3$, $R4$, and $R6$ (overlapping rules), with $R2$ and $R3$ (internal rules) lying completely inside q. We require that the number of overlapping rules of a rectangle *does not exceed* the rule capacity of a single switch.

Pack: The intersection of rectangle q and the overlapping rules (see Figure 2.6(b)) defines actions for packets inside the rectangle. The intersection can also be viewed as the projection of the endpoint policy E on q, denoted as E_q (Figure 2.7(a)). By "packing" E_q on the current switch, all packets falling into q are processed (*e.g.,* dropped or permitted), and the remaining packets are forwarded to the next switch.

Replace: After packing the projection E_q in a switch, we *rewrite* the endpoint policy to avoid re-processing the packets in q: we first add a rule $q^{\text{Fwd}} = (q, \text{Fwd})$ with the highest priority to the policy. The rule q^{Fwd} forwards all the packets falling in q without any mod-



(a) Cover q (b) Switch s_1 (c) After s_1

Figure 2.6: Processing 2-dim endpoint policy E

$r_1 : (q \wedge R2.p, R2.a)$
$r_2 : (q \wedge R3.p, R3.a)$
$r_3 : (q \wedge R4.p, R4.a)$
$r_4 : (q \wedge R6.p, R6.a)$

(a) E_q

$r_1 : (q, \text{Fwd})$
$r_2 : (R1.p, R1.a)$
$r_3 : (R4.p, R4.a)$
$r_4 : (R5.p, R5.a)$
$r_6 : (R6.p, R6.a)$

(b) New rule list

Figure 2.7: Example policy

ification. Second, all internal rules inside q ($R2$ and $R3$) can be safely deleted because no packets will ever match them. The new rewritten endpoint policy and corresponding rule list are shown in Figure 2.6(c) and Figure 2.7(b).

The cover-pack-and-replace operation is recursively applied to distribute the rewritten endpoint policy over the rest of the path. Our heuristic is "greedy": at each switch, we repeatedly pack rules as long as there is rule space available before proceeding to the next switch. We make two observations about the cover-pack-and-replace operation:

- *Whether a feasible rule placement exists becomes clear upon reaching the last switch in the path.* If we can fit all remaining rules on the last switch, then the policy can be successfully implemented; otherwise, no feasible rule placement exists.

- *The total number of installed rules will be no less than the number of rules in the endpoint policy.* This is primarily because only rules inside the rectangle are deleted. A rule that partially overlaps with the selected rectangle will appear on multiple

switches. Secondly, additional rules (q, Fwd) are included for every selected rectangle to avoid re-processing.

2.5.2 Rectangles Searching

Building on the basic framework, we explore *what* rectangle to select and *how* to find the rectangle.

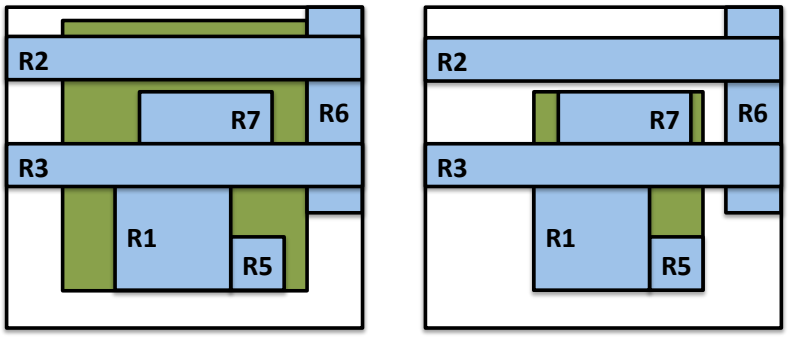
Rectangle selection plays a significant role in determining the efficacy of rule placement. A seemingly natural approach is to find a predicate q that completely covers as many rules as possible, allowing us to remove the most rules from the endpoint policy. However, we must also consider the cost of duplicating the partially-overlapping rules. Imagine we have two candidate rectangles q_1 (with 10 internal rules and 30 overlapping rules) and q_2 (with 5 internal rules and 8 overlapping rules). While q_1 would allow us to delete more rules, q_2 makes more effective use of the rule space. Indeed, we can define the cost-effectiveness of q in a natural way:

$$utility(q) = \frac{\#\text{internal rules} - 1}{\#\text{overlapping rules}}$$

If q is selected, all overlapping rules must be installed on the switch, while only the internal rules can be removed and one extra rule (for q^{Fwd}) must be added[3].

Top-Down search strategy is used in finding the most cost-effective rectangle. We start with rectangle $(\text{src\_ip} = *, \text{dst\_ip} = *)$, and expand the subrectangles $(\text{src\_ip} = 0*, \text{dst\_ip} = *)$, $(\text{src\_ip} = 1*, \text{dst\_ip} = *)$, $(\text{src\_ip} = *, \text{dst\_ip} = 0*)$, and $(\text{src\_ip} = *, \text{dst\_ip} = 1*)$. In the search procedure, we always shrink the rectangle to align with rules, as illustrated by the example in Figure 2.8. Suppose our algorithm selected the predicate p in Figure 2.8(a) (the shadowed one) to cover the rules. We can shrink the predicate as much as possible, as long as the set of rules fully covered by p remains

[3]Cost-effectiveness metrics have been used in other domains to solve covering problems [79].



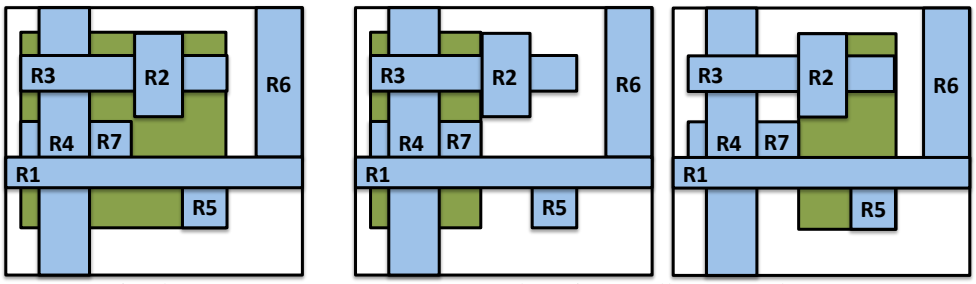
(a) Large cover (b) Small cover

Figure 2.8: Not using unnecessarily large cover.

unchanged. Specifically, we may shrink p as illustrated in Figure 2.8(b), without impacting the correctness of the algorithm. Moreover, for any shrunk predicate, two rules determine the left and right boundaries on the x-axis, resulting in a total of m^2 possible sides along the x-axis. Similarly, the y-axis has a total of m^2 possible sides, resulting in a total number of relevant predicates of m^4.

Even searching $O(m^4)$ predicates in each pack-cover-and-replace operation would be impractical for larger m. To limit the search space, our algorithm avoids searching too deeply, preferring larger rectangles over smaller ones. Specifically, let q be a rectangle and q' be its subrectangle (q' is inside q). When both E_q and $E_{q'}$ can "fit" into the same switch, packing E_q often helps reduce the number of repeated rules. In Figure 2.9, we can use either the large rectangle in Figure 2.9(a) or the two smaller rectangles in Figure 2.9(b). Using the larger rectangle allows us to remove $R3$. Using the two smaller rectangles forces us to repeat $R3$, and repeat $R1$ and $R4$ one extra time. As such, our algorithm avoids exploring all of the small rectangles. Formally, we only consider those rectangles q such that there *exists no* rectangle q' which satisfies both of the following two conditions: (i) q is inside q' and (ii) $E_{q'}$ can be packed in the switch. We call these q the *maximal feasible* predicates.

The pseudo code of the full path heuristic is shown in Figure 2.10. Note that we could have used an existing rule-space partitioning algorithm [32, 49, 72, 78] but they are less effective. The works of [32, 72, 78] take a top-down approach to recursively cut a cover



(a) using larger p (b) using smaller rectangles

Figure 2.9: Only pack maximal rectangle.

PACK-AND-REPLACE-2D(i, E')

1 Let $d_i \leftarrow$ the remaining capacity of s_i.
2 Let Q be the set of maximal feasible predicates with respect to E' that need $\leq d_i$ rules.
3 $q \leftarrow \arg\max_q \{\frac{|\text{Internal}(q,E')|-1}{\|E'_q\|} \mid q \in Q\}$
4 Append the rules E'_q sequentially to the end of the prioritized list of s_i.
5 Let $R \leftarrow \{r \in E' : r.p \text{ is inside } q\}$.
6 $E' \leftarrow E' \backslash R$
7 $E' \leftarrow (q, \text{Fwd}) \circ E'$

COMPILE-RULES-2D$(\{s_1, ..., s_\ell\}, E)$

1 $E' \leftarrow E$
2 **for** $i \leftarrow 1$ **to** ℓ
3 **do** Add a default forward rule for all unmatched packet at s_i.
4 **while** s_i has unused rule space
5 **do** PACK-AND-REPLACE(i, E')

Figure 2.10: Our heuristics for 2-dim chains

into smaller ones until each of the cover fits into one switch. This approach cannot ensure that every switch fully uses its space[4] (See results in Section 2.8). SmartPC [49] takes a bottom-up approach to find the covers. But it searches much less extensively among the set of feasible covers; they do not use the cost-effectiveness metric to control the number of repeated rules either.

[4]For example, imagine at some point in their algorithms, one cover contains $C+1$ rules. Since this cannot fit into one switch, they cut the cover further into two smaller ones. But after this cut, each of the two sub-covers could have only $\approx C/2$ rules, wasting nearly 50% of space in each switch.

2.5.3 Algorithm Generalization

We end this subsection by highlighting a number of extensions to our algorithm.

Smaller flow space: When the flow space for a path is not the full space, we can still use the algorithm except we require that the rectangular covers chosen for the path reside in the corresponding flow space.

Higher dimensions: Our algorithm works when E is a d-dimensional function for a $d \geq 3$. There are two approaches — (i) use"hypercube" predicates instead of rectangular ones in the "cover" phase and cut on all dimensions in searching for predicates. "Packing" and "replace" phases remain the same. Or (ii) still cut along source and destination IP prefix dimensions with all rules projected to rectangles. When we pack a projection of a rectangular predicate q into a switch, all the rules intersect with q are packed and the priorities of the rules are preserved. "Replace" phase is the same. In both ways, our algorithm behaves correctly.

Switch-order independence: Once our algorithm finishes installing rules on switches, we are able to swap the contents of any two switches without altering the behavior of the whole path (Section 2.5.4). This property plays a key role when we tackle the general graph problem.

2.5.4 Correctness

This section proves that our path heuristics is correct.

Proposition 1 *The path heuristics correctly implement the endpoint policy.*

Proof We will prove the proposition by induction on the path length. We also further assume that the first switch selects only one rectangular cover. The analysis for the general case is similar.

Base case: path length $\ell = 1$. In this case, we install the entire policy on the only switch, so the heuristics is correct.

Case: path length $\ell = k+1$. We need more notations. Recall that E is the endpoint policy we want to deploy (we shall also interpret E as a function, *i.e.,*$E(\text{pkt})$ refers to the action needed to be taken when pkt arrives to the ingress). Also, let E_q be the policy deployed at the first switch, *i.e.,*q is the rectangular cover in the first switch. Let E' be the policy rewritten by our path heuristics after it processes the first switch. We need to show that the path correctly implement E. We consider two cases.

Case 1. when pkt $\in q$. We can see that $E_q(\text{pkt}) = E(\text{pkt})$ by the construction of our algorithm. So after pkt visits the first switch, the appropriate actions are taken. Specifically, if $E(\text{pkt})$ is a drop, then the first switch drops the packet. Otherwise, $E(\text{pkt})$ is carried out at the first switch and $E'(\cdot)$ will silently forward pkt. Thus, by induction the composed effect for pkt along the path is exactly identical to E.

Case 2. when pkt $\notin q$. In this case, E_q will simply forward pkt to the second switch. Again, by construction of the path heuristics, $E'(\text{pkt}) = E(\text{pkt})$ in this case. Thus, by induction, appropriate actions will be taken at the rest of the path. Therefore, the composed effect for pkt along the path is again identical to E.

Proposition 2 *Let $R_1, ..., R_\ell$ be the rule placement output by the path heuristics for switch $S_1, ..., S_\ell$. Let* pkt *be any packet. Then there exists exactly one k such that: $R_k(\text{pkt}) = E(\text{pkt})$ and $R_j(\text{pkt}) = \text{Fwd}$ for all $j \neq k$.*

Proof We prove the proposition by induction on path length.

Base case: path length $\ell = 1$. In this case, for any packet pkt, $R_1(\text{pkt}) = E(\text{pkt})$.

Case: path length $\ell = k+1$. Let q be the rectangular cover selected for the first switch. Let E' be the rewritten policy after the first switch. We consider two cases.

Case 1. when pkt $\in q$. We can see that $R_1(\text{pkt}) = E(\text{pkt})$ and $E'(\text{pkt}) = \text{Fwd}$ by the construction of our algorithm. By induction, $R_j(\text{pkt}) = \text{Fwd}$ for all $j > 1$.

Case 2. when pkt $\notin q$. In this case, $R_1(\text{pkt}) = \text{Fwd}$ and $E'(\text{pkt}) = E(\text{pkt})$. Thus, by induction, there exists $k > 1$ such that $R_k(\text{pkt}) = E(\text{pkt})$ and $R_j(\text{pkt}) = \text{Fwd}$ for all $j \neq k$.

We next prove the switch order independence property of our heuristics based on Proposition 2.

Lemma 1 *Let $a_1,...,a_\ell$ be a permutation of $1,2,..,\ell$ and $R_1,...,R_\ell$ be the rule placement output by the path heuristics. If the switches are arranged in the order $S_{a_1},...,S_{a_\ell}$, $R_{a_1},...,R_{a_\ell}$ still implement the endpoint policy E.*

Proof Let us consider a packet pkt and examine what the permuted chain will do on pkt. Let a_k be the switch s.t. $R_{a_k}(\text{pkt}) = E(\text{pkt})$. We can see that the following events happen sequentially:

1. When pkt travels any switches before S_{a_k}, the switch simply forward the packet to the next hop.

2. When pkt arrives at S_{a_k}, the appropriate action will be taken. Specifically, when $E(\text{pkt})$ is a drop action, pkt is dropped at S_{a_k}.

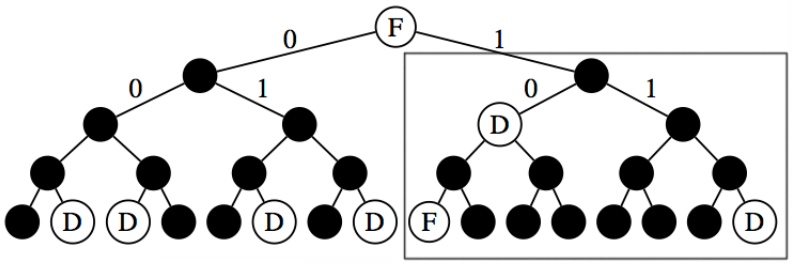
3. If $E(\text{pkt})$ is not a drop action, pkt will continue to travel across the rest of the switches. But all these switches simply forward pkt to the next hop.

Thus, we can see that the action $E(\text{pkt})$ is executed exactly once at S_{a_k}. The effect of going through the chain is identical to the policy E.

2.5.5 Special Case: Single-Dimension Endpoint Policy

In this section, we consider one-dimensional policies E — those that match on a single header field, such as the IP destination prefix. Our goal is to find the minimal capacity C for each of the switch that has a feasible realization of the policy. For this special case, we can achieve provably near optimal results for paths using the same algorithmic framework of the multi-dimensional path heuristics.

For one-dimensional policies, there are known algorithms [75] for computing an optimal list of rules for a *single* switch. The function E can be represented as a prefix-tree



(a) Tree representation of an access-control policy

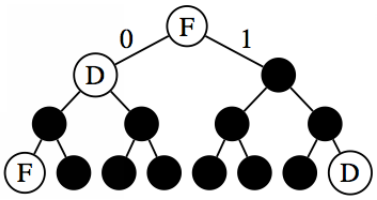
r_1 : (0001 : Drop) r_4 : (0111 : Drop) r_7 : (10$*$: Drop)
r_2 : (0010 : Drop) r_5 : (1000 : Fwd) r_8 : ($*$: Fwd)
r_3 : (0101 : Drop) r_6 : (1111 : Drop)

(b) Prioritized list of rules in a switch

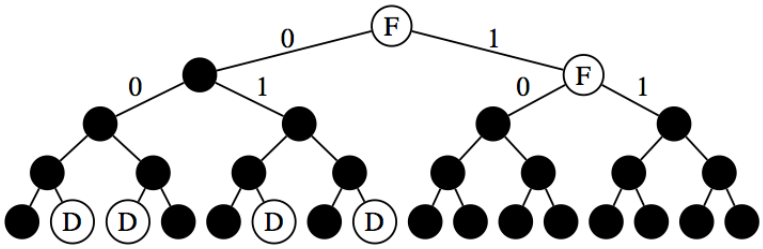
Figure 2.11: Example one-dimensional policy on a switch

on the header field, as shown for an example access-control policy in Figure 2.11(a). Each internal node corresponds to a prefix, and each leaf node corresponds to a fully-specified header value; each prefix can be interpreted as a predicate. A node can specify an action (*e.g.,*forward or drop) for that prefix, as well as descendants that do not fall under a more-specific prefix that includes an action. Computing the prioritized list of rules involves associating each labeled node with a predicate (i.e., the prefix) and an action (i.e., the node label), and assigning priority based on prefix length (with longer prefixes having higher priority), as shown in Figure 2.11(b). Importantly, one can see that the resulting rules will always satisfy the *nesting property*. In other words, given a pair of rules, their predicates will either be disjoint or one a strict subset of the other—there will never be a partial overlap. Moreover, if rules r_1 and r_2 contain predicates p_1 and p_2, respectively, and p_1 is a strict subset of p_2, then r_1 must have higher priority than r_2. (Otherwise, r_1 would be completely covered by r_2 and would go unused.)

Pack-and-Replace on a Prefix Tree: If a single switch cannot store all eight rules in Figure 2.11, we must divide the rules across multiple switches. Our algorithm recursively *covers* a portion of the tree, *packs* the resulting rules into a switch, and *replaces* the subtree with a single node, as shown in Figure 2.12. In the "pack" phase, we select a subtree that



(a) Pack operation



(b) Replace operation (without reoptimization)

Figure 2.12: Pack-and-replace for one-dimensional policies

can "fit" in one switch, as shown in Figure 2.11(a) (the subtree inside the rectangle). If the root of this subtree has no action, the root inherits the action of its lowest ancestor—in this case, the root of the entire tree (see Figure 2.12(a)). The resulting rules are then reoptimized, if possible, before assignment to the switch. In this example, switch 1 would have a prioritized list of four rules—(1000, Fwd), (1111, Drop), (10*, Drop) and (1*, Fwd). More generally, we may pack multiple subtrees into a single switch. To ensure packets in this subtree are handled correctly at downstream switches, we replace the subtree with a single predicate at the root of the subtree (e.g., a single "F" node in Figure 2.12(b)). Then, we can recursively apply the same pack-and-replace operations on the new tree to generate the rules for the second switch.

We now describe our algorithm in further details. As we walk through each switch deciding what rules to place, we maintain an intermediate prioritized list $E' = [r'_1, ..., r'_{n'}]$ to represent the set of unprocessed rules that remain. The set E' starts as E, shrinks as we move along the chain, and ends as a trivial function that forwards all packets.

Our algorithm is "greedy" in packing as many large subtrees as possible, as early in the chain as possible. At the i-th switch, we recursively perform the "pack-and-replace"

PACK-AND-REPLACE(i, E')

1 Let $d_i \leftarrow$ the remaining capacity of s_i.
2 $q \leftarrow \arg\max_q \{\|E_q\| \mid \|E_q\| \leq d_i\}$
3 Append the rules E'_q sequentially to the end of the prioritized list of s_i.
4 $E' \leftarrow E' \backslash E'_q$
5 $E' \leftarrow (q, \text{Fwd}) \circ E'$

COMPILE-RULES$(\{s_1, ..., s_n\}, E)$

1 $E' \leftarrow E$
2 **for** $i \leftarrow 1$ **to** n
3 **do**
4 Add a default forward rule for all unmatched packets at s_i.
5 **while** s_i has unused rule space and E' is non-trivial
6 **do** PACK-AND-REPLACE(i, E')

Figure 2.13: Path heuristics for one-dimensional policy

operation, and have a default rule that forwards all unmatched packets to the next hop. At each step, we pack the largest possible subtree, subject to the switch's capacity constraint, as shown in the pseudocode in PACK-AND-REPLACE in Figure 2.13. We pack as many subtrees as possible in a single switch before proceeding to the next switch, as shown in the inner while loop in COMPILE-RULES in Figure 2.13.

Running time. One can see that a straightforward implementation of our algorithm has time complexity $O(mn \log m)$, where m is the number of rules and n is the number of switches.

Performance analysis We have the following theorem regarding the performance of our one-dimensional algorithm.

Theorem 1 *Consider our path heuristics described above. We have*

- Correctness of the algorithm*: the prioritized rules in the paths correctly implement the policy function* $E = [r_1, ..., r_m]$.

- Approximation ratio*: let C be the minimum capacity so that our algorithm will return a feasible solution. Let C^* be the optimal capacity. We have $\frac{C}{C^*} \leq (1+\varepsilon)$ for any constant ε when m and n are sufficiently large.*

Proof The first part of the Proposition is proved in Lemma 1. We shall focus on the second part. Recall that a lower bound on the optimal capacity C^* for each of the switches is m/n. Thus, we need to show that the capacity C given by our algorithm satisfies $C \leq (1+\varepsilon)m/n$, where ε tends to zero when $m, n \to +\infty$.

We start with the following Lemma.

Lemma 2 *Let C be the capacity of each switch. For a switch s_i, the number of prefixes we can find by invoking* PLACE-RULE(s_i, c_i, q_i, E') *is at most* $\log_2 C$ *is at most* $\log_2 c$.

Proof (Proof of Lemma 2) Let q be an arbitrary prefix. Let q_0 be the prefix obtained by appending a 0 at the end of q (*e.g.,*when $q = 010*$, $q_0 = 0100*$) and q_1 be the prefix obtained by appending a 1 at the end of q. One can see that for any 1-dim policy E,

$$\|E_q\| \geq \|E_{q_0}\| + \|E_{q_1}\| - 1. \tag{2.2}$$

Let us write c_i be the size of available space for s_i while we are packing subtrees for s_i. We claim that when a q in PLACE-AND-PACK is found and packed, either the size of c_i is reduced by at least a half or $q = *$ (*i.e.,*the prefix represents the whole tree). We use an existential argument to prove the claim. Let q be an arbitrary prefix such that $\|E'_q\| > c_i$ while $\|E'_{q_0}\| \leq c_i$ and $\|E'_{q_1}\| \leq c_i$. Such q always exists unless $\|E'\| \leq c_i$, in which case we complete our argument. Now because of $\|E'_q\| > c_i$ and (2.2), one of $\|E'_{q_0}\|$ or $\|E'_{q_1}\|$ must be at least $c_i/2$. Thus, the subtree we found that is packed in s_i has at least $c_i/2$ rules. Therefore, the number of prefixes we can find before $c_i = 0$ is at most $\log_2 c$.

We now continue our analysis by using Lemma 2. Let us define Φ be the sum of the total number of rules deployed in the switch so far and the size of E', which is a changing

variable over the time and is m at the beginning of our algorithm. When our algorithm terminates, one can see that $(n-1)C < \Phi \leq n \cdot C$ (using the fact that all the switches are full except for the last one), *i.e.,*

$$\lceil \frac{\Phi}{n} \rceil = C \tag{2.3}$$

At the point we deploy rules at the i-th switch s_i, Φ changes in the following way:

- Φ is incremented by 1 because we install a default forwarding rule for all unmatched packets.
- Φ is incremented by 2 when we pack a new subtree in a switch.

Thus, the total increment of Φ at s_i is at most $2\log_2 C + 1$. At the end, we have

$$\Phi \leq m + 2(\log_2 C) + n.$$

Together with (2.3) and the fact that $C^* \geq m/n$, we see that $C \leq (1+\varepsilon)C^*$ for any constant ε (when m and n are sufficiently large).

2.6 Decomposition and Allocation

We now describe how we decompose the network problem into paths and divide rule space over the paths.

2.6.1 Decomposition through Cross-Product

We start the "decomposition" by finding all τ paths in the graph $P_1, P_2, ..., P_\tau$, where path P_i is a chain of switches $s_{i_1} s_{i_2} ... s_{i_{\ell_i}}$ connecting one exposed location to another. By examining the "cross-product" of endpoint policy E and routing policy R, we find the flow

space D_i for each P_i. Finally, we project the endpoint policy E on D_i to compute the "path-wise endpoint policy" E_i for P_i. This generates a collection of path problems: for each path P_i, the endpoint policy is E_i and the routing policy directs all packets in D_i over path P_i.

Since each packet can enter the network via a single ingress location and exit at most one egress location (Section 2.2), any two flow spaces D_i and D_j are *disjoint*. Therefore, we can solve the rule-placement problem for each path separately. In addition to solving the τ rule-placement problems, we must ensure that switches correctly forward traffic along all paths, *i.e.,* the routing policy is realized. The algorithm achieves this by placing low-priority *default rules* on switches. These default rules enforce "forward any packets in D_i to the next hop in P_i", such that packets that are not handled by higher-priority rules traverse the desired path.

2.6.2 Rule Allocation through Linear Programming

Ideally, we would simply solve the rule-placement problem separately for each path and combine the results into a complete solution. However, multiple paths can traverse the same switch and need to share the rule space. For each path P_i, we need to allocate enough rule space at each switch in P_i to successfully implement the endpoint policy E_i, while respecting the capacity constraints of the switches. The goal of "allocation" phase is to find a *global* rule-space allocation, such that it is feasible to find rule placements for *all* paths.

Enumerating all possible rule-space partitions (and checking the feasibility by running the path heuristic for each path) would be too computationally expensive. Instead, we capitalize on a key observation from evaluating our path heuristic: the feasibility of a rule-space allocation depends primarily on the *total* amount of space allocated to a path, rather than the *portion* of that space allocated to each switch. That is, if the path heuristic can find a feasible rule placement for Figure 2.4 under the allocation $(c_1 = 4, c_2 = 4, c_3 = 4)$, then the heuristic is likely to work for the allocation $(c_1 = 3, c_2 = 4, c_3 = 5)$, since both allocations have space for 12 rules.

$$
\begin{array}{lll}
\text{max:} & & \perp \\
\text{s.t:} & \forall i \leq n: & \sum_{j \leq \tau} h_{i,j} \cdot x_{i,j} \leq 1 \qquad \text{(C1)} \\
& \forall j \leq \tau: & \sum_{i \leq n} h_{i,j} \cdot x_{i,j} \cdot c_j \geq \eta_j \quad \text{(C2)}
\end{array}
$$

Figure 2.14: Linear program for rule-space allocation

To assess the feasibility of a rule-space allocation plan, we introduce a threshold value η for the given path: if the total rule space allocated by the plan is no less than η ($c_1 + c_2 + c_3 \geq \eta$ in the example), then a feasible rule placement is likely to exist; otherwise, there is no feasible rule placement. Therefore, our rule-space allocation plan consists of two steps: (i) estimate the threshold value η for each path and (ii) compute a global rule-space allocation plan, which satisfies all of the constraints on the threshold values.

This strategy is very efficient and avoids exhaustive enumeration of allocation plans. Furthermore, it allows us to estimate whether *any* feasible solution exists without running the path heuristics.

Estimate the necessary rule space per path: Two factors impact the total rule space needed by a path:

- *The size of endpoint policy:* The more rules in the endpoint policy, the more rule space is needed.
- *The path length:* The number of rectangles grows with the length of the path, since each switch uses at least one rectangle.

Since paths have different endpoint policies and lengths, we estimate the threshold value for the ℓ_i-hop path P_i with endpoint policy E_i. When $\sum_{j \leq \ell_i} c_{i_j} \geq \eta_i$ for a suitably chosen η_i, a feasible solution is likely to exist. In practice, we found that η_i grows linearly with $\|E_i\|$ and ℓ_i. Thus, we set $\eta_i = \alpha_i \|E_i\|$, where α_i is linear in the length of P_i and can be estimated empirically.

Compute the rule-space allocation: Given the space estimates, we partition the capacity of each switch to satisfy the needs of all paths. The decision can be formulated as a

linear programming problem (hereafter LP). Switch s_i can store c_i rules, beyond the rules needed for the default routing for each path. Let η_j be the estimated total rule space needed by path P_j. We define $\{h_{i,j}\}_{i\leq n, j\leq \tau}$ as indicator variables so that $h_{i,j} = 1$ if and only if s_i is on the path P_j. The variables are $\{x_{i,j}\}_{i\leq n, j\leq \tau}$, where $x_{i,j}$ represents the portion of rules at s_i allocated to P_j. For example, when $c_4 = 1000$ and $x_{4,3} = 0.4$, we need to allocate $1000 \times 0.4 = 400$ rules at s_4 for the path P_3. The LP has two types of constraints (see Figure 2.14): (i) capacity constraints ensuring that each switch s_i allocates no more than 100% of its available space and (ii) path constraints ensuring that each path P_j has a total space of at least η_j.

Our LP does not have an objective function since we are happy with any assignment that satisfies all the constraints. [5] Moreover, we apply floor functions to round down the fractional variables, so we never violate capacity constraints; this causes each path can lose at most ℓ_i rules compared to the optimal solution, where the path length ℓ_i is negligibly small.

Re-execution and correctness of the algorithm: When the path algorithm fails to find a feasible solution based on the resource allocation computed by our LP, it means our threshold estimates are not accurate enough. In this case, we increase the thresholds for the failed paths and re-execute the LP and path algorithms and repeat until we find a feasible solution. In the technical report, we show the correctness of the algorithm.

2.6.3 Unwanted Traffic Minimization

One inevitable cost of distributing the endpoint policy is that some unwanted packets travel one or more hops before they are ultimately dropped. For instance, consider an access-control policy implemented on a chain. Installing the entire endpoint policy at the ingress switch would ensure all packets are dropped at the earliest possible moment. However, this solution does not utilize the rule space in downstream switches. In its current

[5] This is still equivalent to standard linear programs; see [62].

form, our algorithm distributes rules over the switches without regard to where the unwanted traffic gets dropped. A simple extension to our algorithm can minimize the cost of carrying unwanted packets in the network. Specifically, we leverage the following two techniques:

Change the LP's objective to prefer space at the ingress switches: In our original linear program formulation, we do not set any objective. When we execute a standard solver on our LP instance, we could get a solution that fully uses the space in the "interior switches," while leaving unused space at the edge. This problem becomes more pronounced when the network has more rule space than the policy needs (*i.e.,* many feasible solutions exist). When our path algorithm runs over space allocations that mostly stress the interior switches, the resulting rule placement would process most packets deep inside the network. We address this problem by introducing an objective function in the LP that prefers a solution that uses space at or near the first switch on a path. Specifically, let ℓ_j be the length of the path P_j. Our objective is

$$\text{max: } \sum_{i \le n} \sum_{j \le \tau} \frac{\ell_j - w_{i,j} + 1}{\ell_j} h_{i,j} x_{i,j}, \tag{2.4}$$

where $w_{i,j}$ is s_i's position in P_j. For example, if s_4 is the third hop in P_6, then $w_{4,6} = 3$.

Leverage the switch-order independence in the path algorithm: At the path level, we can also leverage the switch-order independence property discussed in Section 2.5.3 to further reduce unwanted traffic. Specifically, notice that in our path algorithm, we sequentially pack and replace the endpoint policies over the switches. Thus, in this strategy, more fine-grained rules are packed first and the "biggest" rule (covering the largest amount of flow space) is packed at the end. On the other hand, the biggest rule is more likely to cover larger volumes of unwanted traffic. Thus, putting the biggest rules at or near the ingress will drop unwanted traffic earlier. This motivates us to *reverse the order* we place rules along a chain: here, we shall first pack the most refined rules at the last switch, and progressively

pack the rules in upstream switches, making the ingress switch responsible for the biggest rules.

2.7 Incremental Updates

Network policies change over time. Rather than computing a new rule placement from scratch, we must update the policy *incrementally* to minimize the computation time and network disruption. We focus on the following major practical scenarios for policy updates:

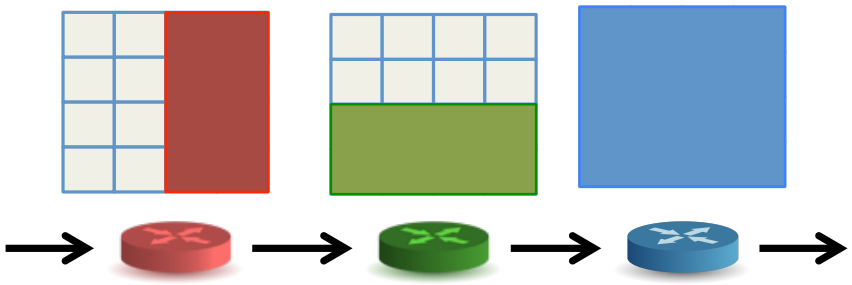
Change of drop or modification actions: The endpoint policy may change the subset of packets that are dropped or how they are modified. A typical example is updating an access-control list. Here, the flow space associated with each path does not change.

Change of egress points: The endpoint policy may change where some packets leave the network (*e.g.,* because a mobile destination moves). Here, the flow space changes, but the routing policy remains the same.

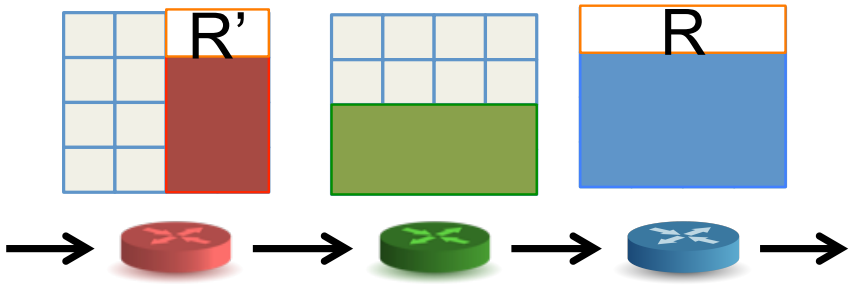
Change of routing policy: When the topology changes, the routing policy also need to be changed. In this case, the network has some new paths, and the flow space may change for some existing paths.

The first example is a "planned change," while the other two examples may be planned (*e.g.,* virtual-machine migration or network maintenance) or unplanned (*e.g.,* user mobility or link failure). While we must react quickly to unplanned changes to prevent disruptions, we can handle planned updates more slowly if needed. These observations guide our algorithm design, which has two main components: a "local algorithm" used when the flow space does not change, and a "global algorithm" used when the flow space does change. [6]

[6] We can use techniques in [64] to ensure consistent updates.



(a) Original placement



(b) Final placement

Figure 2.15: Rule insertion example

2.7.1 Local Algorithm

When the flowspace remains the same (*i.e.,* all packets continue to traverse the same paths), a local update algorithm is sufficient. If a path's policy does not change, the rule placement for that path does not need to change, so we *do not* need to re-execute the path algorithm presented in Section 2.5. We can always convert an original path-wise endpoint policy into the new one by applying one of the following three operations one or more times: (i) insert a new rule, (ii) delete an existing rule, and (iii) alter an existing rule. Thus, we need only design an algorithm to handle each of these operations. Then we may recursively invoke this algorithm to update the policy for the entire path.

Let us focus on rule insertion, *i.e.,* adding a new rule R to the endpoint policy E and the path $P = s_1 s_2 \ldots s_\ell$, as shown in Figure 2.15. Strategies to handle the other two operations are similar. Recall each switch s_i along the path is responsible for some region of flow space, indicated by predicates. In our algorithm, we simply walk through each s_i and see whether

```
INSERT-RULE-PATH({s_1, ..., s_ℓ}, R, E)
1  for i ← 1 to ℓ
2      do Let Q be the set of predicates covered by s_i.
3         for every predicate q ∈ Q
4             do if R.p overlaps with q
5                then Install (R.p ∧ q, R.a) on s_i
6             do if R.p is inside q
7                then return
```

Figure 2.16: Procedure for rule insertion

$R.p$ overlaps with the region ($R.p$ is the predicate of rule R). When an overlap exists, we "sneak in" the projection of R with respect to the region of s_i. Otherwise, we do nothing. Figure 2.16 illustrates the pseudocode.

2.7.2 Global Algorithm

When the flowspace change, our algorithm first changes the forwarding rules for the affected paths. Then we must decide the rule placements on these paths to implement the new policies. This consists of two steps. First, we run the linear program discussed in Section 2.4 *only on the affected paths* to compute the rule-space allocation (notice that rule spaces assigned to unaffected paths should be excluded in the LP). Second, we run the path algorithm for each of the paths using the rule space assigned by the LP.[7]

Performance in unplanned changes. When a switch or link fails, we must execute the global algorithm to find the new rule placement. The global algorithm could be computationally demanding, leading to undesirable delays. [8] To respond more quickly, we can precompute a backup rule placement for possible failures and cache the results at the con-

[7] If the algorithm cannot find an allocation plan leading to feasible rule placements for all affected paths, an overall re-computation must be performed.

[8] In our experiments, we observe the failure of one important switch can cause the recomputation for up to 20% of the paths (see Section 2.8 for details). The update algorithm may take up to 5 to 10 seconds when this happens.

troller. We leave it as a future work to understand the most efficient way to implement this precompute-and-cache solution.

2.8 Performance Evaluation

In this section, we use real and synthetic policies to evaluate our algorithm in terms of (i) rule-space overhead, (ii) running time, and (iii) resources consumed by unwanted traffic.

2.8.1 Experimental Workloads

Routing policies: We use GT-ITM [17] to generate 10 synthetic 100-node topologies. Four core switches are connected to each other, and the other 96 switches constitute 12 subgraphs, each connected to one of the core switches. On average, 53 of these 96 switches lie at the periphery of the network. We compute the shortest paths between all pairs of edge switches. The average path length is 7, and the longest path has 12 hops.

Endpoint policies: We use real firewall configurations from a large university network. There are 13 test cases in total. We take three steps to associate the rule sets with the topology. First, we infer subnet structures using the following observation: when the predicate of a rule list is $(\text{src\_ip} = q_1 \wedge \text{dst\_ip} = q_2)$ (where q_1 and q_2 are prefixes), then q_1 and q_2 should belong to different subnets. We use this principle to split the IP address space into subnets such that for any rule in the ACL, its src_ip prefix and dst_ip prefix belong to different subnets. Subnets that do not overlap with any rules in the ACL are discarded. Second, we attach subnets to edge switches. Third, for any pair of source and destination subnets, we compute the projection of the ACL on their prefixes. Then we get the final endpoint policy E. 4 of the 13 endpoint policies have less than 15,000 rules, and the rest have 20,000–120,000 rules.

In addition, ClassBench [76] is used to generate synthetic 5-field rule sets to test our path heuristic. ClassBench gives us 12 test cases, covering three typical packet-

classification applications: five ACLs, five Firewalls, and two IP Chains. (IP Chain test cases are the decision tree formats for security, VPN, and NAT filter for software-based systems, see [76] for details.)

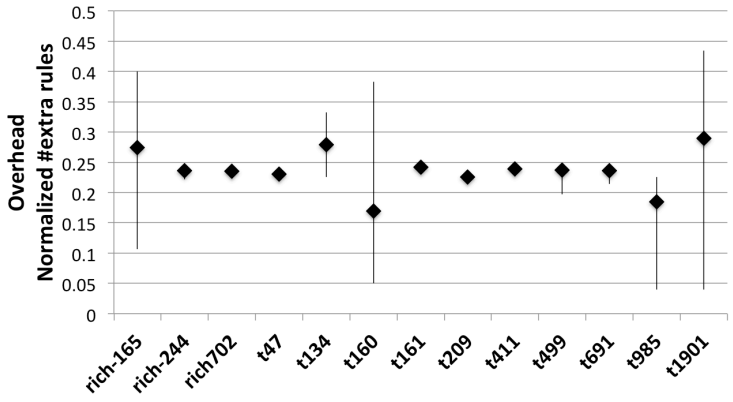
We evaluate our algorithms using two platforms. For stand-alone path heuristic, we use RX200 S6 servers with dual, six-core 3.06 Intel X5675 processors with 48GB RAM. To test the algorithm on graphs, we use the Gurobi Solver to solve linear programs. Unfortunately, the Gurobi Solver is not supported on the RX200 S6 servers so we use a Macbook with OS X 10.8 with a 2.6 GHz Intel Core i7 processor and 8GB memory for the general graph algorithms. Our algorithms are implemented in Java and `C++` respectively.

2.8.2 Rule-Space Utilization

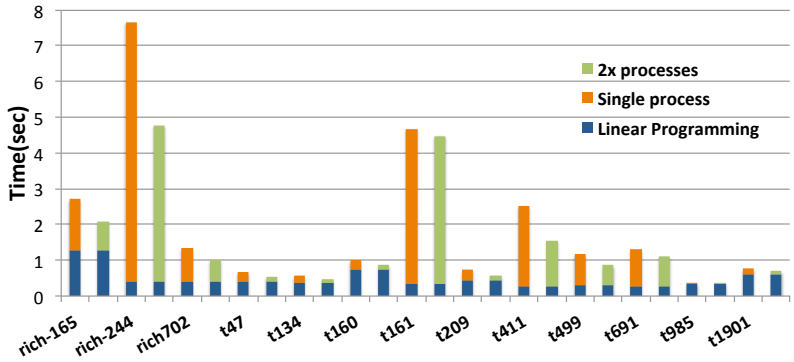
Our evaluation of rule-space utilization characterizes the *overhead* of the algorithm, defined as the number of extra rules needed to implement the endpoint policy E. The overhead comes from two sources:

Decomposition of graph into paths: A single rule in the endpoint policy may need to appear on multiple paths. For example, a rule $(\text{src\_ip} = 1.2.3.4 : \text{Drop})$ matches packets with different destinations that follow different paths; as such, this rule appears in the projected endpoint policies of multiple paths. Our experiments show that this overhead is very small on real policies. The average number of extra rules is typically just twice the number of paths, *e.g.,* in a network with 50 paths and 30k rules in the endpoint policy, the decomposition leads to approximately 100 extra rules.

Distributing rules over a path: Our path heuristic installs additional rules to distribute the path-wise endpoint policy. If our heuristic does a good job in selecting rectangles, the number of extra rules should be small. We mainly focus on understanding this overhead, by comparing to a lower bound of $\|E_i\|$ rules that assumes *no overhead* for deploying rules along path P_i. This also corresponds with finding a solution in our linear program where all α_i's are set to 1.



(a) Rule-space overhead

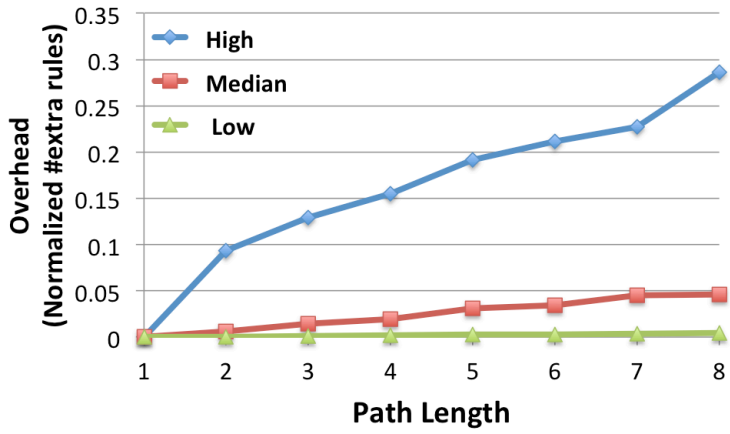


(b) Computation time (for one and two processes)

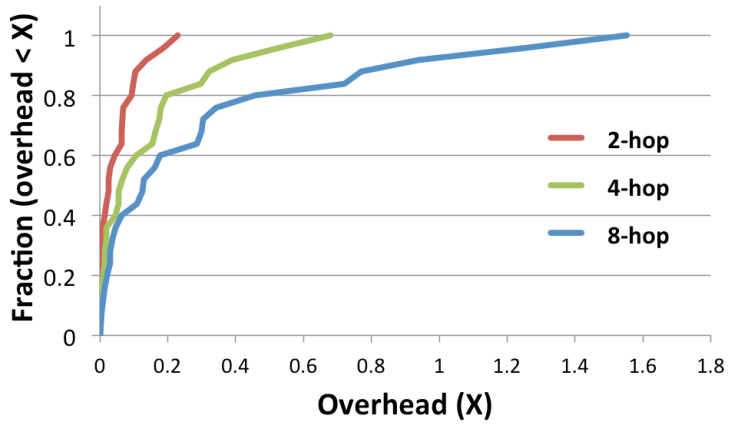
Figure 2.17: The performance of the graph algorithm over different endpoint policies on 100-switch topologies

Our experiments assume all switches have the *same* rule capacity. As such, the overhead is defined as $\frac{C-C^L}{C^L}$, where C is the rule capacity of a single switch, such that our algorithm produces a feasible rule placement, and C^L is the minimum rule capacity given by the LP, assuming no overhead is incurred in the path algorithm.

Results: The overhead is typically between 15% and 30%, as shown in Figure 2.17(a). Even when the overhead reaches 0%, the overhead is still substantially lower than in the strawman solution that places all rules at the first hop [19, 39]—for example, we distribute a policy of 117.5k rules using 74 switches with 2.7k rules, while the first-hop approach needs 32 edge switches with 17k rules each. Figure 2.17(b) shows the running time of our algorithm, broken down into solving the LP and applying the path heuristic. The LP



(a) A large university network data



(b) CDF of all test cases

Figure 2.18: The performance of the path heuristic.

solver introduces a small overhead, and the path heuristic is responsible for the longer delays. Fortunately, the path heuristic can easily run in parallel, with different processes computing rule placement for different paths. Each pair of bars in Figure 2.17(b) compares the running time when using one vs. two processes. The speed-up is significant, except for some policies that have one particularly hard path problem that dominates the running time. The algorithm is fast enough to run in the background to periodically re-optimize rule placements for the entire network, with the incremental algorithm in Section 2.7 handling changes requiring an immediate response.

Evaluating the path heuristic: We also evaluate the path heuristic in isolation to better understand its behavior. These experiments apply the entire endpoint policy to one path of a given length. Figure 2.18(a) plots the rule-space overhead (as a function of path length) for three representative policies (with the lowest, median and highest overhead) from the university firewall data. The median overhead for the 8-hop case is approximately 5% and the worst case is around 28%. For all policies, the overhead grows steadily with the length of the path. To understand the effect of path length, we compare the results for four and eight switches in the median case.

#switches	#rules/switch	#total rules	$\|\|E\|\|$
4	1776	7104	6966
8	911	7288	6966

With eight switches, the number of rules per switch (911) is reduced by 49% (compared to 1776 for four switches). This also means we must search for smaller rectangles to pack rules into the smaller tables. As each rectangle becomes smaller, a rule in the endpoint policy that no longer "fits" within one rectangle must be split, leading to more extra installed rules.

Figure 2.18(b) plots the CDF of the overhead across both the synthetic and real policies for three different path lengths. While overhead clearly increases with path length, the variation across data sets is significant. For 8-hop paths, 80% of the policies have less than 40% overhead, but the worst overhead (from the ClassBench data) is 155%. [9] In this synthetic policy, rules have wildcards in either the source or destination IP addresses, causing significant rule overlaps that make it fundamentally difficult for any algorithm to find good "covering" rectangles. Of the six data sets with the worst overhead, five are synthetic firewall policies from ClassBench. In general, the overhead increases if we tune the ClassBench parameters to generate rules with more overlap. The real policies have lower overhead, since they don't contain many rule overlaps.

[9] Even for this worst-case example, spreading the rules over multiple hops allow a network to use switches with less than one-third the TCAM space than a solution that places all rules at the ingress switch.

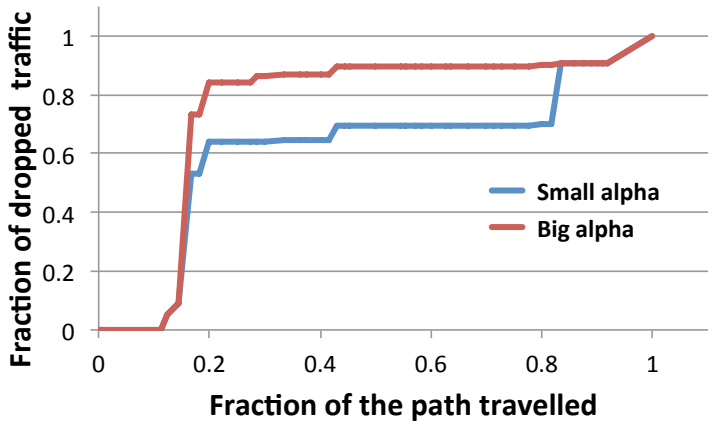


Figure 2.19: CDF of dropped traffic in the graph

Profiling tool: We created a profiling tool that analyzes the structure of the endpoint policies to identify the policies that are fundamentally hard to distribute over a path. The tool searches for all the rectangular predicates in the endpoint policy that can possibly be packed into a single switch. Then, for each predicate q, the tool analyzes the cost-effectiveness ratio between the number of internal rules with respect to q and $\|E_q\|$. The largest ratio here correlates well with the performance of our algorithm. Our tool can help a network administrator quickly identify whether distributed rule placement would be effective for their networks.

2.8.3 Minimizing Unwanted Traffic

We next evaluate how well our algorithm handles unwanted traffic *i.e.,* packets matching a "drop" rule). When ingress switches have sufficient rule space, our LP automatically finds a solution that does not use internal switches. But, when switches have small rule tables, some rules must move to interior switches, causing unwanted packets to consume network bandwidth. Our goal is to drop these packets as early as possible, while still obeying the table-size constraints. We summarize our results using a cumulative distribution function $F(\cdot)$, *e.g.,* $F(0.3) = 0.65$ means that 65% of the unwanted packets are dropped before they travel 30% of the hops along their associated paths.

We evaluate the same test cases in Section 2.8.2 and assume the unwanted traffic has a uniform random distribution over the header fields. Figure 2.19 shows a typical result. We run the algorithm using two sets of α values. When α values are small, LP allocation leave as much unused space as possible; when α values are large, LP allocates more of the available rule space, allowing the path algorithm to drop unwanted packets earlier. In both cases, more than 60% of unwanted packets are dropped in the first 20% of the path. When we give LP more flexibility by increasing α value, the fraction of dropped packets rises to 80%. Overall, we can see that our algorithm uses rule space efficiently while dropping unwanted packets quickly.

2.8.4 Comparison with Palette

We next compare our algorithm with Palette [45], the work most closely related to ours. Palette's main idea is to partition the endpoint policy into small tables that are placed on switches, such that each path traverses all tables at least once. Specifically, Palette consists of two phases. In the first phase (coloring algorithm), the input is the network structure and the algorithm decides the number of tables (*i.e.,* colors), namely k, needed. This phase does not need the endpoint policy information. In the second phase (partitioning algorithm), it finds the best way to partition the endpoint policy into k (possibly overlapping) parts. This phase does not require the knowledge of the graph. We compare the performance of our work and Palette for both the special case where the network is a path and the general graph case. When the network is a path, we only examine the partitioning algorithm (because the number of partitions here exactly equals to the length of the path). Figure 2.20 shows that Palette's performance is similar to ours when path length is a power of 2 but is considerably worse for other path lengths. Moreover, Palette cannot address the scenario where switches have non-uniform rule capacity. Next, we examine Palette's performance over general graphs. Specifically, we execute Palette's coloring algorithm on the general graph test cases presented in Section 2.8.2. The maximum number of partitions found by

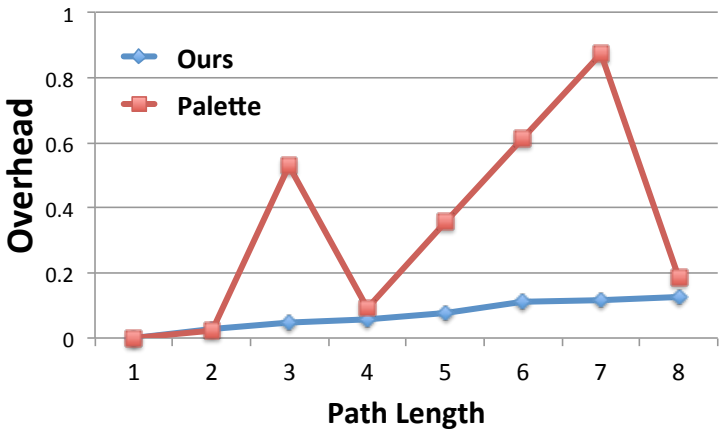


Figure 2.20: Comparing our path heuristic to Palette

their algorithm is four. This means in a test case where an endpoint policy contains 117k rules, Palette requires each switch to contain *at least* $117k/4 \approx 29k$ rules (this assumes no further overhead in their partitioning phase). In contrast, our algorithm produces a solution requiring only 2.5k rules per switch.

2.9 Conclusion

Our rule-placement algorithm helps raise the level of abstraction for SDN by shielding programmers from the details of distributing rules across switches. Our algorithm performs well on real and synthetic workloads, and has reasonable running time. A promising direction for the future work is to explore techniques for handling policies which do not fit within the global network hardware constraints, such as caching [46].

Chapter 3

Alpaca: Compact Network Policies with Attribute-Carrying Addresses

In enterprise networks, policies (*e.g.,* QoS or security) often depend on the categorization of hosts along dimensions such as the organizational role of the host (faculty vs. student), and department (engineering vs. sales). While current best practices (VLANs) help specify policies when hosts are categorized along a single dimension, most policies need to be expressed along multiple orthogonal dimensions.

In this chapter, we introduce *Attribute-Carrying IPs (ACIPs)*, where the IP address allocation process in enterprises considers attributes of a host along all policy dimensions. ACIPs enable flexible policy specification in a manner that may not otherwise be feasible owing to the limited size of switch rule-tables. Further, we present Alpaca, algorithms for realizing ACIPs under practical constraints of limited-length IP addresses. Our algorithms can be applied to different switch architectures, and we provide bounds on their performance. Finally, we demonstrate the importance and viability of ACIPs on data collected from real campus networks.

3.1 Introduction

Managing large enterprise networks is challenging. A typical enterprise has many users who belong to different departments (*e.g.,* sales and engineering, or computer science and history), and play different roles (*e.g.,* faculty, staff, administrators, and students). In addition, the network supports diverse end-hosts running different operating systems and offering different services. In response, network administrators want to enforce policies—such as access control and quality of service—that group hosts along multiple different *dimensions*. For instance, one policy may restrict access to a database to all employees in the sales department, while another may offer a higher bandwidth limit to senior managers across all departments, and yet another may restrict access for old hosts running a less secure operating system.

3.1.1 Enforcing Policies in Today's Enterprises

To enforce policies in today's enterprises, network administrators typically rely on virtual local area networks (VLANs) [85]. A host joining the network is assigned to a VLAN based on its MAC address or the physical port of the access switch. Hosts in the same VLAN are assigned an IP address in the same IP prefix, even if they are not located near each other. Traffic flows freely between hosts in the same VLAN, while traffic between different VLANs traverses an IP router that can enforce policy. Since a host can only belong to a single VLAN, administrators typically assign hosts to VLANs based on a single dimension (*e.g.,* department or role), which has several major limitations:

- The routers interconnecting VLANs need long lists of data-plane "rules" to classify traffic along all relevant "dimensions" of the source and destination hosts.
- No security or QoS policies can be imposed on intra-VLAN traffic, forcing administrators to use a VLAN only to group hosts that should "trust" each other.

- Since VLAN tags are removed from traffic destined to the Internet, the border router must classify all return traffic from the Internet to assign VLAN tags.

For instance, suppose a security policy depends on both user role and department. If VLANs were created based on the user's department, then expressing a policy based on role in a concise fashion is challenging. And, if two users in the different department are close to each other, the traffic follows inefficient paths through intermediate routers to go between VLANs.

The rise of more flexible network switches, with open interfaces to separate control software, enables an attractive alternative. In recent years, switches built with commodity chipsets expose a pipeline of rule-tables that perform match-action processing on packet headers. While the rule tables are relatively small (with small thousands of rules per stage, to limit power and cost), the switches support programmatic interfaces (*e.g.,* OpenFlow, OF-Config, and OVSDB [3]) that enable new ways of controlling the network by installing the rules in support of higher-level policies. One natural approach, adopted in the early Ethane system [19], directs the first packet of each flow to a central controller, which consults a database—containing all the relevant host attributes and high-level policies—and reactively install rules for forwarding the remaining packets of the flow. However, reactive, fine-grained solutions like Ethane have high overhead and do not scale to large enterprises. FlowTags [25] tags packets and uses the tags to enforce network policies. Yet, this approach, when applied to enterprise networks, requires installing many extra tagging rules at the edge switches to classify hosts along all dimensions. Instead, we need a proactive design that can aggregate hosts along many dimensions, while keeping switch rule tables small.

3.1.2 Attribute-Carrying IP Addresses

In the current enterprise networks, although IP addresses are assigned based on attributes (*e.g.,* a separate IP prefix per VLAN), it is just on a *single* dimension. We argue

that a complete IP address management scheme should consider *all* dimensions and enable compact representation of policies. Our key idea is to assign each host an IP address based on *all* dimensions of the policy—that is, an *attribute-carrying IP address* (ACIP). Our solution, Alpaca (algorithms for policies on attribute-carrying addresses), computes *an efficient address assignment*, based on the policy dimensions and the attributes of individual hosts, and *a compact list of switch rules that* realize a specific network policy.

Alpaca proactively generates a small number of coarse-grained rules in the switches, without using VLANs. Alpaca greatly simplifies enterprise management by (i) enabling administrators to specify policies on many orthogonal dimensions while achieving an order of magnitude reduction in rules; (ii) allowing policy to be correctly maintained even when a host connects to the network at a new location; and (iii) simplifying federated management, where different teams manage different parts of the enterprise network.

While Alpaca has many potential advantages, several practical issues must be considered:

Limited IP address space: Since most enterprises have limited IP address space, a naive ACIP assignment can easily exhaust all the available bits.

Heterogenous group sizes: Some combinations of policy attributes are much more common than others. As such, an efficient address assignment cannot simply devote a portion of the bits to representing group size.

Minimizing churn: It is important to support changes in the policy attributes associated with a host (*e.g.,* due to a user moving to a different role or department), or changes in the set of attributes themselves (*e.g.,* due to the creation of a new department).

Multi-stage switch pipelines: Rather than assuming switches have a single rule table (as in OpenFlow 1.0 [51]), a practical solution should capitalize on the multi-stage pipelines in modern switch chipsets [14, 15, 61].

We propose a family of algorithms to generate ACIPs and the associated rule tables, starting with a simple strawman that devotes a separate set of address bits to each policy

dimension. This solution minimizes the number of rules but consumes too much IP address space, making it infeasible in most practical settings. We then propose two other algorithms that can keep the number of rules small, while respecting constraints on the number of bits in the IP address space. Our algorithms optimize based on the characteristics of modern switch chipsets. Conventionally, switch chip-sets have a single TCAM rule-table that supports a few thousands of wildcard rules matching on multiple header fields [8, 38, 51]. More recently, we are seeing the emergence of switch chip-sets with a pipeline of multiple tables, where each table could be a TCAM or a larger SRAM that supports prefix matching on source IP or destination IP [14,15,61]. As such, our first algorithm generates ACIPs that enable policies to be expressed by solely IP *prefixes*, useful for rule tables that support IP prefix matching. Our second algorithm targets both single-table switches and multi-table switches, generating rules that perform arbitrary *wildcard* matching on IP addresses, in exchange for a reduction in the number of rules. Together, these algorithms can capitalize on the unique capabilities of a variety of commodity switch architectures.

In the next section, we present a case study of multiple campus networks, to underscore the need for policies along multiple dimensions. Section 3.3 introduces ACIPs and formulates the optimization problem Alpaca must solve, followed by Section 3.4 that presents our two algorithms. Using access control data from two large campuses, the experiments in Section 3.5 show that Alpaca can reduce the number of ACL rules on existing networks by $60\% - 68\%$ for switches with multiple tables and by $40\% - 96\%$ for switches with a single table, while requiring only 1 more bit of the IP address space than needed to represent the number of hosts in the network. Further, Alpaca can support futuristic scenarios with policies based on multiple dimensions, while requiring an order of magnitude fewer rules than VLAN-based configurations optimized for a single dimension. Section 3.6 presents related work, and Section 3.7 concludes.

3.2 Case Study: Diverse Enterprise Policies

In this section, we present a case study of 25 enterprise networks, to identify the challenges in representing sophisticated policies, and the implications for Alpaca. Specifically:

• We present a qualitative analysis of the security and quality-of-service policies employed by 22 universities, plus one individual department that runs its network separately from the campus IT group. The analysis indicates that networks must often apply policy along several logical *dimensions*, with multiple *attributes* as possible categories in each dimension. However, the analysis also points to policies that are desirable but difficult to realize in practice.

• We analyze router configuration data from two other large campuses. The analysis provides further confirmation that there is significant commonality in policy across hosts, but also points to how an inefficient assignment of IP addresses can lead to an unnecessary "blow-up" in rule-table size.

• We study host-registration data for one department-level network, to understand the dimensions of network policies and the number and size of host "groups" with these attributes. The analysis has important implications for the design of Alpaca.

3.2.1 Policies on Multiple Dimensions

Many universities make descriptions of their high-level network policies available online (see `http://tinyurl.com/pwvlygx` for a summary). Most schools classify hosts by the owner's *role* (*e.g.,* faculty, students, staff, visitors), *department*, *residence* (*e.g.,* a particular dormitory), and *usage* (*e.g.,* research vs. education). In addition, many schools associate each host with a *security level* (with around ten different integer values) and whether the host is currently viewed as *compromised* (with a "yes" or "no" value). Some schools also classify hosts by *bandwidth quota* and *past usage*, to inform rate-limiting policies, and by whether they offer *core services* (*e.g.,* email and web servers). Based on these docu-

ments, and our discussions with the administrators of the computer-science department's network of one university (University A), we learned about the following example policies.

Security: Schools use the security level to limit which external users can access a given host (and in what way). For example, hosts at the lowest security level might be blocked from receiving unsolicited traffic from external hosts; that is, these hosts cannot run public services. Other security levels correspond to different restrictions on which transport port numbers are allowed (*e.g.,* port 80 for HTTP, but not port 22 for SSH or 109, 110, and 195 for POP3). Some schools allow individual departments to state their own access-control lists, applicable only to hosts with IP addresses in that department's address block. When administrators identify an internal host as compromised, they change the *compromised* attribute and significantly restrict the host's access to network services. In addition, users in the *visitor* category typically have access to a limited set of services on the campus (*e.g.,* no access to the printers or campus email servers and compute clusters). One school restricts access to compute clusters in dormitories to the students residing in that particular dorm.

Quality of Service: Some universities impose a different default bandwidth quota based on the host's *role*, but allow students and postdocs to purchase a higher quota. Some universities employ rate-limiting policies that depend on the user's bandwidth usage on previous days (*e.g.,* users whose bandwidth usage exceeded a certain level were rate-limited to a lower level). Hosts offering core services are excluded from bandwidth usage calculations for both the users responsible for the service machines and the owners of the access machines, to avoid that traffic counting against their usage caps. Also, some schools offer higher quality-of-service for hosts assigned for educational use (*e.g.,* for streaming high-quality media in a classroom). The administrators of University A also expressed a desire to perform server load balancing for internal Web services based on user role, to prevent heavy load from one group of users from compromising the performance of other users.

Administrator "wish-lists": Our discussions with the administrators of University A also indicated that there were many additional policies that they would ideally like to implement in the network, but did not do so since they were hard to realize in practice. University A assigns hosts to VLANs based on role (*e.g.,* faculty, staff, and students), for traffic isolation, to prevent packet sniffing and excessive broadcast traffic. The administrators would like to apply access-control policies based on device usage, device ownership, and OS, but do not do so today, since this would require exhaustive enumeration of IP addresses in the switch configuration. Likewise, the administrators expressed a desire to apply flexible QoS policies based on (i) the way a device is used (*e.g.,* research vs. infrastructure machines) and (ii) whether the host is owned by the department (as opposed to a personal "Bring Your Own Device" host).

Our discussions also revealed additional challenges with federated network management. The campus network assigns IP addresses in blocks based on location (*e.g.,* building). This raises challenges in applying security policies that restrict access to users affiliated with computer science department. Currently, the policy works correctly for hosts that are physically in the CS building, since these hosts are assigned a prefix from the CS subnet. However, when a CS user works in another building (common for faculty with dual appointments in other departments), the host receives a different IP address outside the CS subnet and the user is no longer able to access the CS resources. While the administrators could conceivably update network configurations dynamically to reflect the IP addresses that should have access, the management complexity is a deterrent. More generally, federated management would be much easier if network administrators had concise ways to represent security and QoS policies based on host attributes.

3.2.2 Potential for Concise Rules with ACIPs

Existing techniques for assigning IP addresses to hosts can lead to a large numbers of rules in the switches. To quantify this problem, we analyzed the access-control policies

in router configuration files for two university networks (University B and University C). Prior work shows that hosts in a network may be partitioned into a small number of policy units [11]—*i.e.,* a set of hosts that have identical reachability policies in terms of their communication with the rest of the network. Though the number of policy units is small, the number of ACL rules to express policy could still be as large as the square of the number of policy units. Thus, we go beyond [11], and not only identify policy units, but also calculate the number of rules required if ACLs were written in terms of policy units rather than the existing IP assignment.

Specifically, we consider two hosts as belonging to the same source policy unit (SPU) if and only if packets sent by these hosts to all destinations are treated identically in every ACL across all routers. Likewise, we consider two hosts as belonging to the same destination policy unit (DPU) if and only if packets from all sources to these hosts are treated identically in every ACL across all routers. We then compute the total number of rules needed to represent each ACL if it were more compactly expressed in terms of its source and destination policy units. Our results show that the number of ACL rules required is much smaller than the product of the source and destination policy units, and indicates that a smart ACIP allocation, which classifies hosts into their SPUs and DPUs efficiently, can potentially offer significant reduction ranging from 48% to 98% for ACLs of the two universities (Section 3.5).

3.2.3 Diverse Attributes and Group Sizes

To better understand the attributes of hosts, we collected data about the 1491 registered hosts of the CS department of one university (University A). Each host is associated with seven dimensions of information, as summarized in Table 3.1. In this network, (i) hosts are assigned to separate VLANs based on role and (ii) role, security level, and status are considered in access-control policy.

Dimensions	#Attributes	Example Attributes
Role	8	Faculty, Students
Security Level	16	1, 2, ..., 16
Status	6	In service, In testing
Location	7	–
Usage	3	Research, Infrastructure, ...
CS_owned	2	Yes, No
OS	5	MacOS, Windows, Linux, ...

Table 3.1: Host data for CS department (University A)

Given the number of attributes in each dimension, hosts could theoretically have $161,280$ (*i.e.,* $8 \times 16 \times 6 \times 7 \times 3 \times 2 \times 5$) combinations of attributes. In practice, only 287 unique combinations exist; for example, no visitor has a CS_owned host. In addition, some combinations are much more popular than others. One group of hosts—belonging to one Linux-based compute cluster—has 109 members (more than 7% of all hosts). The large number of attributes and the diversity of group sizes have important implications for address assignment in Alpaca, which encodes host attributes in the IP address to enable more compact representations of policies.

Consider a naive address allocation scheme that performs *BitSegmentation*, by (1) concatenating a binary encoding of the host attributes along each dimension, where dimension i with a set of attributes D_i requires $\lceil \log \|D_i\| \rceil$ bits, and (2) using the remaining bits to distinguish hosts with the same attributes along all dimensions, requiring $\lceil \log X \rceil$ bits, where X is the size of the largest group.

The resulting encoding would enable very compact rules in the switches, using wildcard patterns to match on any attribute. However, this solution is impractical, even for this small network. Representing the seven dimensions would require 19 bits, and representing the largest group (with 109 members) would requires 7 bits, for a total of 26 bits—a highly inefficient allocation of IP address space.

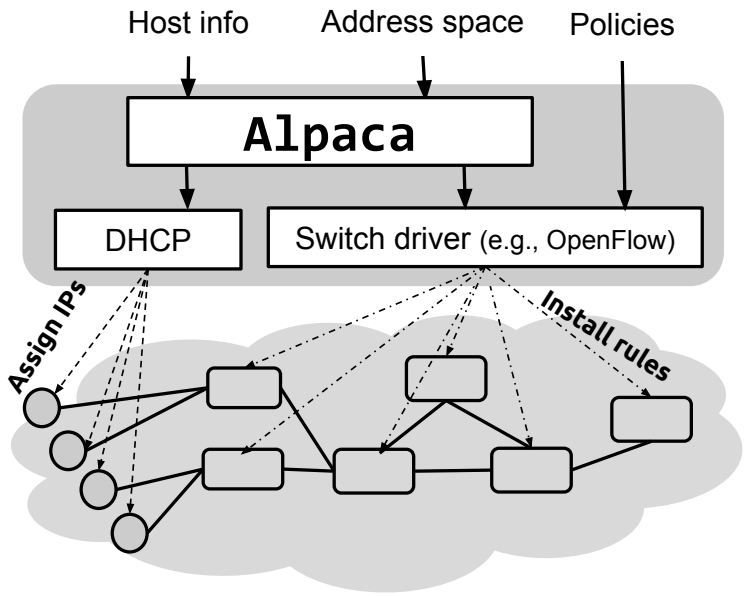


Figure 3.1: Use Alpaca in a network.

3.3 ALPACA Overview

3.3.1 ACIP allocation with Alpaca

We present Alpaca, a system that embeds host attributes in IP addresses to enable compact policies. Figure 3.1 shows an overview of the system. Alpaca takes as inputs from the network operator the set of policy dimensions, and a database that lists the attributes associated with each host. It instructs DHCP servers how to assign IP addresses to hosts based on the results of Alpaca algorithm. If a host needs to be assigned a new address (*e.g.,* it moves to a new location), the original attributes along with the new location are used in determining the new IP address. Meanwhile, the switch driver instantiates the network policies, which are defined on attributes, by installing match-action processing rules on switches. Alpaca coordinates with the switch driver such that rules can correctly classify the IP addresses of hosts to the corresponding attributes along different dimensions.

While ACIP allocation enables operators to express policies defined on multiple dimensions with switch rules, Alpaca must adapt to the following constraints:

Switch rule-table sizes: Switches impose hard constraints on the maximum number of rules to be installed. Therefore, Alpaca should optimize the classification of hosts for different switch architectures so that the attribute-based policies can be compactly expressed with switch rules that stay within the rule-table sizes.

Adddress space: IPv4 is widely deployed and likely to remain for the foreseeable future. Many enterprises have 16 or fewer bits for public IPv4 address space, or up to 24 bits for private IPv4 address space (*i.e.,* 10.0.0.0/8). The limited address space calls for an efficient ACIP allocation that encodes attributes without wasting IPs. Though the address space constraint may be relaxed if IPv6 is fully deployed in a network, the allocation still needs to take the size of address space into account to correctly represent attributes.

Dynamics: Attributes of a host along any dimension may change. Alpaca must be able to handle changes in the attributes of a host while ensuring that only the IP address of that host changes, and that IP addresses of other hosts are not impacted. Alpaca should also handle (1) addition or removal of new attributes in existing dimensions without impacting the existing IP address allocation, and (2) addition or removal of dimensions, which is however relatively rare and may require significant changes.

3.3.2 Problem Formulation

Given an IP address space of W bits, a set U of N hosts ($N \leq 2^W$), a set of M dimensions and the attributes for the hosts along each dimension, an Alpaca algorithm computes an assignment of IPs to individual hosts and M sets of classification rules. Each rule-set corresponds to a dimension. A classification rule consists of an address pattern p and an attribute a; rule (p,a) means that any host with IP matching p has attribute a. The classification rules in the same rule-set have disjoint address patterns. An example is shown in Figure 3.2(a)(b).

The classification rule-sets must be optimized. Consider a multi-table switch architecture. We install the rule-sets to classify source or destination (or both) to the corresponding

attributes for the use by following tables. Figure 3.2(c) shows an example, where the first two tables decide the attributes of "department"and "role" of the source by appending values to metadata, the last table decides to permit or deny the source based on attributes.

Our primary optimization goal is to *minimize the total sizes of M rule-sets,* i.e., *the number of classification rules that decide the attributes of all hosts along all dimensions in a multi-table switch architecture*. We also extend our algorithms (Section 3.4.2) to optimize the rules for a single-table switch architecture, where the installed rule-set is the *product* of all rule-sets in a multi-table architecture (*e.g.,* the three tables in the example) and the total number of installed rules heavily depends on how frequent attributes (or the combination of attributes) are used in the network policies.

3.3.3 Overview of Alpaca algorithms

Alpaca consists of a series of algorithms targeted at different scenarios.

The Prefix algorithm computes prefix classification rules with a proven approximation ratio to the optimal case. It uses address space efficiently, requiring exactly $\lceil log_2 N \rceil$ bits. It is specially designed for multi-table switches with SRAMs that support a large number of prefix rules.

The Wildcard algorithm computes wildcard classification rules. It uses a small address space and can be applied to single-table and multi-table switch architectures.

Both prefix and wildcard algorithms by themselves do not handle dynamics and host attribute changes.

The Slack algorithm refines the prefix and wildcard algorithms by taking advantage of one more bit in the address space for an allocation that works well under dynamics in host attributes.

Input			Output
Hosts	**Dept**	**Role**	**Addresses**
$h_1 - h_5$	CS	Faculty	0000, 0001, 0010, 0011 0111
$h_6 - h_7$	CS	Students	1010, 1011
$h_8 - h_{10}$	EE	Faculty	0100, 0101 0110
$h_{11} - h_{16}$	EE	Students	1000, 1001 1100, 1101 1110, 1111

(a) Address assignment.

Output	
Dept	
p	**a**
0111	CS
101*	CS
00**	CS
0110	EE
010*	EE
100*	EE
11**	EE
Role	
p	**a**
0***	Faculty
1***	Students

(b) Prefix rules.

Dept	
Match src	**Action** append
0111	1
101*	1
00**	1
0110	2
010*	2
100*	2
11**	2

→

Role	
Match src	**Action** append
0***	1
1***	2

→

Match metadata	**Action**
1,2	permit
2,1	permit
*	deny

(c) Rules on multiple tables to permit CS Students and EE Faculty

Figure 3.2: Example allocation: $W = 4, N = 16, M = 2$.

3.4 ALPACA Algorithms

In this section we describe Alpaca algorithms for assigning IPs to individual hosts. The first algorithm is designed for switch chipsets that allow prefix rules while the second solution applies for more general chipsets with tables allowing wildcard rules.

3.4.1 Prefix Solution

This section presents an address allocation algorithm that optimizes the number of *prefix rules* to represent attributes along multiple dimensions. It targets at multi-table switch

architectures with IP prefix matching tables. We first introduce the notation, then discuss the optimal solution for a single dimension and the generalization to multiple dimensions.

We use the following notations when illustrating the algorithms. Let α be a dimension and $A = \{a_1, a_2, ...\}$ be the set of associated attributes. We view α as a function that maps every host to an attribute, *i.e.,* $\alpha(x) \in A$ is the attribute of host x. Let T be an ACIP allocation. We use $C_\alpha(T)$ to denote the minimum number of rules to represent dimension α. Likewise, for a set of dimensions $\mathscr{D} = \{\alpha, \beta, \dots\}$, $C_\mathscr{D}(T)$ represents the total number of rules to present all the dimensions in $\mathscr{D}$ using the allocation T, *i.e.,* $C_\mathscr{D}(T) = \sum_{\phi \in \mathscr{D}} C_\phi(T)$. We define $opt_\alpha = \min_T C_\alpha(T)$ and $opt_\mathscr{D} = \min_T C_\mathscr{D}(T)$ to be the minimum number of rules to represent dimension α and the set of dimensions, respectively.

A single dimension: We start with the simplest case: assigning addresses to represent exactly *one dimension*. Consider the dimension *Role*: each attribute of Role, such as Faculty, Students or Visitors, should have its own set of rules for the hosts. As a prefix pattern matches a power-of-two number of hosts (*e.g.,* 0*** stands for 8 hosts and 111* stands for 2 hosts), one attribute might need several rules. The rules of different attributes do not overlap, i.e., matches are disjoint. Below, we describe a simple algorithm that finds an optimal address allocation to represent one dimension.

Given the dimension function $\alpha : U \to A$, the algorithm returns the address allocation function $T : U \to \{0,1\}^W$. The core idea is to treat the number of hosts for each attribute as the sum of power-of-twos and use a prefix rule for each power-of-two. Specifically, we first partition hosts into $|A|$ sets based on their attributes. Let n_i be the number of hosts with attribute a_i ($i = 1, ..., |A|$) and $bin(n_i)$ be the binary representation of n_i. We represent n_i as the sum of *distinct* power-of-twos based on $bin(n_i)$. For example, $bin(14) = b1110$ and 14 is represented as $8+4+2$. Next, for each attribute a_i, we further partition the set of hosts into subsets according to sum representation of n_i. For example, if $n_i = 14$, we will partition the set of hosts into 3 subsets with size $2, 4$ and 8 respectively. The last step is to sort all the subsets for different attributes in non-increasing sizes. Hosts are ordered based on the

SINGLEDIM(α, W)

1 **for** attribute $a_i \in A$
2 **do** $U_i \leftarrow \{x \in U$, where $\alpha(x) = a_i\}$
3 $n_i \leftarrow |U_i|$
4 **for** $i \leftarrow 0$ **to** W
5 **do if** 2^j & $n_i > 0$
6 **then** Create a subset of 2^j hosts selected from U_i
7 Remove these 2^j hosts from U_i
8 **return** the address allocation of hosts

Figure 3.3: Optimal algorithm for a single dimension α

subsets they belong to. The resulting address allocation gives the k-th host address $bin(k)$. The pseudo-code of the algorithm is shown in Algorithm 3.3.

Let $\|bin(n_i)\|$ be the number of 1s in the binary representation of n_i, *e.g.,* $\|bin(14)\| = 3$. The above algorithm constructs $\sum_{i=1}^{|A|} \|bin(n_i)\|$ subsets. We show that the resulting address allocation needs exactly $\sum_{i=1}^{|A|} \|bin(n_i)\|$ rules for α. In other words, each subset takes a single rule to represent. To prove it, we consider a subset of size 2^i. Since subsets are sorted in non-increasing sizes, any previous subset must have a size of 2^j for some $j \geq i$. Hence, the sum of the sizes of all previous subsets are multiples of 2^i. This guarantees that all 2^i hosts in the current set can be all represented by a single prefix rule with exactly i wildcards. To prove the optimality of the algorithm, we further show that an attribute shared by n_i hosts requires at least $\|bin(n_i)\|$ rules.

Property 1 *For a dimension α, let $n_i = |\{x \in U | \alpha(x) = a_i\}|$ be the number of hosts that have to be mapped to an attribute a_i, $i = 1, 2, \cdots, |A|$. The minimal number of rules that can represent α in any ACIP allocation satisfies $opt_\alpha = \min_T C_\alpha(T) = \sum_{i=1}^{|A|} \|bin(n_i)\|$.*

Two dimensions: Let $\alpha : U \rightarrow A, \beta : U \rightarrow B$ be the dimensions under consideration, where $B = \{b_1, \cdots, b_{|B|}\}$ is the set of attributes in the second dimension. We observe a clear tradeoff between shortening the representation of these two dimensions. While we could choose the address allocation to be the optimal for α and use the minimal number of rules

for α, we may have to use many more rules to represent β. Since the address allocation is shared by both dimensions, in most cases we cannot find an allocation that favors both dimensions. Below, we show the property on the relationship between the optimal allocation for two dimensions and the optimal allocation for each dimension. We recall that the optimal allocation minimizes the sum of the number of rules to represent each dimension.

Property 2 *The optimal allocation for the dimensions α, β satisfies $opt_{\alpha,\beta} \geq opt_\alpha + opt_\beta$. An equality $opt_{\alpha,\beta} = opt_\alpha + opt_\beta$ is achieved if there exists an allocation that is optimal for the dimension α as well as for the dimension β.*

To obtain an upper bound on the optimal number of rules, we construct a special allocation below. Let γ be a new dimension, which is the product of α and β. The corresponding set of attributes $C = \{c_1, \ldots, c_{|A|\cdot|B|}\}$. For a host $x \in U$, if $\alpha(x) = a_i, \beta(x) = b_j$ then $\gamma(x) = c_{(i-1)\cdot|B|+j}$. The dimension γ, denoted by $\gamma = \alpha \times \beta$, has the property that $\gamma(x)$ determines the attributes $\alpha(x), \beta(x)$ for the same host x. Consider the representation of γ under some allocation T with $C_\gamma(T)$ rules. We can obtain a representation of the dimension α (or of β) with the same number of rules by only modifying the attribute of each rule, *i.e.,* replacing attribute in C with the corresponding attribute in A (respectively in B). Therefore, $C_\alpha(T) \leq C_\gamma(T)$, $C_\beta(T) \leq C_\gamma(T)$. Finally,

$$C_\alpha(T) + C_\beta(T) = C_{\alpha,\beta}(T) \leq 2C_\gamma(T) \tag{3.1}$$

We remark here that these are not necessarily the minimum representations of α, β with the address allocation T: we can further compress rules for each dimension.

Next we show that an optimal allocation for γ is *a 2-approximation* of the optimal allocation for α, β, *i.e.,* the number of rules it generates is at most twice of the minimum. Consider some allocation T, it must satisfy

$$C_\gamma(T) \leq C_\alpha(T) + C_\beta(T) = C_{\alpha,\beta}(T) \tag{3.2}$$

This is because a group of hosts that cannot be represented using a single rule in γ must have different attributes in at least one of α and β, thus requiring a minimum of two rules to represent in that dimension. Let T_γ be the optimal solution of γ and $T_{\alpha,\beta}$ be the optimal solution for α, β together. Substituting T with $T_{\alpha,\beta}$ in Equation 3.1 and 3.2, we obtain

$$C_\gamma(T_{\alpha,\beta}) \leq C_\alpha(T_{\alpha,\beta}) + C_\beta(T_{\alpha,\beta}) = C_{\alpha,\beta}(T_{\alpha,\beta}) \leq 2C_\gamma(T_{\alpha,\beta})$$

Meanwhile, since $C_{\alpha,\beta}(T_{\alpha,\beta}) = opt_{\alpha,\beta}$ and $C_\gamma(T_{\alpha,\beta}) \leq C_\gamma(T_\gamma) = opt_\gamma$, we conclude the following property:

Property 3 *Let T_γ be an optimal allocation for $\gamma = \alpha \times \beta$. Then, $C_{\alpha,\beta}(T_\gamma) \leq 2 \cdot opt_{\alpha,\beta}$.*

To summarize, for two given dimensions α, β, we calculate $\gamma = \alpha \times \beta$ and find its optimal allocation T_γ by the algorithm for a single dimension. We then use this allocation to represent each of the dimensions α, β.

General number of dimensions: We can generalize the above solution for two dimensions to handle a set $\mathscr{D}$ of an arbitrary number of dimensions $M = |\mathscr{D}|$. Similar to computing γ for the two dimensions, we introduce a dimension $\Pi_{\mathscr{D}}$, whose attributes for a host $x \in U$ is a vector of length $|\mathscr{D}|$ with the attributes of all dimensions in $\mathscr{D}$ for that host. We show that the optimal allocation for $\Pi_{\mathscr{D}}$ is an M-approximation to the optimal allocation for $\mathscr{D}$. We omit the proof for brevity.

3.4.2 Wildcard Solution

In this section, we present an algorithm that generates wildcard rules by optimizing the output of the prefix solution. To illustrate the algorithm, we use the same example in Figure 3.2. For clarity, we use *host group* to refer the set of hosts with the same attributes along all dimensions; we use *rules* and *patterns* interchangeably to refer the compact ACIP representation of host groups and attributes.

	CS	EE
Faculty	5	3
Student	2	6

(a) Group sizes

$\Rightarrow$

Id	(Dept, Role, Value)
1	CS, F, 1
2	EE, F ,1
3	EE, S, 2
4	CS, S, 2
5	EE, F, 2
6	EE, S, 4
7	CS, F, 4

(b) Initial nodes

Figure 3.4: Create nodes from input of Figure 3.2(a).

We revisit our example. The prefix solution uses 1 pattern for the CS Students group and 2 patterns for each of the rest groups, because it views the size of a host group as the sum of powers-of-twos (*e.g.,* $5 = 1 + 4$), each of which corresponds to a prefix pattern. Hence, it uses $2 + 2 = 4$ prefix rules to represent Faculty attribute. But if we assign $\{0111, 010*\}$ to EE Faculty and $\{0110, 11**\}$ to CS Faculty, then we can compress these patterns to a single pattern *1** to represent Faculty attribute. The key observation is that *if two host groups share common attribute(s), it is beneficial to assign them similar patterns that can be compressed to reduce the number of rules for the common attribute*.

Potential compression. Our first task is to find out all the potential compression of patterns among host groups. Starting with the output of prefix solution, which uses the sum of power-of-two terms to denote the size of a host group, we map every power-of-two term to a *node*. The node saves the value of the term and copies the attributes of the host group. For example, we can create two nodes for the CS Faculty group: (CS, Faculty, 1) and (CS, Faculty, 4), as the group size $5 = 1 + 4$. Figure 3.4(b) shows the full list of nodes.

Two nodes can be compressed if their values are equal and they share some common attribute(s). The result of compression is a new node that (1) has a value equal the sum of the values of the two nodes, (2) "inherits" the shared attributes and (3) has $\emptyset$ attributes for other dimensions. For example, (CS, Faculty, 1) and (EE, Faculty, 1) can be compressed into a new node ($\emptyset$, Faculty, 2). We call the new node a *super-node* and the two original

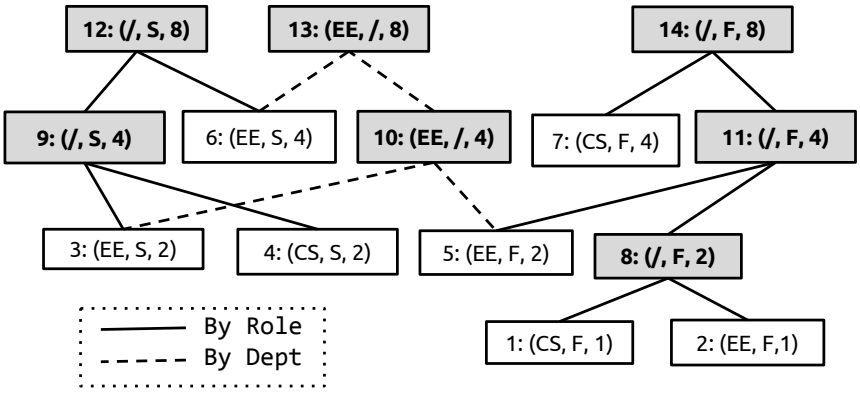


Figure 3.5: The compression graph: a node has an id, attributes and a value. Colored nodes are super-nodes.

nodes *sub-nodes*. The compression suggests that we could use the super-node instead of listing two sub-nodes individually to represent their common attributes.

A super-node can be compressed with other nodes, as long as they share the same attributes (except ∅). But a node cannot be compressed twice on the same dimension, *i.e.,* once (CS, Faculty, 1) and (EE, Faculty, 1) are compressed, neither of them can be compressed with other nodes that have Faculty attribute. We repeat the compression until no new super-nodes can be produced. We plot graphs to denote the compression relationship by creating edges from sub-nodes to their super-nodes (Figure 3.5).

In the graph, a node with value 2^k can be assigned a wildcard pattern with exactly k wildcards, which represent 2^k hosts. As we work on the output of the prefix solution, the initial nodes (Figure 3.4(b)) should be assigned prefix patterns.

Compressible patterns. Two patterns are compressible if they negate at exactly one bit, *e.g.,* *00* and *10* are compressed into **0*. When two sub-nodes (of a super-node) are assigned compressible patterns (*e.g.,* *00* and *10*), the resulting pattern (*e.g.,* **0*) can be assigned to the super-node to achieve a reduction of one rule in representing the common attribute, where we can use the compressed pattern instead of listing two patterns independently. In the example, we can use the pattern for (∅, Faculty, 2) to represent CS attribute rather than two patterns for (CS, Faculty, 1) and (EE, Faculty, 1). Therefore, our

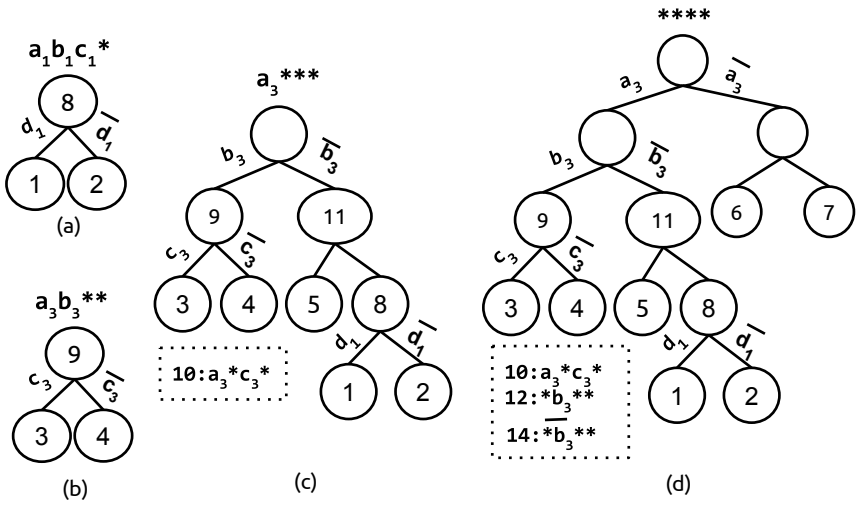


Figure 3.6: Flip bits to compress nodes

goal is to assign patterns to nodes to *maximize the number of pairs of sub-nodes with compressible patterns*, *i.e.,* the total reduction in the number of rules to represent attributes.

Key idea: flip one bit. Let $a_i b_i c_i d_i$ be the pattern assigned to the i-th node, where $a_i, b_i, c_i, d_i \in \{0, 1, *\}$. Consider Node 1 (CS, Faculty, 1) and Node 2 (EE, Faculty, 1). $a_1 b_1 c_1 d_1$ and $a_2 b_2 c_2 d_2$ are compressible if they negate at one bit, *i.e.,* $a_1 b_1 c_1 d_1 = \overline{a_2} b_2 c_2 d_2$, or $a_2 \overline{b_2} c_2 d_2$, or $a_2 b_2 \overline{c_2} d_2$, or $a_2 b_2 c_2 \overline{d_2}$. Our key idea to enable compression is *to choose a bit (*e.g., *a, b, c or d) to flip*. If we flip d, the compressed pattern $a_1 b_1 c_1 *$ (or $a_2 b_2 c_2 *$) can be assigned to the super-node ($\emptyset$, Faculty, 2), *i.e.,* Node 8. As a result, $a_8 b_8 c_8 d_8 = a_1 b_1 c_1 *$. We plot the equality in Figure 3.6(a).

Similarly, we can compress patterns of Node 3 and Node 4 by flipping c ($d_3 = d_4 = *$) as shown in Figure 3.6(b). We can further compress (1) Node 5 and Node 3 by flipping b (as c_3 is flipped before) and (2) Node 5 and Node 8 by flipping c (shown in Figure 3.6(c)). However, we are unable to compress Node 6 ($a_6 b_6 c_6 d_6 = \overline{a_3} b_3 * *$) and Node 10 ($a_{10} b_{10} c_{10} d_{10} = a_3 * c_3 *$), because their patterns do not match. We finish the procedure by compressing Node 6 and Node 9, Node 7 and Node 10 (Figure 3.6(e)). To translate the results to patterns, we set all variables, *i.e.,* a_3, b_3, c_3, d_1, to 0.

Dept	
p	**a**
0110	CS
001*	CS
11**	CS
0111	EE
10**	EE
0*0*	EE

Role	
p	**a**
*1**	Faculty
*0**	Students

Figure 3.7: Wildcard rule-sets.

To summarize, with the idea of bit flipping, we construct *equality and inequality* between the bits of patterns, *i.e.,* a, b, c, d, assigned to nodes. For the patterns of each pair of sub-nodes, if the equality (or inequality) is not determined before, we choose the last possible bit to flip. When there is no such a bit (*i.e.,* all the patterns negating at one bit are already used), then we choose to flip more than one bit until the resulting patterns do not overlap with any used ones. After checking all the pairs of sub-nodes, we obtain the full equality and inequality. The final step is to set all free bit variables to 0.

We can represent each attribute with rules given the pattern assignment (Figure 3.7). For example, to represent Student, we can use *0** for super-node ($\emptyset$, Student, 8), *i.e.,* Node 12. Similarly, we use 10**, 0*0* and 0111 to represent EE. In total, we need $3 + 3 + 1 + 1 = 8$ rules to represent all the attributes, whereas prefix solution needs 9 rules (Figure 3.2(b)).

In what follows, we discuss the order to process pairs of sub-nodes (or super-nodes) to achieve the optimization goal, extend the solution to generate prefix rules and how to handle weighted attributes to support the single-table switch architecture (Section 3.3).

Processing order of sub-nodes. The order we use to process sub-nodes matters, as the compression of one pair of sub-nodes may restrict the compression of another (due to the equality and inequality between bits). The algorithm calculates the *order values* for super-node n as the total number of super-nodes in the tree rooted at n in the graph. For example, the tree rooted at ($\emptyset$, S, 4) only contains one super-node (*i.e.,* itself); the tree rooted at ($\emptyset$, F, 8) contains three. Super-nodes are sorted according to their order values and examined

one by one. When examining one super-node, we process all the pairs of sub-nodes in its tree. If the compression failed for one pair (*i.e.,* we cannot find a bit to flip), we roll back all the previous compressions of sub-node pairs in the tree and continue to examine the next super-node in the sorted list; if the compressions of all pairs of sub-nodes succeed, we remove these sub-nodes from the trees of other super-nodes, re-calculate order values of the affected super-nodes and sort again.

Extension: minimize prefix rules. Although the above algorithm is designed to generate wildcard rules, with a simple trick we could use it to minimize prefix rules as well. The key observation is that wildcard patterns are produced when we choose to flip *non-trailing bits* to compress patterns. For example, when compressing nodes $a_1b_1c_1*$ and $a_2b_2c_2*$, if we choose c_1 then the result a_1b_1** is a prefix pattern, otherwise the pattern (*e.g.,* a_1*c_1* or $*b_1c_1*$) is a wildcard pattern. Hence, to generate prefix rules, we only need to constrain the algorithm to flip the last non-wildcard bit (*e.g.,* c_1 in the pattern $a_1b_1c_1*$).

Extension: weighted attributes. The basic algorithm minimizes the total number of rules to represent all the attributes. But attributes may not be equally important in the single-table switch architecture. For example, Students may be used more often than Faculty. It is preferred to use fewer rules to represent Students despite the increased number of rules to represent Faculty. We can extend the wildcard algorithm to minimize the total number of rules when attributes are weighted. The intuition is to change how the order values of super-nodes are calculated. We introduce the weight of a super-node as the sum of weights of its non-$\emptyset$ attributes. To calculate the order value of a super-node, instead of counting the number of super-nodes in its tree, we sum up the weights of the super-nodes in the tree. The sorting and compression procedure remains the same. We can also handle weighted combinations of attributes (*e.g.,* CS Faculty) with a similar modification to the calculation of weights and order values of super-nodes.

3.4.3 Handle Changes in Host Attributes

Our algorithms proposed so far support address allocation given the attributes of each host. In practice, attributes of a host may change over time (*e.g.,* the department of the corresponding user might change), or new attributes may be added (*e.g.,* a new department may be created). In handling changes, a key consideration is ensuring that only the IP addresses of impacted hosts are modified to the extent possible.

We employ two techniques to handle changes in attributes. First, to handle growth in the number of hosts that have a certain attribute, we introduce *slack,* and budget for more hosts than actually exist. A straight-forward solution is to provision for a growth in the number of hosts corresponding to a given attribute by a fixed percentage (*e.g.,* 10%), though information about projected trends could be used when available. For example, a university can estimate the number of hosts in the coming semester based on the number of newly admitted students.

Second, to handle growth in the number of attributes along each dimension, we introduce a *"ghost" attribute* for each dimension (an additional attributes with which no host is currently associated) and decide the group sizes for combinations of ghost and real attributes (*e.g.,* the number of hosts with ghost department and Students, or the number of hosts with ghost department and ghost role).

Given the input with slacked group sizes and ghost attributes, Alpaca algorithms compute ACIP allocation. When the updates only occur for the existing attributes, we change the addresses of the affected hosts to unused ACIP from the patterns computed for their new attribute. In the case that the provisioned slack of a group is exhausted, we partition the address space of the associated ghost groups, whose attributes are either ghost attributes or attributes of the exhausted group, and allocate part of the space to the exhausted one. For example, if the ACIPs of (Student, CS) are used up, we could partition the address space of (Student, GhostDept), (GhostRole, CS) or (GhostRole, GhostDept) and assign new space to (Student, CS). When the updates involve a new attribute in one dimension, *e.g.,* De-

	CS (9)	EE (8)	Ghost_dept (6)
Faculty (6)	3	1	2
Students (11)	4	5	2
Ghost_role (6)	2	2	2

↓

	CS (**16**)	EE (8)	Ghost_dept (6)
Faculty (**8**)	**5**	1	2
Students (**16**)	**9**	5	2
Ghost_role (6)	2	2	2

Table 3.2: An example of slack

partment, we run Alpaca algorithms on the address space for the ghost attribute to split the space into two parts: one for the new attribute and the other for the ghost attribute. Afterwards, the addresses of affected hosts are changed accordingly.

Benefits of slack and ghost attributes. The above two techniques offer another important advantage: *further compacting network policies beyond the optimal solution*. Consider an example where there are 7 CS hosts and 7 EE hosts. Alpaca needs at least 3 rules for each attribute, as $7 = 4 + 2 + 1$ (Section 3.4.1). With slack, we can round 7 to 8, thus allowing Alpaca to use only 1 rule per attribute. In fact, if we round the group size for every attribute to the nearest power-of-two upper bound, we at most double the number of addresses to use. Namely, *we use at most one extra bit to encode attributes given the slacked group sizes.*

We create extra hosts with "mix-matched" attributes such that the number of hosts for every attribute is power-of-two (Algorithm 3.8). Let p_a be the target power-of-two and g_a be the number of hosts for the attribute a. We choose attribute v_i from i-th dimension such that $p_{v_i} > g_{v_i}, \forall i \in [1, M]$, and create $h = \min_i\{p_{v_i} - g_{v_i}\}$ hosts with attribute $v_1, .., v_M$. When all attributes in a dimension reach their target power-of-two (*i.e.,* $p_a = g_a$), we use the ghost attribute as default, assuming its target power-of-two is infinite. We repeat the procedure until all attributes reach their target power-of-two (except ghost attributes). Consider the example in Table 3.2. The numbers of hosts for CS, Faculty and Students should be rounded

```
SLACK(D, G, P)
 1  while (True)
 2      do for dimension d_i ∈ D
 3             do v_i ← ghost attribute
 4                for attribute a ∈ d_i
 5                    do if P_a > G_a
 6                           then v_i ← a
 7          if ∀i, v_i is ghost attribute
 8            then Break
 9          h ← min_i{P_{v_i} − G_{v_i}}
10          G_{v_i} ← G_{v_i} + h, ∀i
11          Create h hosts with attribute v_1, ..., v_M
12  return the address allocation of hosts
```

Figure 3.8: Slack algorithm

to $16, 8$ and 16. We create 2 CS Faculty hosts in the first iteration and create 5 CS Students afterwards.

For more complex updates that involve additions of new dimensions, it may be desirable to recompute IP allocations from scratch. However, we make several points. First, such scenarios are relatively infrequent. We envision that Alpaca algorithms are run with a conservative set of dimensions, even if some of these dimensions are not currently used as part of network policy. Addition of new dimensions is likely to happen over long time-scales — operators typically collect host attribute information using device registration information filled by owners, and introducing new dimensions would require new data collection for registered devices. Second, when such scenarios do occur, it is feasible to temporarily deal with it by splitting the unused address space of other dimensions and introducing less compact classification rules to identify a given set of hosts. Finally, changes in address allocation can be incrementally handled using DHCP.

3.4.4 Practical Issues

Layer-3 routing. In Alpaca, we consider L3 routing as a policy that forwards packets based on the "location" of their destinations (*e.g.,* the edge switches of the L3 network). Hence, Location is regarded as one dimension in the ACIP allocation. We can run Alpaca to generate classification rules for location dimension, *i.e.,* the rule-set for routing. In some cases, operators may want to pre-assign subnets to the edge switches, *i.e.,* the classification rules for Location are pre-determined. Alpaca can work with the requirement as well. The prefix solution naturally decides the prefix patterns for one dimension after another, it can compute the rules given the pre-assigned prefixes for Location; the wildcard solution can construct the equality and inequality of bits in the patterns for nodes based on the Location prefixes first, and make the later ACIP allocation to comply the prefixes.

Mobility. There are two common solutions to ensure connection affinity when hosts move. One approach is to keep the IPs of end-hosts unchanged and update routing rules instead [41]; the other proposes protocols for end-hosts to maintain connections when both IPs can change [59, 87]. While seamless migration is orthogonal to our work, Alpaca can work well with either approach. In the former case, we do not update Alpaca's classification results, as the attributes of the host is unchanged except location, and the change of location (used by the routing policy) is handled by the proposed solution; in the latter case, we can freely assign a new ACIP to the host based on its updated attributes (including location).

3.5 Evaluation

In this section, we evaluate Alpaca's effectiveness in producing concise rules under two scenarios: (i) actual policies in existing networks and (ii) futuristic scenarios where operators may express policy along many orthogonal dimensions. For existing settings, we evaluate Alpaca using the network configuration files of University B and University C

	#ACLs	Total #Rules	#SPU	#DPU
University B	13	17868	577	624
University C	5027	32401	523	87

Table 3.3: Network policies of two universities.

(Section 3.5.1). For futuristic settings, we use the host attribute data obtained from University A (Section 3.5.2). Details of both data-sets were presented in Section 3.2.

Overall, our results show that Alpaca can reduce the number of rules by $60\% - 68\%$ and $40\% - 96\%$ as compared to the current IP address allocation for multi-table switches and single-table switches, respectively. Meanwhile, it has the potential to reduce the total number of rules by over an order of magnitude as compared to the traditional single dimensional approaches (*e.g.,*VLAN) in futuristic scenario where the policy is expressed on many dimensions. Our experiments further demonstrate that Alpaca can handle changes in hosts gracefully, with only a small extra number of rules.

Our evaluations explore the performance of both Alpaca variants: Prefix (ALP_PFX) and Wildcard (ALP_WC), and for comparison purposes we also consider the BitSegmentation scheme (BitSeg). Unless otherwise mentioned, both our prefix and wildcard algorithms use the algorithm (with prefix extension) in Section 3.4.2 and the extension with slack in Section 3.4.3 by default. We evaluate the schemes for multi-table and single-table switches. However, our evaluations with single-table switches is limited to the wildcard algorithm, since the prefix algorithm can only be applied to the multi-table architectures with prefix matching tables.

3.5.1 Benefits with Existing Policies

Alpaca for multi-table switches

We extract the source and destination policy units (SPUs and DPUs) from the low-level configuration files for both University B and University C, as discussed in Section 3.2. Table 3.3 shows the total number of ACL rule-sets, ACL rules across rule-sets, and the

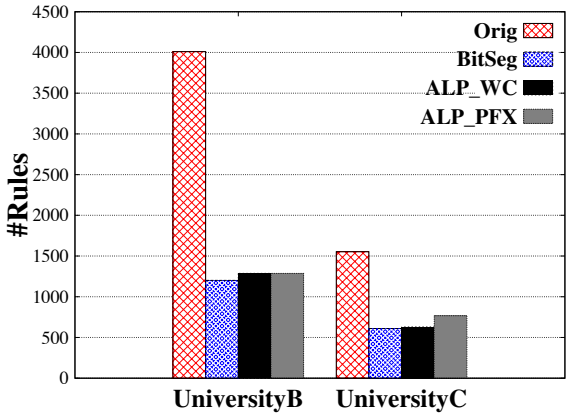
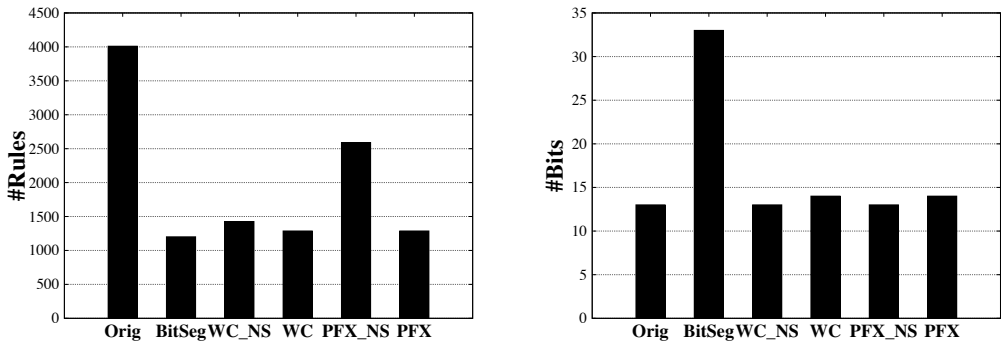


Figure 3.9: Optimize network policies on mutli-table switches.

number of SPUs and DPUs for both universities. Given a pipeline of tables, we install the classification rules that associate a given IP with its appropriate SPU and DPU in the first two tables, and the actual policy action (*e.g.,* permit or deny) based on the SPUs and DPUs in the last table. We focus on the number of classification rules in the first two tables for a given policy, since the last table is the same in all approaches.

Figure 3.9 compares the number of rules used by (1) the original IP allocation (Orig), (2) BitSeg, (3) ALP_PFX and (4) ALP_WC for University B and University C, respectively. The original IP allocation needs the most rules. BitSeg takes the least, as it uses one rule for each policy unit. Specifically, the number of rules used by BitSeg equals the number of SPUs and DPUs. Both ALP_PFX and ALP_WC perform closely to BitSeg, achieving 68% reduction in rule consumption as compared to the original. It confirms that Alpaca can efficiently encode policy units.

Benefits of slack. We compare the case with and without slack operations to show the benefits of trading an extra bit for significant reduction in number of rules. Figure 3.10(a) presents the reduction in the number of rules for University B. We use NS to indicate running Alpaca without slack. While WC_NS (3rd bar) is competitive with other approaches, PFX_NS (5th bar) performs slightly worse, giving a reduction of 35.4%. The reasons are two-fold: for one thing, prefix patterns fundamentally restrict the potential of using fewer



(a) Rules (b) Bits

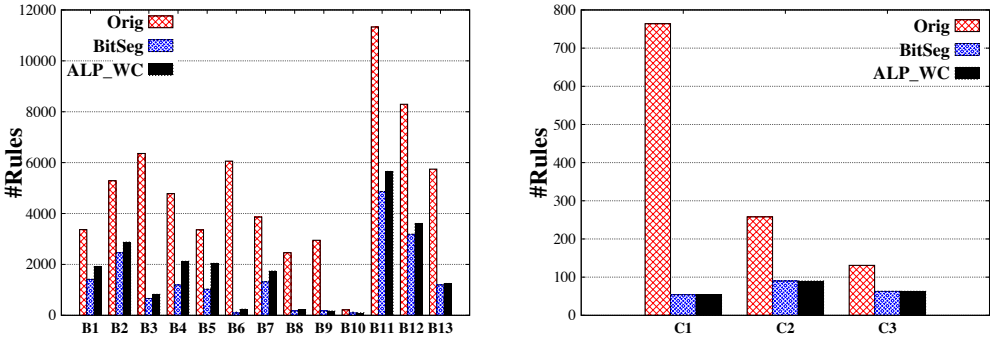
Figure 3.10: Benefits of slack. BS, WC and PFX denote Wildcard, Prefix and BitSegmentation schemes. NS indicates variant without slack.

rules (as compared to wildcard patterns); for the other, PFX_NS solutions represent the exact group size of every combination (*i.e.,* each SPU and DPU pair) without any slack. If the group size is not power-of-two, PFX_NS solutions need many more rules. We bridge the gap by adding slack and rounding group sizes. As a result, PFX offers similar performance to the optimal (*i.e.,* BitSeg). In the remainder of the evaluation, we run slack algorithm before IP allocation by default.

Moving to the number of bits for encoding (Figure 3.10(b)), we find BitSeg performs the worst, using as many as 34 bits (more than IPv4!). We calculate the least number of bits that can sufficiently number all the hosts (13 for University B). While the original allocation uses the least bits, PFX_NS takes exactly one bit more as the slack algorithm makes use of an extra bit to round group sizes (Section 3.4.3). Alpaca strikes a balance in both the number of rules and bits, using almost as few rules as BitSeg and one extra bit than the least number of bits.

Alpaca for single-table switches

Generally, a switch with a single table takes more rule space than the one with multiple tables to implement the same policy, because the rules installed in the former case are the cross-product of rules in the multiple tables in the latter case. Alpaca uses the frequency



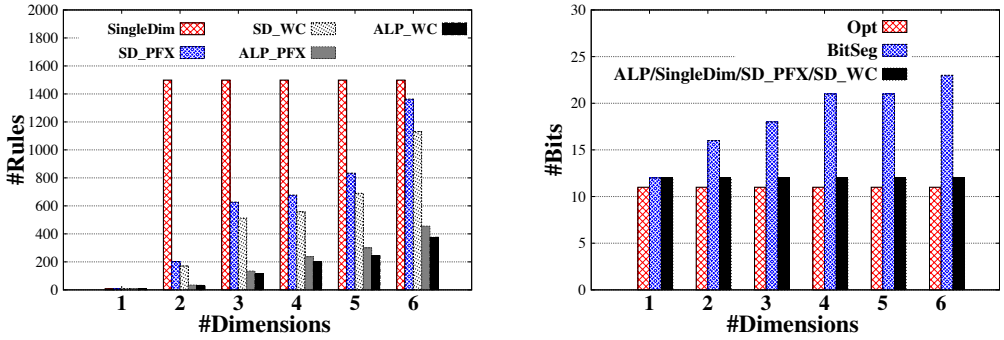
(a) University B (b) University C

Figure 3.11: Optimize network policies on single-table switches.

of attributes (*i.e.,* SPU or DPU) used in ACLs to minimize the resulting rules. We compare three approaches: ALP_WC, BitSeg and the original IP allocation. Figure 3.11(a)(b) demonstrate the effectiveness of Alpaca in compacting large ACL rule-set. Alpaca wildcard compacts the original policies by $40\% - 96\%$, competitive with BitSeg. We would like to point out that the original ACLs are written with respect to the resource constraints of the deployed switches in the networks. As a result, all the original ACLs could fit into the switches. But even so, the reduction by Alpaca is significant. It suggests that with Alpaca, the network operators can use cheaper switches with smaller rule-tables to support today's policies, or plan for larger policies in the future with the current switches.

3.5.2 Benefits with Futuristic Policies

We demonstrate Alpaca's capability to support flexible attribute-based policies with a series of experiments on the host information at the CS department of University A (Table 3.1). In the current CS network, operators deploy VLANs to group hosts with the same Role, which is used in most network policies. But they would like to use Security Level, Status and Operating System for access control and have flexible QoS policies defined on Usage, CS_owned as well. Hence, we examine the cost in terms of rules and address space



(a) Rules (b) Bits

Figure 3.12: Encode attributes with increased #dims.

to support the futuristic scenarios, where policies are defined on attributes along multiple dimensions.

We compare Alpaca with three approaches:

(1) *SingleDim* (*e.g.,* VLAN), which assigns addresses based on a single dimension. SingleDim uses a few rules to represent attributes for one dimension: VLAN uses one rule (the subnet) for each Role attribute; a host is assigned a random address in the subnet corresponding to its Role attribute. However, given a second dimension or more, SingleDim has to enumerate every single host and list their attributes.

(2) *SD_PFX*, which applies an optimal algorithm [53] to minimize the number of prefix rules for attributes, given the SingleDim address assignment.

(3) *SD_WC*, which uses an efficient heuristic [54] to compute the wildcard rules to represent attributes based on the SingleDim address assignment, as minimizing wildcard rules is NP-hard [50].

We remark that SD_PFX and SD_WC minimize the number of rules by assuming rule priority. Both methods generate overlapping rules for different attributes. In contrast, Alpaca generates non-overlapping rules for different attributes, i.e., does not apply rule priority. Below, we show that even without using rule priority, Alpaca significantly outperforms the two compression methods.

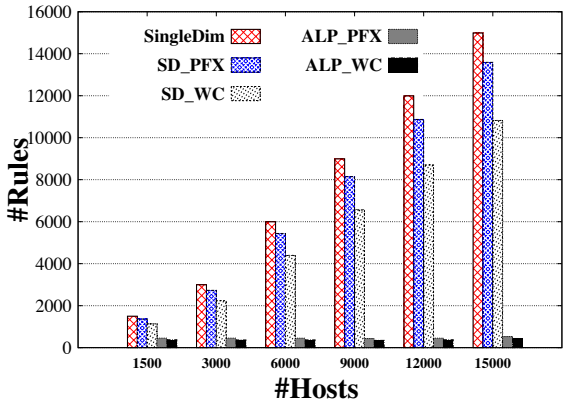


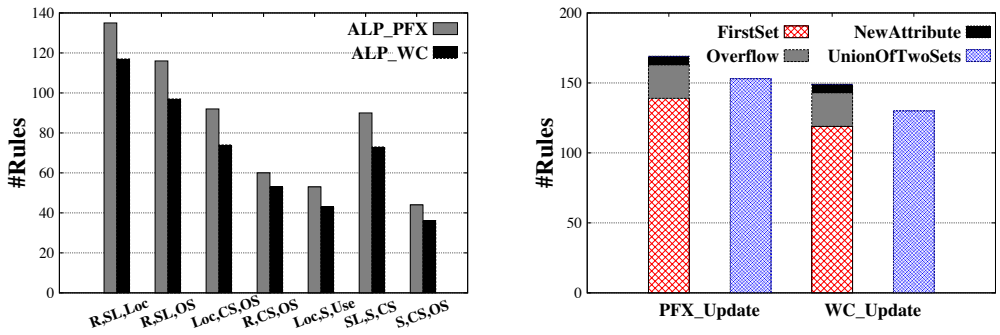
Figure 3.13: Encode attributes with increase #hosts.

Scale with more dimensions. We evaluate Alpaca's encoding efficiency and scalability with increasing number of dimensions. Six dimensions are chosen (in order): Role, Security Level, Location, Status, CS_owned and Usage. The initial set of dimensions only contains Role. Then, in each iteration, we add one more dimension to the current set and run Alpaca algorithms to generate classification rules. Figure 3.12(a) plots the number of rules generated by SingleDim, SD_PFX and SD_WC and Alpaca over the six iterations. Given one dimension, all approaches generate a small number of rules. Moving to two dimensions (*i.e.,* Role and Security Level), SingleDim has to potentially enumerate hosts and their attributes, taking as many rules as the number of hosts in the data. The number of rules used by SingleDim is unchanged over the iterations then. SD_PFX and SD_WC generates less rules than SingleDim, as they apply compression algorithms for a smaller rule set to represent attributes. Yet, when there are six dimensions, the number of rules (1130 wildcard rules and 1363 prefix rules) is very close to the number of hosts. In contrast, both Alpaca prefix and wildcard scales well with increasing number of dimensions. Alpaca uses 376 wildcard rules or 456 prefix rules for six dimensions, which are significantly smaller. We show the number of bits in Figure 3.12(b). Both Alpaca and SingleDim approaches use 12 bits, while the least number of bits (denoted as Opt) is 11 ($1491 < 2^{11}$). BitSeg is infeasible in practice after two dimensions, as it takes more than 16 bits to encode attributes.

Scale with more hosts. Our data only covers a single department, but an entire university (with dozens of departments) has more hosts and enterprises can be even larger in sizes. We examine Alpaca's scalability with more hosts, by synthesizing the host information. We copy each host 2 to 10 times and obtain the scaled-up host data. Figure 3.13 compares several approaches to classify hosts in 6 dimensions. Alpaca scales well, using 436 wildcard rules or 528 prefix rules for around 15000 hosts. Its performance is very stable, due to the use of aggregated patterns (*e.g.,* wildcard or prefix matches) to classify groups of hosts. The number of hosts does not impact its performance. In contrast, SingleDim potentially needs 15000 rules to enumerate every host; it does not scale to larger networks. The compression algorithms do not help much: SD_PFX and SD_WC need 13589 and 10821 rules, respectively, because the single dimension based allocation does not help the aggregation on other dimensions.

Encode different sets of dimensions. While Alpaca performs well for three dimensions: Role, Security Level and Location (as shown above), we are curious about its performance on a different set of three dimensions, such as Role, CS_owned and Operating System. Hence, we fix the number of dimensions to encode and run the algorithm on various sets of dimensions. Figure 3.14(a) shows the performance of Alpaca to encode seven sets of three dimensions. We do observe the fluctuation: the number of rules generated by Alpaca ranges from 36 to 117 for wildcard case and 44 to 135 for prefix case. Upon closer examination, we find out that the performance is highly correlated with the possible combinations of attributes. Specifically, for the set of dimension {Role, Security Level, Location}, there are 80 combinations of attributes which at least one host is associated with; for the set {Status, CS_owned, OS}, there are only 22 combinations. Given increasing numbers of combinations of attributes, Alpaca is more likely to generate many rules.

Update the assignment for new hosts. We divide hosts into two equal-sized sets based on their created time and run Alpaca to encode four dimensions: Role, Security Level, Status and Usage. We use the first set for the initial hosts and the second set for the newly



(a) Different sets of dims (b) Update assignment

Figure 3.14: Property of Alpaca algorithm

added hosts. For the first set, Alpaca provisions slack and creates ghost attributes for all dimensions. The second set not only inserts more hosts with the existing attributes but also introduces 4 more new attributes in Security Level. To assign addresses to new hosts with existing attributes, Alpaca uses the slack in the corresponding group. But if the group size is insufficient, Alpaca has to "steal" flow space from the related ghost groups (Section 3.4.3). To handle the new attributes, Alpaca splits the address space of ghost attributes in the same dimension as well. In our evaluation (Figure 3.14(b)), the first set (left red bar) uses 139 prefix rules or 119 wildcard rules to represent the four dimensions. Fixing the assignment for the first set, we calculate the extra rules needed to handle the second set. The extra rules come from two parts: (1) hosts with new attributes and (2) overflowed group sizes. As a result, we need an extra 6 rules for the new attributes and 24 rules for the overflowed groups. The overhead is very small compared to 153 prefix rules or 130 wildcard rules for the union of the two sets, where the assignment is computed from scratch without any incremental updates.

3.6 Related Work

Rule optimization: Minimizing prefix rules matching one header field is easy [23], but minimizing prefix rules or wildcard rules in general cases is NP-hard [50, 53]. Optimal

solutions are developed to minimize prefix rules in special cases [9, 23, 60, 66, 75]. Heuristics [47,53,68] are presented to compress rules in general cases. In particular, [55] suggests to decompose a single rule list into a pipeline of rule lists to minimize the total number of rules. All of these works take the rule-set as an input and explore the potential for minimization, which, in fact, is limited by the original (unoptimized) address allocation. In comparison, Alpaca *generates* the rule list as an output through a smart address allocation process to minimize the number of rules.

Address permutation: Wei et al. propose to swap addresses between two blocks of users to reduce the number of rules [82], but the algorithm can only handle up to two dimensions. Meiners et al. use permutation of the bits in addresses to create prefix patterns so compression algorithms can apply [54]. But it only discovers the optimization potential within the original address allocation.

Information encoding: Huffman coding encodes attributes of a single dimension using prefixes, but its goal is to minimize the weighted sum of the prefix lengths of all attributes. Hence the prefixes do not match the group sizes. SoftCell [41] embeds two dimensional information, *i.e.,* location and middlebox service chain, in the NAT-ed IP addresses. Its encoding mechanism is a special case of BitSegmentation. Algorithms to encode forwarding rules with minimum bits are proposed in [65, 67].

Attribute-based policy enforcement: Ethane [19] proposes to implement access control at the network edge by directing the first packet of every flow to a controller, which consults the attributes of hosts and install microflow rules on the switch. FlowTags [25] tag packets based on host attributes and match tags to enforce network policies. Another approach, NetAssay [22], supports network traffic monitoring policies by pushing specific switch-rules for each host given their current IPs. All these work do not optimize IP allocation and install many host-specific rules.

3.7 Conclusion

In this paper, we have made three contributions. First, we show the importance and feasibility of considering attributes in IP address allocation. Second, we present the Alpaca system, and two algorithms which cope well with constraints on the IP address space, enterprise churn, and heterogeneity in group sizes. When evaluated with configuration data from two universities, ALP_WC and ALP_PFX reduce the number of rules by $50\% - 68\%$ and $60\% - 68\%$ respectively for multi-table switches. Further, the algorithms have the potential to reduce the total number of rules by over an order of magnitude compared to single dimension based IP address allocation schemes. This can in turn lower the barriers for network administrators to express richer policies involving multiple dimensions.

Chapter 4

Niagara: Efficient Traffic Splitting on Commodity Switches

Traffic often needs to be split over multiple equivalent backend servers, links, paths, or middleboxes. For example, in a load-balancing system, switches distribute requests of online services to backend servers. Hash-based approaches like ECMP have low accuracy due to hash collision and incur significant churn during update. In a Software-Defined Network (SDN), the accuracy of traffic splits can be improved by crafting a set of wildcard rules for switches that better match the actual traffic distribution. The drawback of existing SDN-based traffic-splitting solutions [13, 33, 70, 81] is poor scalability as they generate too many rules for small rule-tables on switches.

In this chapter, we present the design of Niagara, an SDN-based traffic-splitting scheme that achieves accurate traffic splits while being extremely efficient in the use of rule-table space available on commodity switches. Niagara uses an incremental update strategy to minimize the traffic churn given an update. We prototype Niagara for a hybrid load balancer comprising of a hardware switch and multiple software switches and propose an efficient update scheme to preserve connection affinity. Experiments demonstrate that Niagara (1) achieves nearly optimal accuracy using only $1.2\% - 37\%$ of the rule space

of the current state-of-art, (2) scales to tens of thousands of services with the constrained rule-table capacity and (3) offers nearly minimum churn.

4.1 Introduction

Network operators often spread traffic over multiple components (such as links, paths, middleboxes, and backend servers) that offer the same functionality or service, to achieve better scalability, reliability, and performance. Managing these distributed resources effectively requires a good way to balance the traffic load, especially when different components have different capacity. Rather than deploying dedicated load-balancing appliances, modern networks increasingly rely on the underlying *switches* to split load across the replicas [4, 7, 28–30, 63, 81, 88]. For example, server load-balancing systems [28, 63] use hardware switches to spread client requests for each service over multiple software load balancers, which in turn direct requests to backend servers. Another example is multi-pathing [37, 77, 88], where a switch splits the flows with the same destination over multiple paths.

The most common traffic-splitting mechanism is ECMP [37, 77], which is available in most commodity switches and widely used for load balancing [28, 63] and multi-pathing purposes [4, 30]. ECMP splits a set of flows (typically flows with the same destination prefix) *uniformly* over a group of next-hops based on the hash values of the packet-header fields. Weighted-Cost Multi-Path (WCMP) [88] is an extension of ECMP that supports a weighted splits by repeating the same next-hop multiple times in an ECMP group. ECMP and WCMP both partition the flow space, assuming equal traffic load in each hash bucket. The splitting accuracy of ECMP degrades significantly due to hash collision [5, 13]. Furthermore, ECMP incurs unnecessary traffic shifts during updates. When a next-hop is added or removed in ECMP, any hash function shifts at least 25% to 50% of the flow space to a different next-hop [37].

In this paper, we are interested in designing a generic and accurate traffic-splitting scheme for commodity switches. The emergence of open interfaces to commodity SDN switches such as OpenFlow [16, 51], enables operators to have a controller that installs rules on switch rule-tables to satisfy the load-balancing goals [13,33,81]. These rule-tables (*e.g.,* TCAM) are optimized for high-speed packet-header matching, however they have small capacities on the order of a few thousand entries [8,38,68]. The simplest SDN-based solution [33] directs the first packet of each flow to the controller, which reactively installs an exact-match (microflow) rule on the switch. More efficient approaches [13, 26, 69, 81] proactively install wildcard rules that direct packets matching the same header patterns to the same next-hop, but they do not use the rule-table space efficiently and cannot scale to large networks.

This paper presents Niagara, an efficient traffic-splitting scheme that computes switch rules to minimize traffic *imbalance* (*i.e.,* the fraction of traffic sent to the "wrong" next-hop, based on the target load balancing weights), subject to rule-table constraints. Niagara handles multiple *flow aggregates*—sets of flows with the same destination or egress. Each flow aggregate is splitted according to distinct target weights. Our experiments demonstrate that Niagara scales to tens of thousands of flow aggregates and hundreds of next-hops with a small imbalance. After a brief discussion of traffic-splitting use cases and related work (Section 4.2), we present the traffic-splitting optimization problem and a high-level overview of Niagara (Section 4.3).

We make the following contributions.

Efficient traffic-splitting algorithm: Niagara approximates load-balancing weights accurately with a small number of wildcard rules. For each flow aggregate, Niagara can flexibly trade off accuracy for fewer rules (Section 4.4). Niagara packs rules for multiple flow aggregates into a single table, and allows sharing of rules across multiple aggregates with similar weights (Section 4.5). Given an update, Niagara computes incremental changes

to the rules to minimize churn (*i.e.,* the fraction of traffic shuffled to a different next-hop due to the update) and traffic imbalance (Section 4.6).

Realistic prototype: We implement the Niagara OpenFlow controller and deploy the controller (i) in a physical testbed with a hardware Pica8 switch interconnecting four hosts and (ii) in Mininet [34] with Open vSwitches [3] and a configurable number of hosts. We recently conducted a live demonstration of Niagara at an SDN-based Internet eXchange Point (IXP) in New Zealand [1], where Niagara load balanced DNS and web requests to backend servers in a production environment. In Section 4.7, we present the design and implementation of a Niagara-based load balancer with an efficient update scheme to preserve connection affinity.

Trace-driven large-scale evaluation: We evaluate the performance of Niagara for server load balancing and multi-path traffic splitting through extensive simulation against real and synthetic data and validate the simulation results subject to the limitations of our prototype (Section 4.8). Experiments demonstrate that Niagara (1) achieves nearly optimal accuracy outperforming ECMP and other SDN-based approaches, (2) scales to tens of thousands of aggregates using as little as $1.2\% - 37\%$ of the rule space compared to alternative solutions and (3) handles update gracefully with nearly minimal churn.

4.2 Traffic split background

4.2.1 Use cases

We provide three examples that illustrate how hardware switches are used to split traffic over next-hops.

Server load balancing. Cloud providers host many services, each replicated on multiple servers for greater throughput and reliability. Load balancers (*e.g.,* Ananta [63], Duet [28]) rely on hardware switches to spread service requests over servers. Ananta uses switches to forward requests over software load balancers (SLB), which then send requests

to backends; Duet requires switches to distribute requests to backends for popular services directly, besides forwarding to SLBs. Depending on the server capacity and deployments (*e.g.,* server allocation in racks, maintenance and failures), a switch is required to spread requests evenly or in a weighted fashion [28, 81]. Both Ananta and Duet use hash-based traffic-splitting schemes (Section 4.2.3).

Data center multi-pathing. Data center topologies [4,30,88] offer many equal-length paths that switches can use to increase bisection bandwidth. In a fully symmetric topology, a switch splits traffic of each destination prefix equally over available paths. A recent study [88] found that data-center topologies tend to be asymmetric due to failures and heterogeneous devices. In such a topology, a switch should split traffic in proportion to the capacity of the equal-length paths.

Wide area traffic engineering. Wide Area Networks (WAN) carry a huge amount of inter-datacenter traffic. WAN traffic engineering systems (TE) establish tunnels among data center sites and run periodic algorithms to optimize the bandwidth allocation of tunnels to different applications. The underlying switches should split traffic for each application over tunnels according to the algorithm's results for the best network utilization. Existing TE solutions (*e.g.,* SWAN [36], B4 [40]) use hash-based approaches as their default traffic-splitting schemes (Section 4.2.3).

4.2.2 Requirements

Accuracy. Traffic-splitting schemes should be accurate. Commodity servers can handle a limited number of requests; an inaccurate traffic split can easily overload a server, thus incurring long latencies and request failures. In the network, inaccurate splits create congestion and packet loss.

Scalability. The scheme should scale. Data centers host up to tens of thousands of services (*i.e.,* flow aggregates), which are collectively handled by a handful of SLBs (*i.e.,* next-hops); multi-path routing requires an ingress switch to handle hundreds of destination

prefixes (*i.e.,* flow aggregates) and dozens of paths (*i.e.,* next-hops). A scalable traffic-splitting scheme should handle the heterogeneity in the numbers of flow aggregates and next-hops, given the constraints in rule-table capacity.

Update efficiency. Failures or changes in capacity require updating the split of flow aggregates. However, transitioning to this new split comes at some cost of reshuffling packets among servers (*i.e.,* churn). This requires extra work to ensure consistent handling of TCP connections already in progress [21, 63, 64]. A good traffic-splitting scheme needs to be updatable with limited churn.

4.2.3 Prior Traffic-Splitting Schemes

Hash-based approaches. ECMP aims at an equal split over a group of next-hops (*e.g.,* SLBs) by partitioning the flow space into equal-sized hash-buckets, each of which corresponds to one next-hop. WCMP handles weighted splits by repeating next-hops in an ECMP group, thus assigning multiple hash-buckets to the same next-hop. ECMP is available on most commodity switches, which gives rise to its popularity [28, 36, 40, 63]. However, it splits the *flow space* equally, rather than the actual traffic. It is common that certain parts of the flow spaces (*e.g.,* a busy source) contribute more traffic than others [5, 12, 30]; an even partition of the flow space does not guarantee the equal split of traffic. Moreover, the size of the ECMP table, which is a TCAM with hundreds to thousands rules on commodity switches [88], severely restricts the achievable accuracy of WCMP. Finally, updating an ECMP group unnecessarily shuffles packets among next-hops. It is shown that when a next-hop is added to a $N-1$-member group, at least $\frac{1}{4}+\frac{1}{4N}$ of the flow space are shuffled to different next-hops [37], while the minimum shuffle is $\frac{1}{N}$.

SDN-based approaches. SDN supports programming rule-tables in switches, enabling finer-grained control and more accurate splitting. Aster*x [33] directs the first packet of each flow to a controller, which then installs micro-flow rules for forwarding the remaining packets, making the controller load and hardware rule-table capacity quickly become

bottlenecks. MicroTE [13] proactively decides routing for every pair of edge switches (*i.e.,* ToR-to-ToR flows in a data center), but still generates many rules. A more scalable alternative installs coarse-grained rules that direct a consecutive chunk of flows to a common next-hop. A preliminary exploration of using wildcard rules is discussed in [81]. Niagara follows the same high-level approach, but presents more sophisticated algorithms for optimizing rule-table size, while also addressing churn under updates. We discuss [81] in detail in Section 4.4.1.

Other approaches for multi-pathing. The traffic-splitting problem has been studied extensively in the past in the context of multi-pathing. LocalFlow [70] achieves perfectly uniform splits, but cannot produce weighted splits and may split a flow, causing packet reordering. Conga [6] and Flare [44] load balance flowlets (bursts of packets within a flow) to avoid reordering but require advanced switch hardware support. In comparison, Niagara load balances traffic without packet reordering using off-the-shelf OpenFlow switches. An alternative approach to these schemes is centralized flow scheduling such as Hedera [5]. Hedera reroutes "elephant" flows based on global information. Niagara could provide the default routing scheme for a centralized flow-scheduler which then installs specific flow-rules for elephant flows. The third type of approaches is host-controlled routing, which changes the paths of packets by customizing extra fields in ECMP hash functions [43] or round-robin forwarding to intermediate switches [18]. Niagara does not directly compete with these approaches by design, as it does not touch the end-hosts.

4.3 Niagara Overview

Niagara generates wildcard rules to split the traffic within the constrained rule-table size. Incoming traffic is grouped into flow aggregates, each of which is divided over the same set of next-hops according to a weight vector. The per-aggregate weight vector is calculated with consideration on the bandwidth of both downstream links and capacity of

Match		Action
DIP	SIP	Next-hop
63.12.28.42	∗0	17.12.11.1
63.12.28.42	∗	17.12.12.1
63.12.28.34	∗00100	17.12.11.1
63.12.28.34	∗000	17.12.11.1
63.12.28.34	∗0	17.12.12.1
63.12.28.34	∗	17.12.13.1

(a) Load balancing two services.

Match	Action
DIP	Tag
63.12.28.42	1
63.12.28.53	1
63.12.28.27	1
63.12.28.34	2
63.12.28.43	2

$\Longrightarrow$

Match		Action
Tag	SIP	Next-hop
1	∗0	17.12.11.1
1	∗	17.12.12.1
2	∗00100	17.12.11.1
2	∗000	17.12.11.1
2	∗0	17.12.12.1
2	∗	17.12.13.1

(b) Grouping and load balancing five services.

Figure 4.1: Example wildcard rules for load balancing.

next-hops. In the load balancing example, incoming packets are grouped by their destination IPs (*i.e.,* services). Traffic of each service is divided over next-hops (*i.e.,* SLBs) according to their capacity (*e.g.,* bandwidth, CPU, the number of backend servers they connect to). Figure 4.1(a) shows an example of wildcard rules generated by Niagara for load balancing. Each rule matches on destination IP to identify the service and source IP to forward packets to the same SLBs. Packets are forwarded based on the first matching rule. In addition to wildcard rules, Niagara leverages the metadata tags supported by latest chip-sets [16] and generates tagging rules to group services of similar weight distributions, thus further reducing the number of rules (Figure 4.1(b)).

In this section, we formulate the optimization problem for computing wildcard rules in the switch and outline the five main components of our algorithm. For easy exposition of the rule generation algorithm, we use *suffixes of source IP address* and assume a proportional split of the traffic over suffixes (*e.g.,* ∗0 stands for 50% traffic). We relax this assumption in Section 4.4.1.

4.3.1 Rule Optimization Problem Formulation

The algorithm computes the rules in the switch, given the per-aggregate weights and the switch rule-table capacity. A hardware switch should approximate the target division of traffic over the next-hops accurately. The misdirected traffic may introduce congestion over downstream links and overload on next-hops. As such, an important challenge is to minimize the *imbalance*—the fraction of traffic that routes to the "wrong" next-hops.

The weights of each aggregate vary due to differences in resource allocation (*e.g.,* bandwidth), next-hop failures, and planned maintenance. Each aggregate v has non-negative weights $\{w_{vj}\}$ for splitting traffic over the M next-hops $j = 1,2,\ldots,M$, where $\sum_j w_{vj} = 1$. (Table 4.1 summarizes the notation.) The traffic split is not always exact, since matching on header bits inherently discretizes portions of traffic. In practice, splitting traffic *exactly* is not necessary, and aggregates can tolerate a given error bound e, where the actual split is w'_{vj} such that $|w'_{vj} - w_{vj}| \leq e$. The value of e depends on the deployment: an aggregate with a few next-hops requires a smaller e value (usually in $[0.001, 0.01]$). Ideally, the hardware switch could achieve w'_{vj} with wildcard rules. But small rule-table sizes thwart this, and instead, we settle for the lesser goal of approximating the weights as well as possible, given a limited rule capacity C at the switch.

To approximate the weights, we solve an optimization problem that allocates c_v rules to each aggregate v to achieve weights $\{w'_{vj}\}$ (*i.e.,* $c_v = numrules(\{w'_{vj}\})$). Aggregate v has traffic volume t_v, where some aggregates contribute more traffic than others. We define the total imbalance as the sum of over-approximated weights. The goal is to minimize the total

Variable	Definition
N	Number of aggregates ($v = 1, \ldots, N$)
M	Number of next-hops ($j = 1, \ldots, M$)
C	Hardware switch rule-table capacity
w_{vj}	Target weight for aggregate v, next-hop j
t_v	Traffic volume for aggregate v
d_v	Traffic distribution for aggregate v over the flow space
e	Error tolerance $\lvert w'_{vj} - w_{vj} \rvert \leq e$
w'_{vj}	Actual weight for aggregate v, next-hop j
c_v	Hardware rule-table space for aggregate v

Table 4.1: Table of notation, with inputs listed first.

traffic imbalance, while approximating the weights:

$$
\begin{aligned}
&\text{minimize } \textstyle\sum_v (t_v \times \sum_j E(w'_{vj} - w_{vj}, e)) \quad s.t. \\
&\qquad w'_{vj} \geq 0 \qquad\qquad \forall v, j \\
&\qquad \textstyle\sum_j w'_{vj} = 1 \qquad\qquad \forall v \\
&\qquad c_v = numrules(\{w'_{vj}\}) \qquad\qquad \forall v \\
&\qquad \textstyle\sum_v c_v \leq C \\
&\text{where} \quad E(x, e) = \begin{cases} x & \text{if } x > e \\ 0 & \text{if } x \leq e \end{cases}
\end{aligned}
$$

given the weights $\{w_{vj}\}$, traffic volumes $\{t_v\}$, rule-table capacity C, and error tolerance e as inputs.

4.3.2 Overview of Optimization Algorithm

Our solution to the optimization problem introduces five main contributions, starting with the following three ideas:

Approximating weights for a single aggregate (Section 4.4.1): Given weights $\{w_{vj}\}$ for aggregate v and error tolerance e, we compute the approximated weights $\{w'_{vj}\}$ and the

associated rules for each aggregate. The algorithm expands each weight w_{vj} in terms of powers of two (*e.g.,* $\frac{1}{6} \approx \frac{1}{8} + \frac{1}{32}$) that can be approximated using wildcard rules.

Truncating the approximation to use fewer rules (Section 4.4.2): Given the above results, we can truncate the approximation and fit a *subset* of associated rules into the rule table. This results in a *tradeoff curve* of traffic imbalance versus the number of rules.

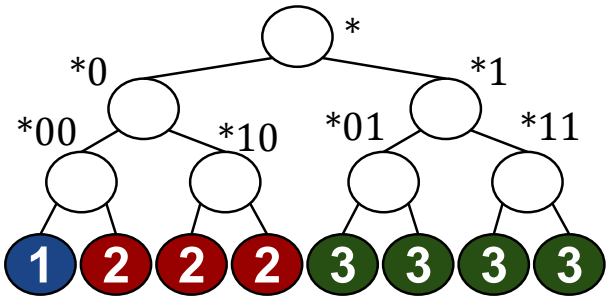
Packing multiple aggregates into a single table (Section 4.5.1): We allocate rules to aggregates based on their tradeoff curves to minimize the total traffic imbalance. In each step of the packing algorithm, we allocate one more rule to the aggregate that achieve the highest ratio of the *benefit* (the reduction in traffic imbalance) to the *cost* (number of rules), until the hardware table is full with a total of $C = \sum_v c_v$ rules. Consequently, more rules are allocated to aggregates with larger traffic volume and easy-to-approximate weights.

Together, these three parts allow us to make effective use of a small rule table to divide traffic over next-hops.

Thousands of aggregates with dozens of next-hops can easily overwhelm the small wildcard rule table (*i.e.,* TCAM) in today's hardware switches. Fortunately, today's hardware switches have multiple table stages. For example, the popular Broadcom chipset [16] has a table that can match on destination IP prefix and set a metadata tag that can be matched (along with the five-tuple) in the subsequent TCAM. Niagara can capitalize on this table to map an aggregate to a tag—or, more generally, *multiple* aggregates to the same tag. Our fourth algorithmic innovation uses this table:

Sharing rules across aggregates with similar weights (Section 4.5.2): We associate a tag with a group of aggregates with similar weights over the same next-hops. We use k-means clustering to identify the groups, and then generate one set of rules for each group. Furthermore, we create a set of default rules of low priority, which are shared by all groups.

Transitioning to new weights (Section 4.6): In practice, weights change over time, forcing Niagara to compute *incremental* changes to the rules to control the churn.



(a) Suffix allocation

Pattern	Action
$*000$	fwd to 1
$*100$	fwd to 2
$*10$	fwd to 2
$*1$	fwd to 3

(b) Naive approach

Pattern	Action	Priority
$*000$	fwd to 1	high
$*0$	fwd to 2	low
$*1$	fwd to 3	low

(c) Use subtraction and priority

Figure 4.2: Naive and subtraction-based rule generation for weights $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ and approximation $\{\frac{1}{8}, \frac{3}{8}, \frac{4}{8}\}$.

4.4 Single Aggregate Optimization

We begin with generating rules to approximate the weight vector $\{w_{vj}\}$ of a single aggregate v within error tolerance e. We then extend the method to account for constrained rule-table capacity C.

4.4.1 Approximate: Binary Expansion

Naive approach to generating wildcard rules. A possible method to approximate the weights [81] is to pick a fixed suffix length k and round every weight to the closest multiple of 2^{-k} such that the approximated weights still sum to 1. For example by fixing $k = 3$, weights $w_{v1} = \frac{1}{6}$, $w_{v2} = \frac{1}{3}$, and $w_{v3} = \frac{1}{2}$ are approximated by $w'_{v1} = \frac{1}{8}$, $w'_{v2} = \frac{3}{8}$, and $w'_{v3} = \frac{4}{8}$. The visualized suffix tree is presented in Figure 4.2(a). To generate the corresponding wildcard rules, an approximate weight $b \times 2^{-k}$ is represented by b k-bit rules. In practice, allocating similar suffix patterns to the same weight may enable combining some of the

Iteration	$\mathbf{w}'_{v1}$	$\mathbf{w}'_{v2}$	$\mathbf{w}'_{v3}$
0	0	0	1
1	0	$\frac{1}{2}$	$1-\frac{1}{2}$
2	$\frac{1}{8}$	$\frac{1}{2}-\frac{1}{8}$	$1-\frac{1}{2}$
3	$\frac{1}{8}+\frac{1}{32}$	$\frac{1}{2}-\frac{1}{8}-\frac{1}{32}$	$1-\frac{1}{2}$

(a) Approximation iterations

Pattern	Action	Corresponding terms
$*00100$	fwd to 1	$\frac{1}{32}$ in w'_{v1} and $-\frac{1}{32}$ in w'_{v2}
$*000$	fwd to 1	$\frac{1}{8}$ in w'_{v1} and $-\frac{1}{8}$ in w'_{v2}
$*0$	fwd to 2	$\frac{1}{2}$ in w'_{v2} and $-\frac{1}{2}$ in w'_{v3}
$*$	fwd to 3	1 in w'_{v3}

(b) Wildcard rules

Figure 4.3: Wildcard rules to approximate $(\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$

rules, hence reducing the number of rules. The corresponding wildcard rules are listed in Figure 4.2(b).

Shortcomings of the naive solution. The naive approach always expresses b as the "sums" of power of two (for example $\frac{3}{8}$ is expressed as $\frac{2}{8}+\frac{1}{8}$) and only generates non-overlapping rules. In contrast, our algorithm allows *subtraction* as well as longest-match *rule priority*. In the above example, $\frac{3}{8}$ can be expressed as $\frac{4}{8}-\frac{1}{8}$ to achieve the same approximation with one less rule (Figure 4.2(c)). The generated rules overlap and the longest-matching rule is given higher priority: $*000$ is matched first and "steals" $\frac{1}{8}$ of the traffic from rule $*0$.

The power of subtractive terms and rule priority. Our algorithm approximates weights using a series of *positive and negative* power-of-two terms. We compute the approximation $w'_{vj} = \sum_k x_{jk}$ for each weight w_{vj} subject to $|w'_{vj} - w_{vj}| \leq e$. Each term $x_{jk} = b_{jk} \times 2^{-a_{jk}}$, where $b_{jk} \in \{-1, +1\}$ and a_{jk} is a non-negative integer. For example, $w_{v2} = \frac{1}{3}$ is approximated using three terms as $w'_{v2} = \frac{1}{2} - \frac{1}{8} - \frac{1}{32}$. As we explain later, each term x_{jk} is mapped to a suffix matching pattern. In what follows, we show how to compute the approximations and how to generate the rules.

Approximate the weights

We start with an initial approximation where the biggest weight is 1 and the other weights are 0. The initial approximation for $w_v = (\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$ is $w'_v = (0, 0, 1)$ (Figure 4.3(a)).

The errors, namely the difference between the w'_v and w_v, are $(-\frac{1}{6}, -\frac{1}{3}, \frac{1}{2})$. w_{v1}, w_{v2} are under-approximated , while w_{v3} is over-approximated.

We use error tolerance $e = 0.02$ for the example. The initial approximation is not good enough; w_{v2} is the most under-approximated weight with an error $-\frac{1}{3}$. To reduce its error, we add one power-of-two term to w'_{v2}. At the same time, this term must be subtracted from another over-approximated weight to keep the sum unchanged. We move a power-of-two term from w_{v3} to w_{v2}.

We decide the term based on the current errors of both weights. The term should offer the biggest reduction in errors. Let the power-of-two term be x. Given the current errors of w_{v2} and w_{v3}, *i.e.,* $-\frac{1}{3}$ and $\frac{1}{2}$, we calculate the new errors as $-\frac{1}{3}+x$ and $\frac{1}{2}-x$. Hence, the reduction is

$$\begin{aligned} \triangle &= |-\frac{1}{3}|+|\frac{1}{2}|-|-\frac{1}{3}+x|-|\frac{1}{2}-x| && (4.1) \\ &= 2\times(\min(\frac{1}{3},x)+\min(\frac{1}{2},x)-x) && (4.2) \end{aligned}$$

The function is plotted as red line in Figure 4.4. When $x = 1, \frac{1}{2}$ and $\frac{1}{4}$, the reduction is $-\frac{1}{3}, \frac{2}{3}$ and $\frac{1}{2}$ respectively. In fact, Equation $\triangle$ is a concave function, which reaches its maximum value when $x \in [\frac{1}{3}, \frac{1}{2}]$. Hence, we choose $\frac{1}{2}$. In a more general case, where multiple values give the maximum reduction, we break the tie by choosing the biggest term. After this operation, the new approximation becomes $(0, \frac{1}{2}, 1-\frac{1}{2})$ with errors $(-\frac{1}{6}, \frac{1}{6}, 0)$.

We repeat the same operations to reduce the biggest under-approximation and over-approximation errors iteratively. In the example, w_{v3} is perfectly approximated (the error is 0). We only move terms from w_{v2} to w_{v1}. Two terms $\frac{1}{8}, \frac{1}{32}$ are moved until all the errors are within tolerance. Eventually, each weight is approximated with an expansion of power-of-two terms (Figure 4.3(a)).

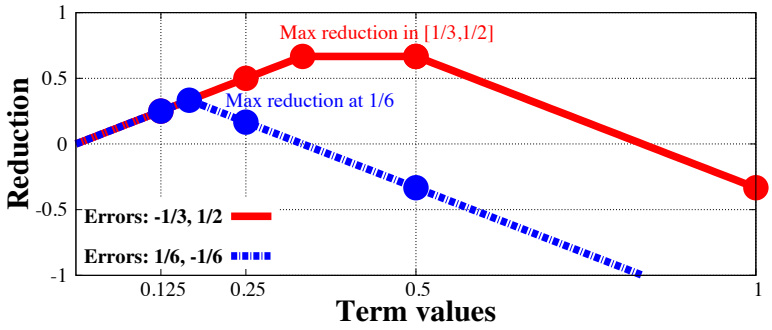


Figure 4.4: $\triangle$ plots with different errors.

We make three observations about this process. First, the errors are non-increasing, as each time we reduce the biggest errors. Second, the chosen power-of-two terms are non-increasing, because the terms with the maximum $\triangle$ always lie between two errors (Figure 4.4). For a term that gives the best $\triangle$ in the current iteration, only smaller terms may have a bigger reduction in the next iteration[1]. Finally, the reduction $\triangle$ is non-increasing, as Equation $\triangle$ is monotonic with both errors and the chosen power-of-two term. *In other words, we gain diminishing return on $\triangle$ for the term-moving operation, as we are getting closer to the error tolerance.*

Generate rules based on approximations

Given the approximation w'_v, we generate rules by mapping the power-of-two terms to nodes in a suffix tree. Each node in the tree represents a 2^{-k} fraction of traffic, where k is the node's depth (or, equivalently, the suffix length). Figure 4.5 visualizes the rule-generation steps for our example from Figure 4.3(a) with $w_{v1} = \frac{1}{6}$, $w_{v2} = \frac{1}{3}$, and $w_{v3} = \frac{1}{2}$. When a term is mapped to a node, we explicitly assign a color to the node. Initially, the root node is colored with the biggest weight to represent the initial approximation (Figure 4.5(a)). Color j means that the node belongs to w'_{vj}. Each uncolored node implicitly *inherits* the color of its closest ancestor. We use dark color for explicitly colored nodes and light color for the unassigned nodes.

[1] A term may be picked in multiple consecutive iterations.

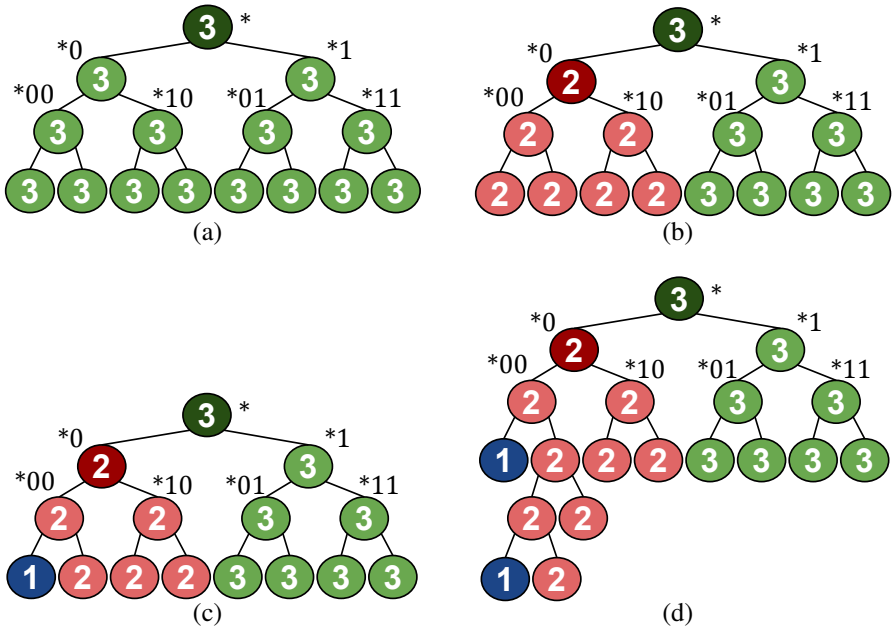


Figure 4.5: Generate rules using a suffix tree.

We process the terms in the order that they are added to the expansions (*i.e.,* $\frac{1}{2}$, $\frac{1}{8}$, $\frac{1}{32}$). Then, one by one, the terms are mapped to nodes as follows. Let x be the term under consideration, which is moved from weight w_{vb} to w_{va}. We map it to a node representing x fraction of traffic with color b. The node is then re-colored to a. In the example, we map $\frac{1}{2}$ to node $*0$ and color the node with w_{v2} (Figure 4.5(b)). Subsequently, $\frac{1}{8}, \frac{1}{32}$ are mapped to $*000, *00100$, which are colored to w_{v1} (Figure 4.5(c) (d)).

Once all terms have been processed, rules are generated based on the explicitly colored nodes. Figure 4.3(b) shows the rules corresponding to the final colored tree in Figure 4.5(d).

Use non-power-of-two terms

We discuss the case that each suffix pattern may not match a power-of-two fraction of traffic. For example, there may be more packets matching $*0$ than those matching $*1$. Niagara's algorithm can be extended to handle the unevenness, once the fractions of traffic for suffixes are measured [35, 57, 83, 89].

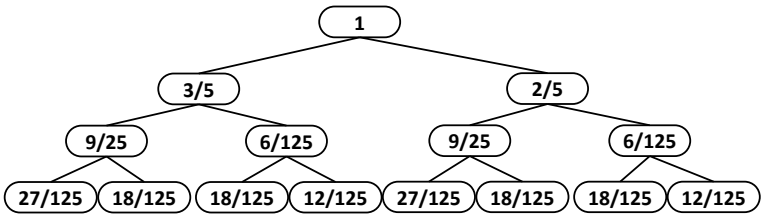


Figure 4.6: An example traffic distribution with a suffix tree. Each number represents the fraction of traffic matched by the suffix, *e.g.,* *11 matches $\frac{4}{25}$ traffic.

We refine the approximation iteratively. In each iteration, a suffix (*i.e.,* a term) is moved from an over-approximated weight to an under-approximated weight to maximize the reduction of errors. The only difference is that the candidate values of this term are no longer powers of two, but all possible fractions denoted by suffixes belonging to the over-approximated weight. We use the concaveness of Equation $\triangle$ to guide our search for the best term value. Instead of brute-force enumeration, we can scan all candidate values in decreasing order, and stop when $\triangle$ starts decreasing.

To illustrate the extended algorithm, we use $w_v = (\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$ as an example and assume an uneven traffic distribution over the flow space shown in Figure 4.6. We start with the approximation $w'_v = (0,0,1)$ (Figure 4.7(a)) and move a suffix from the over-approximated weight w_{v3} to the most under-approximated weight w_{v2} in the first iteration. Based on Equation $\triangle$, among all suffixes of w'_{v3}, *1 with term $= \frac{2}{5}$ maximizes $\triangle$ and is moved to w_{v2} (Figure 4.7(b)). The approximation becomes $(0, \frac{2}{5}, 1-\frac{2}{5})$. In the next iteration, we move suffix *100 with term $= \frac{18}{125}$ to w_{v1}, reducing the approximation error to $w'_v - w_v = (\frac{18}{125} - \frac{1}{6}, \frac{2}{5} - \frac{1}{3}, (1 - \frac{2}{5} - \frac{18}{125}) - \frac{1}{2})$. Finally, moving *111 with term $= \frac{8}{125}$ to w_{v2} completes the approximation. The resulting suffix tree is shown in Figure 4.7(d).

We also remark that it is not necessary to use suffix matches to approximate traffic volume. As long as the traffic distribution is measured for some bits in the header fields, we could apply the above algorithm to generate patterns matching those bits.

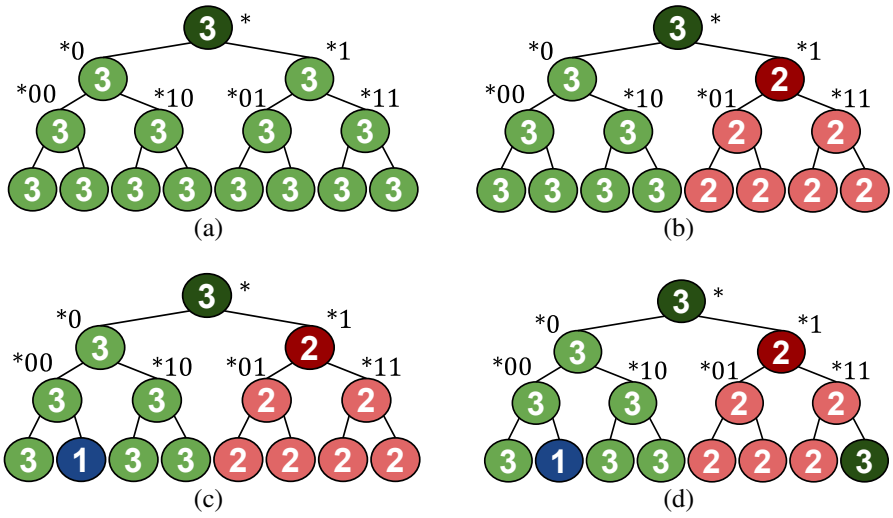


Figure 4.7: Generate rules using a suffix tree, given the traffic distribution in Figure 4.6.

4.4.2 Truncate: Fit Rules in the Table

Given the restricted rule-table size, some generated rules might not fit in the hardware. Therefore, we *truncate* rules to meet the capacity of rule table. We refer to the switch rules as P^H. P^H achieves a coarse-grained approximation of the weights while $numrules(P^H)$ stays within the rule-table size C. We capture the total over-approximation error as *imbalance*, *i.e.,*

$$t_v \times \sum_j E(w^H_{vj} - w_{vj}, e)$$

where t_v is the expected traffic volume for aggregate v and w^H_{vj} is the approximation of weight w_{vj} given by P^H.

We pick the C lower-priority rules from the rule-set generated in Section 4.4.1 as P^H. This is because rules are generated with increasing priority and decreasing $\triangle$ values (*i.e.,* the reduction in imbalance). The C lowest-priority rules give the overall biggest reduction of imbalance. For example, when $C = 3$ the rules in Figure 4.3(b) are truncated into P^H containing the last three rules.

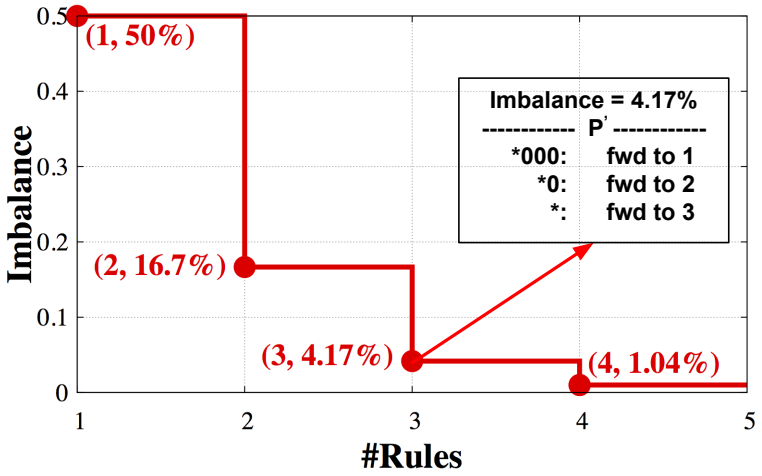


Figure 4.8: Stairstep curve (imbalance v.s. #rules) for Aggregate v with weights $w_v = \{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ and $t_v = 1$.

Stairstep plot. Figure 4.8 shows the imbalance as a function of C. Each point in the plot (r, imb) can be viewed as a *cost* for rule space r, and the corresponding *gain* in reducing imbalance imb. This curve helps us determine the gain an aggregate can have from a certain number of allocated switch rules, which is used in packing rules for multiple aggregates into the same switch table (Section 4.5.1).

4.5 Cross Aggregates Optimization

In this section, we generate rules for multiple aggregates using two main techniques: (1) *packing* multiple sets of rules (each corresponding to a single aggregate) into one rule table and (2) *sharing* the same set of rules among aggregates.

4.5.1 Pack: Divide Rules Across Aggregates

The stairstep plot in Figure 4.8 presents the tradeoff between the number of rules allocated to an aggregate and the resulting imbalance. When dividing rule-table space across multiple aggregates, we use their stairstep plots to determine which aggregates should have

more rules, to minimize the total traffic imbalance. Figure 4.9 shows the weight vectors, traffic volumes and stairsteps of two aggregates.

To allocate rules, we greedily sweep through the stairsteps of aggregates in steps. In each sweeping step, we give one more rule to the aggregate with *largest per-step gain* by stepping down one unit along its stairstep. The allocation repeats until the table is full.

We illustrate the steps through an example of packing two aggregates v_1 and v_2 using five rules (Figure 4.9). We begin with allocating each aggregate one rule, resulting in a total imbalance of 50% $(27.5\% + 22.5\%)$. Then, we decide how to allocate the remaining three rules. Note that v_1's per-step gain is 18.33% $(27.5\% - 9.17\%)$, which means that giving one more rule to v_1 would reduce its imbalance from 27.5% to 9.17%, while v_2's gain is 11.25% $(22.5\% - 11.25\%)$. We therefore give the third rule to v_1 and move one step down along its curve. The per-step gain of v_1 becomes 6.88% $(9.17\% - 2.29\%)$. Using the same approach, we give both the fourth and fifth rules to v_2, because its per-step gains $(22.5\% - 11.25\% = 11.25\%$ and $11.25\% - 0\% = 11.25\%)$ are greater than v_1's. Therefore, v_1 and v_2 are given two and three rules, respectively, and the total imbalance is 9.17% $(9.17\% + 0\%)$. The resulting rule-set is a combination of rules denoted by point $(2, 9.17\%)$ in v_1's stairstep and $(3, 0\%)$ in v_2's.

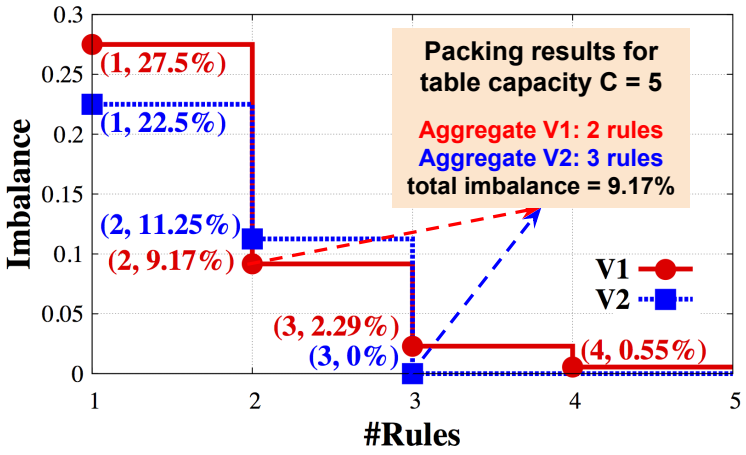
A natural consequence of our packing method is that aggregates with heavy traffic volume and easy-to-approximate weights are allocated more rules. Our evaluation demonstrates that this way of handling "heavy hitters" leads to significant gains.

4.5.2 Share: Same Rules for Aggregates

In practice, a switch may split thousands of aggregates. Given the small TCAM in today's hardware switches, we may not always be able to allocate even one rule to each aggregate. Thus, we are interested in *sharing* rules among multiple aggregates, which have the same set of next-hops. We employ sharing on different levels, creating three types of rules (with decreasing priority): (1) rules specific to a single aggregate (Section 4.4); (2)

Aggregate	Weights	Traffic Volume
v_1	$w_{11} = \frac{1}{6}, w_{12} = \frac{1}{3}, w_{13} = \frac{1}{2}$	$t_1 = 0.55$
v_2	$w_{21} = \frac{1}{4}, w_{22} = \frac{1}{4}, w_{23} = \frac{1}{2}$	$t_2 = 0.45$

(a) Weights and traffic volume of v_1 and v_2.



(b) Packing v_1 and v_2 based on stairsteps.

Figure 4.9: An example of packing multiple aggregates.

rules shared among a group of aggregates (Section 4.5.2), and (3) rules shared among *all* aggregates, called *default rules* (Section 4.5.2).[2]

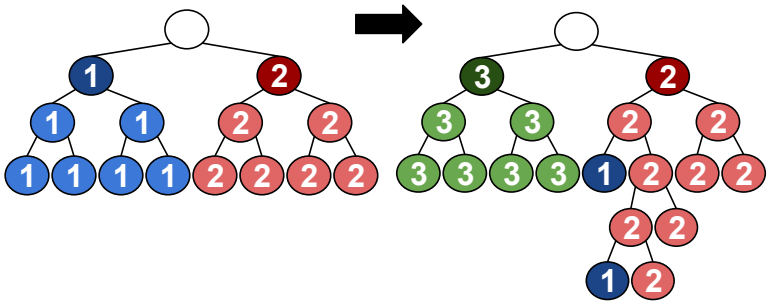
Default rules shared by all aggregates

Default rules have the lowest priority and are shared by all aggregates. There are many ways to create default rules, including approximating a certain weight vector using algorithm in Section 4.4. Here we focus on the simplest and most natural one—*uniform default rules* that divide the traffic equally among next-hops.

Assuming there are M next-hops where $2^k \leq M < 2^{k+1}$, we construct 2^k default rules matching suffix patterns of length k and distributing traffic evenly among the first 2^k next-hops. [3] These rules provide an initial approximation w^E of the target weight vector: $w^E_i = 2^{-k}$ for $i \leq 2^k$ and $w^E_i = 0$ otherwise, which can then be improved using more-specific per-

[2]Default rules do not require extra grouping table.

[3]When M is the power of two, the uniform default rules gives an equivalent split to ECMP.



(a) Initial (left) and final (right) suffix trees for $w'_{v1} = \frac{1}{2} - \frac{1}{2} + \frac{1}{8} + \frac{1}{32}$, $w'_{v2} = \frac{1}{2} - \frac{1}{8} - \frac{1}{32}$, $w'_{v3} = \frac{1}{2}$ (pool).

Rules	Pattern	Action
Rules for aggregate v	$*00101$	fwd to 1
	$*001$	fwd to 1
	$*0$	fwd to 3
Shared default rules	$*0$	fwd to 1
	$*1$	fwd to 2

(b) Rules that approximate v.

Figure 4.10: Generate rules for $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ given default rules

aggregate rules. If aggregates do not use the same set of next-hops, the default rules will only balance over the common set of next-hops and the per-aggregate rules will rebalance the loads of the rest of next-hops.

We revisit the example $(\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$. The initial approximation $w^E = (\frac{1}{2}, \frac{1}{2}, 0)$. $w_{v1} = \frac{1}{6}$ is over-approximated with error $\frac{1}{3}$; $w_{v3} = \frac{1}{2}$ is under-approximated with error $-\frac{1}{2}$; we move $\frac{1}{2}$ from w_{v1} to w_{v3}. The rest operations are similar to Section 4.4.1. Figure 4.10(a) shows the corresponding suffix tree. Initially, the tree is colored according to the uniform default rules. Next, we refine the approximation and obtain terms $\frac{1}{2}$, $\frac{1}{8}$, $\frac{1}{32}$ and the final rules (Figure 4.10(b)). The total number of rules is five, compared to four rules without using default rules (Figure 4.3(b)). However, only three of the five rules are "private" to aggregate v, as the two default rules are shared among all aggregates. This illustrates that default rules may not save space for one (or even several) aggregates, but will usually bring significant table space savings when the number of aggregates is large (Section 4.8).

Grouping aggregates with similar weights

To further save the table space, we group aggregates and tag aggregates in each group with the same identifier.

We use k-means clustering to group aggregates with similar weights. The centroid of each group is computed as the average weight vector of its member aggregates; to prioritize "heavy" aggregates, the average is weighted using t_v (the expected traffic volume of aggregate v). We begin by selecting the top-k aggregates with highest traffic volume as the initial centroid of the groups, where the choice of k depends on the available rule table space (Section 4.8). Then, we assign every aggregate to the group whose centroid vector is closest to the aggregate's target weight vector (using Euclidean distance). After assignment, we re-calculate group centroids. The procedure is repeated until the overall distance improvement is below a chosen threshold (*e.g.,* 0.01% in our evaluation).

Putting it all together. Niagara's full algorithm first (i) groups similar aggregates, then (ii) creates one set of default rules (*e.g.,* uniform rules) that serve as the initial approximation for all the groups, (iii) generates per-group stairstep curves, and finally (iv) packs groups into a rule table.

4.6 Graceful rule update

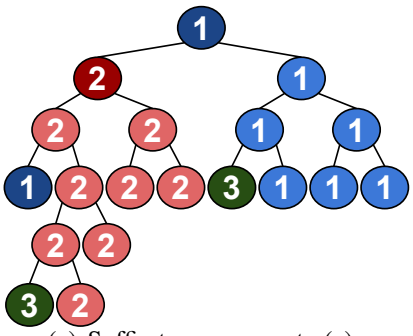
Weights change over time, due to next-hop failures, rolling out of new services, and maintenance. When the weights for an aggregate change, Niagara computes new rules while minimizing (i) churn due to the difference between old and new weights and (ii) traffic imbalance due to inaccuracies of approximation. Niagara has two update strategies, depending on the frequency of weight changes. When weights change frequently, Niagara *minimizes churn* by incrementally computing new rules from the old rules (Section 4.6.1). When weights change infrequently, Niagara minimizes *traffic imbalance* by computing the new set of rules from scratch and installs them in stages to limit churn (Section 4.6.2).

Pattern	Action
∗00100	fwd to 3
∗100	fwd to 3
∗000	fwd to 1
∗0	fwd to 2
∗	fwd to 1

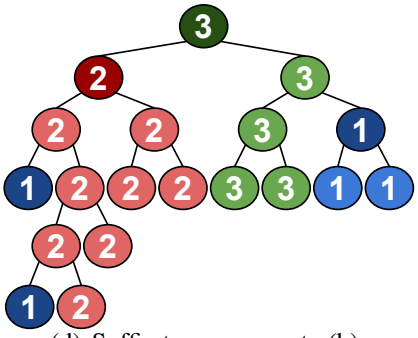
(a) Target rules.

Pattern	Action
∗00100	fwd to 1
∗000	fwd to 1
∗11	fwd to 1
∗0	fwd to 2
∗	fwd to 3

(b) Intermediate rules.



(c) Suffix tree corresp. to (a).



(d) Suffix tree corresp. to (b).

Figure 4.11: Rule-sets (and corresponding suffix trees) installed during the transition from $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ to $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$.

4.6.1 Incremental Rule Computation

When weights change, Niagara computes new rules to approximate the updated weights. New rules not only determine the new imbalance, but also the traffic churn during the transition. We use an example of changing weights from $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ to $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$ to illustrate the computation of new rules. Initial rules are given in Table 4.3(b) and the corresponding suffix tree in Figure 4.5(d). In this example, any solution must shuffle at least $\frac{1}{3}$ of the flow space (assuming a negligible error tolerance e), namely the minimal churn is $\frac{1}{3}$.

Minimize imbalance (recompute rules from scratch). A strawman approach to handle weight updates is to compute new rules from scratch. In our example, this means that action "fwd to 1" in Table 4.3(b) become "fwd to 3" and vice versa. This approach minimizes the traffic imbalance by making the best use of rule-table space. However, it incurs two drawbacks. First, it leads to heavy churn, since recoloring $\frac{1}{2} + \frac{1}{8} + \frac{1}{32}$ fraction of the suffix tree in Figure 4.5(d) means that nearly $\frac{2}{3}$ of traffic will be shuffled among next-hops.

Second, it requires significant updates to hardware, which slow down the update process. As a result, this approach does not work well when weights change frequently.

Minimize churn (keep rules unchanged). An alternative strawman is to keep the switch rules "as is". This approach minimizes churn but results in significant imbalance and overloads on next-hops. In the example, both the churn and the new imbalance are roughly $\frac{1}{3}$.

Strike a balance (incremental rule update). The above two approaches illustrate two extremes in computing the new rules. Niagara intelligently explores the tradeoff between churn and imbalance by iterating over the solution space, varying the number of old rules kept. In the example, keeping two old rules ($*000$ fwd to 1, and $*0$ fwd to 2) leads to the rule-set shown in Figure 4.11(a) and the suffix tree in Figure 4.11(c). The imbalance is $\frac{1}{32}$, the same with computation from scratch; the churn is $\frac{1}{32} + \frac{3}{8}$, which is slightly higher than the minimum churn $\frac{1}{3}$, as suffixes $*00100, *011, *11$ are re-colored to 1. In practice, when computing new rules for an aggregate, Niagara does not use more rules than the old ones.

4.6.2 Multi-stage Updates

Incurring churn during updates is inevitable. Depending on the deployment, this traffic churn might not be tolerable. Niagara is able to bound the churn by dividing the update process into multiple stages. Given a threshold on acceptable churn, Niagara finds a sequence of intermediate rule-sets such that the churn generated by transitioning from one stage to the next is always under the threshold.

Continuing the example in Section 4.6.1, we limit maximum acceptable churn to $\frac{1}{4}$. The churn for the direct transition from the old rules to the new rules is $\frac{1}{32} + \frac{3}{8}$, exceeding the threshold. Hence, we need to find an intermediate stage so that both the transition from the old rules to the intermediate rules and from the intermediate rules to the new rules do not exceed the threshold.

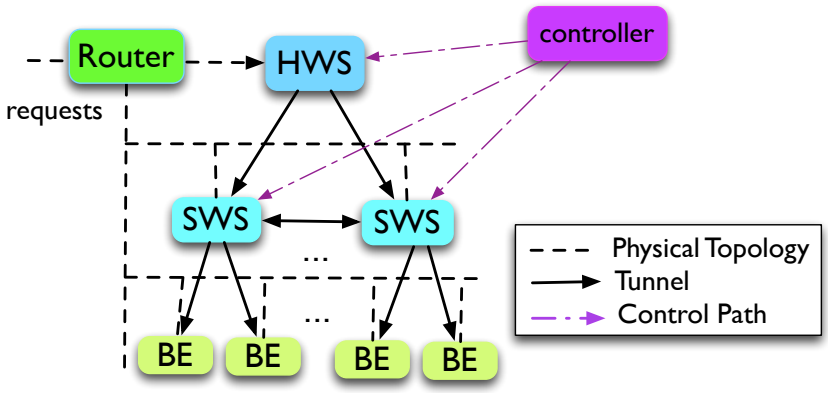


Figure 4.12: Niagara prototype architecture overview.

To compute the intermediate rules, we pick the pattern $*11$, which is the *maximal* fraction of the suffix tree that can be recolored within the churn threshold. The intermediate tree (Figure 4.11(d)) is obtained by replacing the subtree $*11$ of the old one (Figure 4.5(d)) with the new one's (Figure 4.11(c)). The intermediate rules are computed accordingly. Then, transitioning from the intermediate suffix-tree in Figure 4.11(d) to the one in Figure 4.11(c) recolors only $\frac{1}{32} + \frac{1}{8}$ ($< \frac{1}{4}$) of the flow space and therefore we can transition directly to the rules in Figure 4.11(a) after the intermediate stage.

We note that performing a multi-stage update naturally results in lengthy update process for aggregates with frequent weight changes. To mitigate this, Niagara may rate limit the update frequency of aggregates.

4.7 Niagara Application: Load Balancer

We prototype Niagara to show how to apply the output of our algorithm to an actual load balancer system comprising multiple switches (both stateless hardware and stateful software), as well as how to update switches consistently and ensure connection affinity.

System design. Figure 4.12 shows the network of switches and backends, with a router connecting to clients. The architecture is similar to Ananta [63]. All devices attach to a

shared L3 network and connect via GRE tunnels. We configure the router to direct all incoming requests to the hardware switch (HWS), which then forwards to a collection of software switches (SWSs). We program these SWSs to act as simple Software Load Balancers and distribute requests to backends (BEs). Return traffic is not tunnelled via SWSs but instead uses direct server return (DSR). We chose to implement both HWS and SWS atop regular Linux servers using *iptables* to reduce our implementation work at the expense of forwarding performance. Iptables can be configured remotely via `ssh` by the controller. Iptables allows the controller to create a collection of routing tables that match on arbitrary packet-header fields and set per-packet metadata. In addition, iptables can be configured to track L4 connections.

Software rules. SWSs have large forwarding tables optimized for exact-match rules, and fast rule updates, contrasting to the constrained rule table on the HWS. Hence, in addition to packing subsets of rules into the hardware rule table(Section 4.5.1), we could store the finer-grained unchosen rules that have higher priority in SWSs. We revisit the example in Figure 4.9. We pack five hardware rules: two rules for aggregate V1 and three rules for aggregate V2. Two higher-priority rules for V1 (Figure 4.3(b)) are not selected. We could store them in the SWSs. Namely, we install ($*$, fwd to 3) and ($*0$, fwd to 2) in the HWS, and install ($*000$, fwd to 1) and ($*00100$, fwd to 1) in the three SWSs. As a result, when the HWS splits aggregate V1 to SWSs, giving a rough approximation $(0, \frac{1}{2}, \frac{1}{2})$, SWSs could bounce back the "imbalanced" traffic to the correct SWSs. For example, a flow 00100 first is forwarded to 2nd SWS by the HWS, then matches to the software rules ($*00100$, fwd to 1) and finally arrives at the 1st SWS.

An important requirement in designing load balancers is to avoid breaking connections when changes occur (*e.g.,* backend addition and weight changes). The rule generation for hardware switches is stateless, *i.e.,* the update scheme for hardware switch (Section 4.6.1) does not protect existing connections. In our load balancer prototype, we design SWSs to

preserve ongoing connection during updates (Section 4.7.1). We present the implementation of the prototype in Section 4.7.2.

4.7.1 Preserve Connection Affinity

When performing updates, we must ensure that ongoing TCP connections remain pinned to the same backend ("connection affinity") regardless of where the new policy would send the flow. We could wait for old flows to terminate before applying a new policy [81] but this could delay updates indefinitely. The alternative, storing per-flow state in HWS, does not scale. Niagara chooses to track the connection-to-backend mapping at the software layer. Each time a new L4 connection is observed, an SWS maintains its routing decision in a table whose priority supersedes the load-balancing policy, thus pining connection mappings across changes in routing tables. HWS is freed from L4-related tracking tasks. All state tracking is done in abundant DRAM on SWSs.

Connection tracking. The idea of letting SWSs automatically generate a new micro-flow rule for each L4 routing choice follows the local-autonomy principle of DevoFlow [21]. Niagara's local micro-flows gain global significance whenever rules are updated as flows may bounce between switches. At those times, it is important to synchronize local microflows among all SWS. This could be done either via (i) eager periodic broadcast from the switches, (ii) controller-initiated poll-and-broadcast when there is a global policy update, or (iii) lazy schemes in which switches query upon receiving unexpected packets.

Policy versioning. Large sets of forwarding rules are tracked and applied atomically using versions. We tie each packet to the active policy via version tag in the packet. HWS always holds exactly one policy version and labels each routed packet accordingly. SWSs match their version to the routing label on the received packet.

A global policy update consists of the five steps as shown in Figure 4.13. We first install the new policy version (both hardware and software rules) on all SWSs (Step 1).

POLICY-UPDATE($version\_id, P^H, P^S$)
1 Install P^H and P^S on SWSs
2 SWSs apply new policy
3 Synchronize connection registry
4 Install P^H on HWS
5 Remove unmatched rules on SWSs

Figure 4.13: Global policy update scheme

These new rules remain shadowed until HWS stamps the new version number into forwarded packets; alternatively, we may instruct SWSs to re-stamp the new version individually (Step 2). Now, new connections are forwarded using the new version while existing connections remain routed as before. Note, all new flows are now being deflected to their target SWS by (another) SWS. Then, we synchronize the connection registries among all SWSs (Step 3) to ensure that any existing connection established under an old version is recognized and forwarded consistently by all SWSs. We then install new hardware rules at HWS (Step 4), so the "new connections" no longer need to be deflected by SWSs. However, connections from previous versions need to be deflected until they terminate. Finally, we garbage collect unmatched rules on SWSs.

In fact, the whole system applies the new policy to incoming packets after Step 2, irrespective of HWS's forwarding behavior. We could choose to never update HWS and things would continue to work. Updating HWS (Step 4) is important to reduce deflection. Not updating HWS keeps forwarding all packets of pre-existing connections to their "correct" SWSs per some previous policy version. This becomes less desirable as old connections die out and traffic churn begins to consist only of the new connections (established under the new policy); Once HWS is updated, only old connections need to be deflected by SWSs. We demonstrate this churn tradeoff between new and old flows when we evaluate the update dynamics of our prototype (§4.8).

4.7.2 Prototype

We discuss the implementation of the load balancer prototype in this section.

Packet processing. The controller begins by creating one routing table at each switch for the current policy version. Each version corresponds to one specific VLAN-tag. Upon receiving a packet, the switch translates the VLAN tag to a per-packet internal metadata *vmark*, and uses it to select the routing table. The rules inside the routing table, which are directly translated from the output of the algorithm, set additional metadata *rmark* denoting the next-hop for the packet. At the network ingress, HWS sets the first *vmark* (a.k.a. VLAN-tag) on any incoming packet.

Connection tracking. IPConntrack in iptables maintains a state table of active local flows, where we save the next-hop information (*rmark*) for the first packet of the connection. SWSs are configured to first check for each incoming packet if it belongs to an existing connection. If so, the packet is immediately forwarded according to the previously-saved *rmark*. Therefore, each flow is routed only once, when adding the flow to the state table; policy changes do not impact ongoing connections.

Since HWS update can reshuffle flow-to-SWS mappings, we need to synchronize connection mappings across SWSs. `Conntrackd` was built as an iptables add-on exactly for this purpose. In our prototype, we configured multicast state replication among SWSs. This multicast group effectively combines the local state tables into one logically shared global connection table, ensuring that packets of the same connection are forwarded to the same BE, even if they traverse different SWSs. To prevent conntrack state from blowing up, we must ensure fast garbage collection as connections expire. To this end, we set up route-exceptions at the BEs (also through iptables) to route all SYN-ACK and FIN packets through SWSs instead of sending them DSR. In practice there are a few more packets that need this exception treatment (*e.g.,* ICMP messages, RST, etc.).

Rule updates. We implement the update mechanism (§4.7.1) in our prototype. The update first creates the tables in SWSs that contain the complete rule-set of the new version.

When all SWSs are primed with the new version, we change the *vmark* at HWS. However, we do not to install the new rules at HWS immediately (§4.7.1). Instead, we proceed with a later HWS update to minimize traffic churn (§4.8).

Failures. The current prototype keeps an unbounded history of policy versions to avoid having to deal with wrap-around version numbers and out-of-sync SWSs.

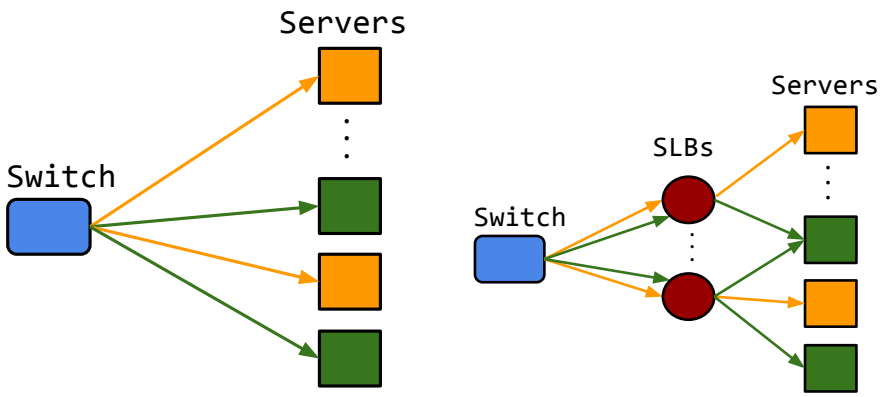
Practical observations. The HWS rule-set is completely stateless, matching only on L3 bits and can be mapped to the tables of a standard packet-forwarding chip like Broadcom's. The use of GRE tunnels is not always necessary (*e.g.,* L2 fabrics) and GRE causes trouble as it reduces MTU size, consumes CPU cycles, and often lacks NIC offload support. In L2 fabrics it may suffice to drive packets to the right SWS by forwarding the packet to the corresponding destination MAC address. Finally, we realize that multicasting the connection table is not going to scale. Instead we propose synchronizing each SWS against a few replicas of a sharded global connection table. Then on policy update, the global controller would initiate a push of connection-table entries from this sharded repository to SWSs, as fallback, SWSs would poll the sharded connection state table on receipt of unexpected packets.

4.8 Evaluation

This section presents the evaluation of Niagara in two scenarios: server load balancing and multi-path traffic splitting. We conduct both trace-driven analysis and synthetic experiments to demonstrate Niagara's splitting accuracy, scalability and update efficiency.

4.8.1 Niagara for Server Load Balancing

We evaluate Niagara's accuracy against real packet traces and load balancing configuration from a campus network. We further use large-scale synthetic data-center load



(a) University (b) Data center

Figure 4.14: Load balancer architecture.

balancing configuration to examine its scalability and update efficiency. Before diving into the results, we first describe the experiment setup and data for the two scenarios.

Setup. We use two different load balancer architectures for the campus network and the data center network (Figure 4.14). In the campus network, the switch directly forwards VIP requests to backend servers. VIPs are deployed on different servers, hence the switch cannot use default rules that are intended to be shared by all aggregates (*i.e.,* VIPs). In the data center network, the switch directs requests to an intermediate layer of Software Load Balancers (SLBs) [28, 63], which encapsulate packets to a pool of backend servers. In such a case, all VIP requests are distributed over the same set of SLBs, although the weights for each VIP can be different depending on the deployment of backend servers behind SLBs.

University traces and configuration. The campus network hosts around 50 services (*i.e.,* VIPs). Each VIP is served by 2 to 5 backends. VIP requests should be evenly distributed over backends. We collected a 20-minute Netflow traces from the campus border router and extracted the top 14 popular VIPs from the traces for our evaluation as the other VIPs saw only negligible traffic.

Synthetic weight distribution. In a large-scale data center network, the weights of a VIP depend on various factors such as capacity of next-hop servers and deployment plans. To reflect this variability, we use three different distribution models to choose VIP weights:

Gaussian, Bimodal Gaussian, and Pick Next-hop. Weights of a VIP v are drawn from these models and normalized such that $\sum_j w_{vj} = 1$.

Gaussian distribution. Weights are chosen from $N(4,1)$. Since the variance is small, the generated weights are close to uniform. This distribution models a setting where requests should be equally split over next-hops.

Bimodal Gaussian distribution. Here, each weight is chosen either from $N(4,1)$ or $N(16,1)$, with equal probability. The generated weights are non-uniform, but VIPs exhibit certain similarity. This distribution models a setting where some next-hops can handle more VIP requests than others.

Pick Next-hop distribution. In this model, we pick a subset of next-hops uniformly at random for each VIP. For the chosen next-hops, we draw the weights from the Bimodal Gaussian distribution and set the weights for the remaining unchosen next-hops to zero. The generated weights are non-uniform, making it hard for grouping. This case models a setting where different VIPs should be split over different subsets of next-hops.

Synthetic VIP traffic volume distribution. We use a Zipf traffic distribution where the k-th most popular VIP contributes $1/k$ fraction of the total traffic. The traffic volume is normalized so that $\sum_v t_v = 1$.

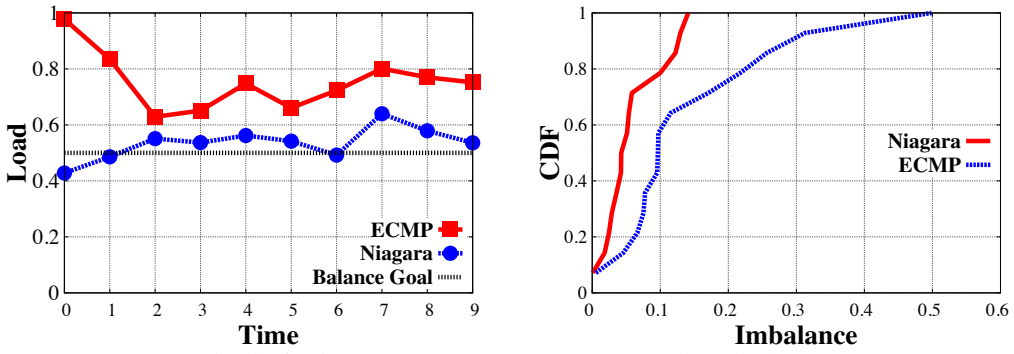
Metrics. We calculate *imbalance_lb* as

$$\sum_v (t_v \times \sum_j E(w'_{vj} - w_{vj}, 0))$$

where t_v is the traffic volume of VIP v, w_{vj} is the desired fraction of loads on next-hop j by VIP v and w'_{vj} is the actual load.A total imbalance $\leq 10\%$ is considered low.

Accuracy

We assume that the hardware switch directly forwards VIP requests to the backend servers (Figure 4.14(a)). The collected traffic traces exhibit stable traffic distribution over



(a) Server load (single VIP)

(b) Imbalance CDF

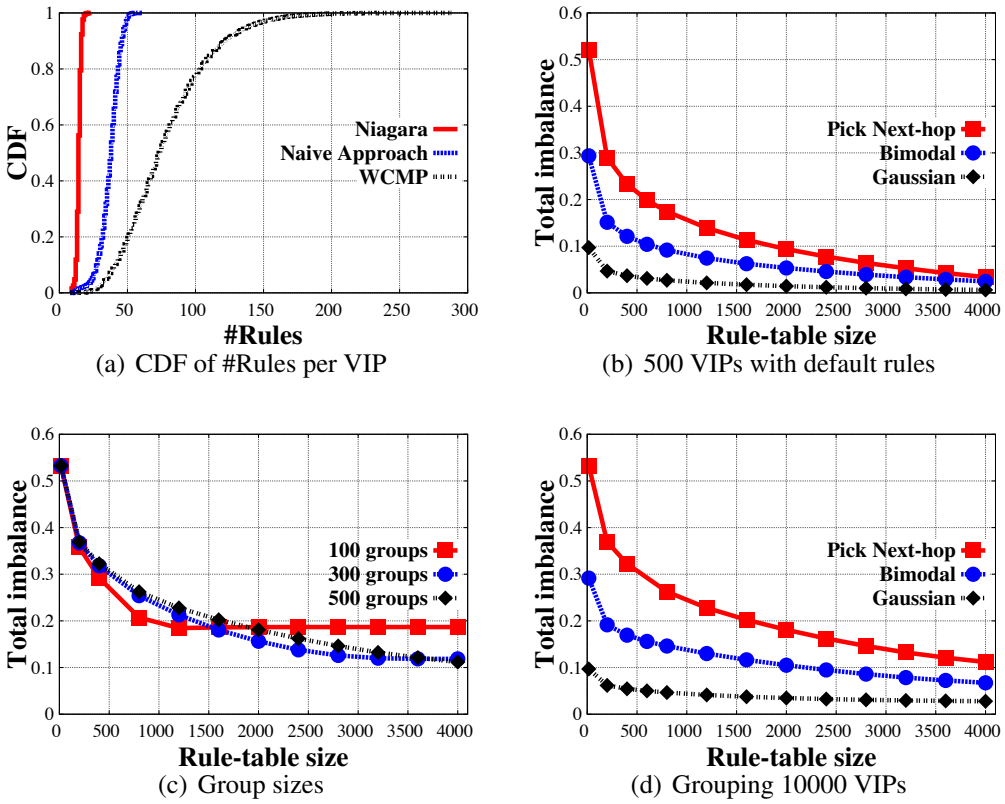
Figure 4.15: Accuracy of uniform server load balancing.

last 8 bits of source IP. In the experiment, *we run Niagara once with the profiled traffic distribution.*

We slice the 20-min trace into 2-min timeframes and compute the load of each backend using Niagara and ECMP. The ECMP hash function is SHA. We first examine one VIP with two backends each with 50% target load. Figure 4.15(a) shows the load of one of the backends. ECMP gives extremely unbalanced backend loads as part of the flow space contributes more traffic than the rest. On average, 80% of the load is absorbed by this backend and the total imbalance is $80\% - 50\% = 30\%$. In contrast, Niagara achieves a roughly balanced load with 1% imbalance. Figure 4.15(b) presents the CDF of imbalance for all VIPs. Even for uniform load balancing, ECMP still has a much longer imbalance tail than Niagara, because it merely splits the flow space equally regardless of the actual traffic distribution.

Rule Efficiency and Scalability

Next, we focus our attention to server load balancing in large-scale data center network setting (such as Duet [28] and Ananta [63]) with tens of thousands of VIPs, where hardware switches forward VIP requests to SLBs, which further distribute requests over backend servers (Figure 4.14(b)).



(a) CDF of #Rules per VIP
(b) 500 VIPs with default rules
(c) Group sizes
(d) Grouping 10000 VIPs

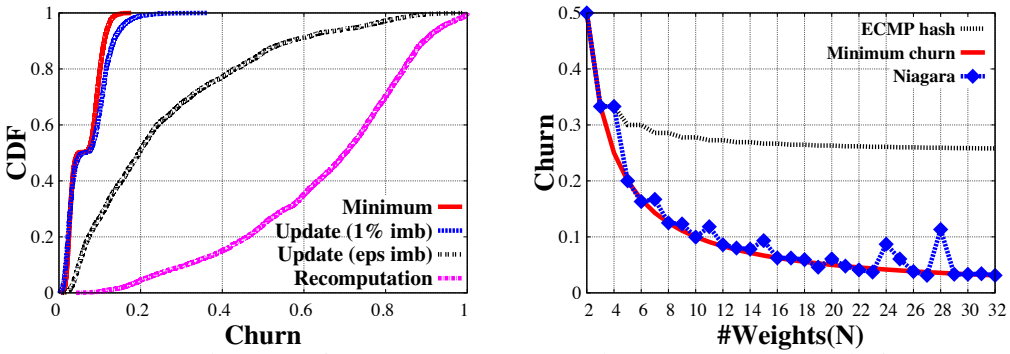
Figure 4.16: Weighted server load balancing for multiple VIPs.

Approximate weights for a single VIP. We examine the number of rules needed to approximate the target weights of a single VIP assuming a balanced distribution of traffic over flow space. We randomly generate 100000 distinct sets of 8 weights (*i.e.,* 8 SLBs) with error tolerance $e = 0.001$. Figure 4.16(a) compares the CDF of the performance of three strategies (Section 4.4.1): *WCMP*, which repeats next-hop entries in ECMP, *Naive approach*, which rounds weights to the nearest multiples of powers of two and Niagara, which uses expansions of power-of-two terms to approximate weights. WCMP performs the worst and needs as many as; 288 rules to reach the error tolerance. Its performance is very sensitive to the values of the target weights. A slight change of weights (*e.g.,* from 0.1 to 0.11) may cause a dramatic change in number of rules. In fact, we see similar results for less tight error tolerance as well. The naive approach performs slightly better with a median of 38 rules, but still uses more rules (61 in the worst case) compared to Niagara.

In comparison, Niagara generates the fewest rules (median is 14) with small variation. Niagara's performance is largely due to using both power-of-two terms and exploiting rule priorities to have both additive and subtractive terms.

Load balance multiple VIPs. Moving on to multiple VIPs, we use 16 weights per VIP (*i.e.,* 16 SLBs) and draw weights from the three synthetic models. We assume all VIPs share a set of uniform default rules. Figure 4.16(b) shows the total imbalance achieved by packing and sharing default rules for 500 VIPs, as a function of rule-table size. The leftmost point on each curve shows the imbalance given by the default rules (*i.e.,* ECMP). The initial imbalance for Gaussian, Bimodal and Pick Next-hop are $10\%, 30\%$ and 53% respectively. With Niagara, as the rule-table size increases, the imbalance drops nearly exponentially, reaching 3.3% at 4000 rules for Pick Next-hop model. This performance is due to the packing algorithm prioritizing "heavy-flows" when bumping up against rule-table capacity. Allocating rules to heavier-traffic sections of flow-space naturally minimizes imbalance given a fixed number of rules.

Our grouping technique (Section 4.5.2) groups VIPs with similar weight vectors. The maximal number of VIP groups affects approximation accuracy. When the VIPs are classified into more groups, the distance between each VIP's target weight vector and the centroid vector of its group is reduced, thus creating more groups containing only VIPs of more similar weights. However, as soon as rule capacity is reached, finer-grained VIP groups actually reduce overall performance because each group can push a small number of rules into the switch. Depending on number of groups, there is a tradeoff between grouping accuracy and approximation accuracy. When the VIPs are classified into more groups, the distance between each VIP's target weight vector and the centroid vector of its group is reduced, making the grouping more accurate. However, the approximation is less accurate for a bigger number of groups given limited rule capacity. Figure 4.16(c) illustrates this tradeoff by comparing the imbalance of classifying 10000 VIPs into 100, 300, and 500 groups. When there are less than 500 rules, classifying the VIPs into 100 groups performs



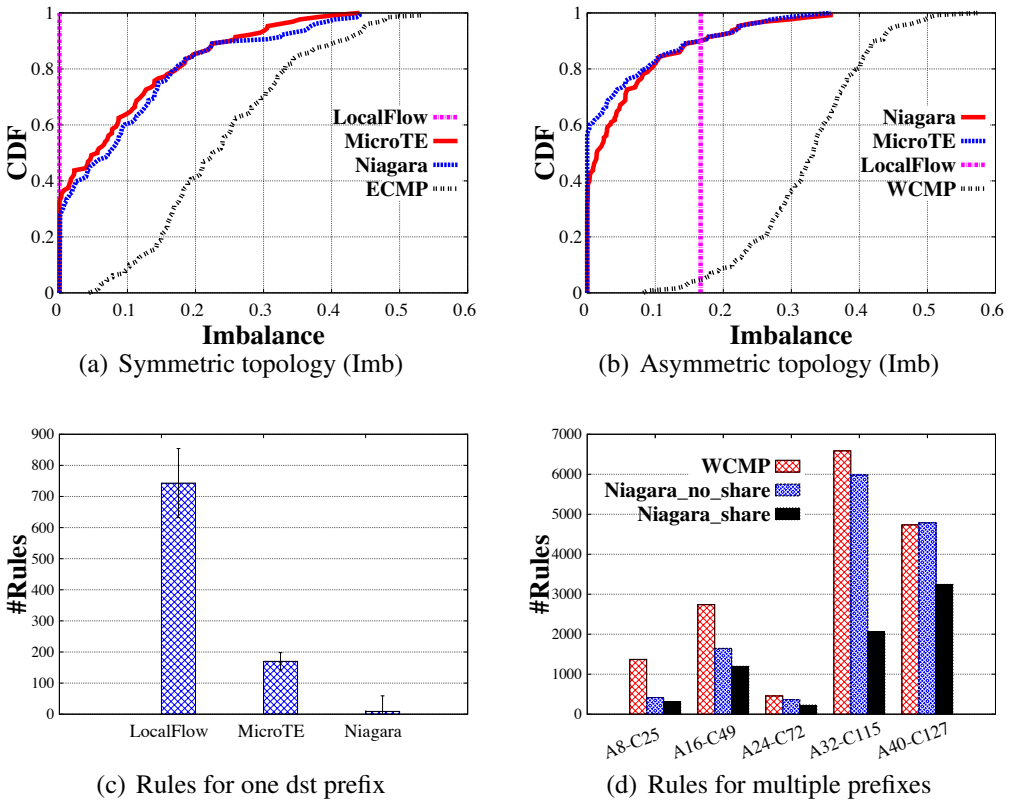
(a) Update strategies (b) Compare ECMP and Niagara

Figure 4.17: Incremental Update.

best, because it is easier to pack 100 groups and the centroids of groups still give a reasonable approximation for aggregates. As rule-table sizes increase, using more fine-grained VIP groupings is advantageous, since the distance between each aggregate and its group's centroid, which "represents" the aggregate during packing, decreases. For example, given 1500 rules, 300-group outperforms 100-group.

Figure 4.16(d) shows the effectiveness of grouping for different weight models. Given the number of rules, we classify the VIPs into 100, 300, or 500 groups (picking the option which yields the smallest imbalance). At 4000 rules, we reach 2.8% and 6.7% imbalance for the Gaussian and Bimodal Gaussian models respectively, and 11.1% imbalance for Pick Next-hop, which is much tougher to group. In contrast, ECMP incurs imbalance of $9.6\%, 29.1\%$ and 53.2% (the leftmost point), respectively.

Time. The algorithm performs well on a standard Ubuntu server (Intel Xeon E5620, 2.4 GHz, 4 core, 12MB cache). The prototype single-threaded C++ implementation completes the computation of the stairstep curves for a 16-weight vector ($e = 0.001$) in $10ms$. The time of packing grows linearly with the number of aggregates and is dominated by the computation of stairstep curve, which could be parallelized. The grouping function using k-means clustering takes at most 8 sec. to complete. If the traffic distribution is skewed and VIPs use similar weight distributions the algorithm tends to converge faster and requires fewer iterations. We do not expect to update aggregate groups frequently: if two aggregates



(a) Symmetric topology (Imb)

(b) Asymmetric topology (Imb)

(c) Rules for one dst prefix

(d) Rules for multiple prefixes

Figure 4.18: Multipathing

are grouped together, they must have similar deployment in the network and are unlikely to be changed dramatically in a short term.

Incremental Update

We evaluate the churn and imbalance caused by Niagara's incremental update strategy. Given the old weight vector, we randomly clear one non-zero weight and renormalize the rest to obtain new weights, or vice versa, simulating a server failure or addition. The minimum churn is the weight of the failed (or added) server.

Incremental update with low churn and imbalance. Our performance baseline is an approach where the load balancer recomputes all forwarding rules from scratch in response to a weight change. This baseline approach completely ignores churn and prior assignments by recalculating all rules. This strategy does minimize the number of rules, however

at the expense of incurring unnecessary traffic churn. In contrast, the incremental update algorithm in Niagara is aware of the cost of switching flows from one next-hop to another and tries to minimize churn. It keeps partial rules from the old rule-set and computes a small number of new rules to achieve the new weights while staying within bounded rule-space capacity. Figure 4.17(a) plots the CDF of the churn among 5000 weight vectors drawn from Bimodal distribution. The full recomputation approach (pink curve) incurs about 70% churn in 50% of test cases while Niagara's incremental update approach (black curve) only incurs 20% churn for half of test cases. This suggests that Niagara's intuition that an old rule-set serves as a good approximation for updated weights holds up in practice. Furthermore, this observation holds across the weight models used in this study.

Although Niagara's strategy explained above already reduces churn, it can be further improved by allowing a small margin for imbalance. The above strategy ignores larger rules-sets (than the minimum) that gives less churn. Based on this observation, we evaluate an alternative update strategy which installs truncated rules of larger rule-sets with up to 1% imbalance. The resulting curve (blue line in Figure 4.17(a)) almost overlaps with the curve of minimum churn (red). This confirms that an allowance for small imbalance will greatly reduce churn during updates.

Comparison with hash-based approaches. The theoretical lower bound of churn for ECMP, *i.e.,* assuming a perfect balanced traffic distribution over the flow space, is $\frac{1}{4} + \frac{1}{4N}$ for removing one member from a N-sized group (or adding one member to $(N-1)$-sized group) contrasting to the minimum churn of $\frac{1}{N}$ [37, 77]. We compare the churn of ECMP and Niagara using a uniform weight distribution, *e.g., N* weights of $\frac{1}{N}$. We create random server failure and additions as described in the previous experiment. For each value of N, Niagara generates the rule-set with minimum churn, while (1) staying within the number of rules needed by recomputation and (2) incurring less than 1% imbalance. Figure 4.17(b) presents the comparison of Niagara and ECMP. Niagara's performance (the blue line with diamonds) closely follows the curve of minimum churn; the fluctuation in

performance (*e.g.,* $N = 24, 28$) is due to the differences in approximating $\frac{1}{N}$. Niagara gives a much smaller churn than ECMP for $N \geq 5$. When $N = 32$, Niagara reduces the churn by 87.5% compared to ECMP.

Time. Given a rule-set of 30 rules, if we enumerate the number of lower-priority rules kept in the new rule-set, the incremental computation takes about $30 \times 10\text{ms} = 300\text{ms}$ to complete, which is in the same order of magnitude as rule insertion and modification on switches (3.3ms to 18ms [42, 48]). This is sufficient for updates on the timescale of management tasks. For planned updates, we can also pre-compute the new rule-set in advance.

Connection Affinity

We evaluate our rule update mechanism for connection affinity in our prototype. The setup includes one HWS, two SWSs (SW1 and SW2), and two BEs (BE1 and BE2) serving a single VIP v. We connect BE1 to SW1 and BE2 to SW2. Thus, BE1 is the only backend of SW1 and similarly for SW2. Each SWS sends all requests to its only backend unless the packets should be deflected. We inject client traffic destined to VIP v into the network, and monitor the bytes received at BEs as well as packet deflection.

In the experiment, we transition from weights $\{w_{v1} = \frac{3}{4}, w_{v2} = \frac{1}{4}\}$ to weights $\{w_{v1} = \frac{1}{4}, w_{v2} = \frac{3}{4}\}$. Both the old policy and the new policy achieve weights using hardware rules. The old policy map $*00$ to BE2 and the rest to BE1; the new policy. change the mapping of $*01$ and $*10$ to BE2. During the update, the existing connections of $*01$ and $*10$ should be pinned to BE1, but new connections should be directed to BE2. We start eight TCP connections to VIP v matching patterns $000, 001, \ldots, 111$, where connections end asynchronously and new connections of the same pattern start afterwards. Then, we update switches and keep recording the packets received by BEs and traffic churn during the update.

Figure 4.19 shows three runs of the experiment, where the only difference is the timing of HWS update:

Update HWS and SWSs together (top): At the beginning, the eight TCP connections create 3 : 1 throughput ratio at BE1 and BE2. No packets are deflected. At 90 sec. we update HWS and both SWSs. As a consequence, for active flows 001, 010, 101, and 110, HWS sends their packets to SW2 and SW2 directs them to BE1. Therefore, although the throughputs of BEs do not change, we see a sudden increase in traffic churn consisting of old flows. This traffic churn gradually disappears, as these flows finish. Finally, the throughput ratio at BE1 and BE2 becomes 1 : 3.

Update HWS after all old flows end (center): If we update HWS *after* all old flows end, we see no traffic churn immediately after updating SWSs (at 90 sec.), since packets from old flows still hit their original SWSs. However, churn increases as new flows arrive. For example, when a new flow 001 starts, HWS sends its packets to SW1 based on the old rules; SW1 applies the new rules, and redirects packets to BE2. Eventually, HWS is updated after old flows end (160 sec.), stopping the deflection of new flows.

Update HWS at an optimized time (bottom): Since new flows keeps expanding and old flows are shrinking, we can find a "sweet-spot" that minimizes the traffic churn. In the example, we update HWS at 125 sec. The churn contains only new flows before the update and old flows afterwards.

4.8.2 Niagara for Multi-pathing

This section presents Niagara's performance for splitting traffic over multiple equivalent outgoing links by simulating real data center traces [12] on both symmetric and asymmetric topologies [88].

Metrics. We calculate *imbalance_mp* as

$$\sum_i \max(0, F_i - \frac{W_i}{\sum_k W_k})$$

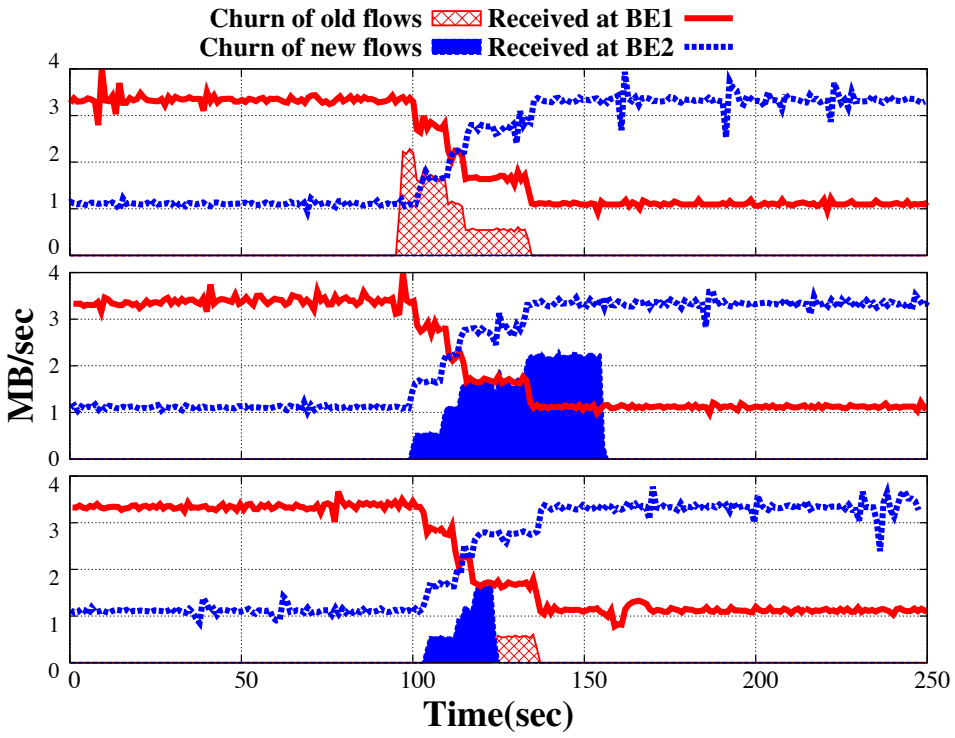


Figure 4.19: Top: update of SWSs and HWS together; Center: update HWS after old flows finish; Bottom: update HWS at an optimized time.

where F_i is the fraction of traffic sent on i-th link and W_i is the weight of i-th link (*i.e.,* the relative bandwidth capacity). It characterizes the total oversubscription when the switch operates at its full bandwidth capacity.

Accuracy in symmetric topology. We simulate 1-hour real packet traces [12] to a popular /16 prefix on a single switch with 4 equal-capacity outgoing links. We slice the trace into 30-second time frames and calculate the imbalance within each time frame.

We compare the splitting performance of Niagara, ECMP, MicroTE [13] and LocalFlow [70]. As MicroTE schedules forwarding paths for ToR-to-ToR flows, we assume that each /24 prefix in the traces correspond to a ToR and compute the utilization and imbalance accordingly. Figure 4.18(a) shows the CDFs. ECMP performs much worse than Niagara, as it only splits the flow space equally without taking into account the actual flow sizes. ECMP gives $< 10\%$ imbalance in around 10% of the time frames. In comparison,

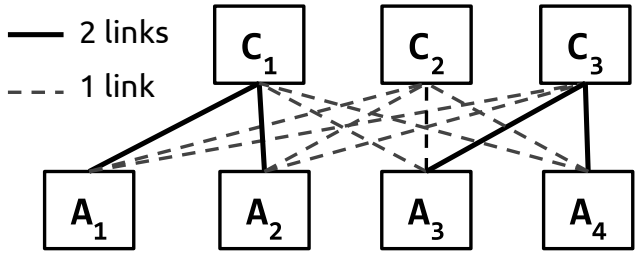


Figure 4.20: Topology: $N_C = 3, N_A = 4, L_C = 6, L_A = 4$

Niagara achieves $< 10\%$ imbalance in 61% of the time frames. MicroTE and Niagara offer similar splitting performance. We notice that Niagara incurs high imbalance for some of the time frames (*e.g.,* 15% time frames have $> 20\%$ imbalance). Upon close examination of the traces, we found that these time frames contain large "elephant" flows; Niagara could not achieve balanced split as it does not split a single flow over multiple links to avoid packet reordering. This also explains why LocalFlow, which splits flows, performs the best.

Accuracy in asymmetric topology. We experiment with a simple asymmetric topology in Figure 4.20, where there are three core switches and four aggregation switches with 4 links each. We look at the traffic splitting at A_1. A_1 can split traffic destined to A_2 evenly on the 4 uplinks, as A_1 and A_2 have the same bandwidth capacity to all core switches. For traffic to A_3, although A_1 has two links connected to C_1, it cannot send more traffic to C_1 than C_2 or C_3, because C_1 only has one link to A_3. Therefore, A_1 should split traffic destined to A_3 in proportion to $\frac{1}{2} : \frac{1}{2} : 1 : 1$ (*i.e.,* $w = (\frac{1}{6}, \frac{1}{6}, \frac{1}{3}, \frac{1}{3})$) over the 4 uplinks.

Figure 4.18(b) shows the imbalance CDF for splitting traffic for A_3 at A_1. It is no surprise that Niagara gives a much better result than WCMP. Niagara offers similar performance to MicroTE. For smaller imbalance ($< 2\%$), Niagara performs slightly worse than MicroTE, because it schedules bulks of flows (matching wildcard patterns) rather than ToR-to-ToR flows. This allows Niagara to use much fewer rules than MicroTE. Both Niagara and MicroTE offer $< 10\%$ imbalance for 82% of timeframes. LocalFlow's imbalance is steady at 16.6%, as it always splits traffic evenly.

Rule efficiency. We compare the number of rules generated by Niagara, MicroTE and LocalFlow to split the flows of a single destination prefix evenly (Figure 4.18(c)). LocalFlow uses the most rules: 743 on average and 854 in the worst case, because it needs finer-grained rules, which even match on bits outside 5-tuple for splitting a single flow, to balance link loads. MicroTE uses fewer rules (149 rules on average and 198 in the worst case) but still significantly more than Niagara, because it schedules ToR-to-ToR traffic. Niagara uses an average of 9 rules (59 in the worst case), which is 1.2% of the rule consumption of LocalFlow and 6% of MicroTE. In fact, *the rule consumption of MicroTE and LocalFlow heavily depends on the traffic pattern (*e.g., *active flows and active ToR pairs), making them hard to scale and less accurate when splitting multiple destination prefixes is needed.* Consider a rule-table with 4000 rules, LocalFlow and MicroTE can at most handle 5 and 26 flow aggregates given similar traffic patterns. In contrast, Niagara can handle more than 400 aggregates.

To compare the number of rules needed to balance multiple flow aggregates between Niagara and WCMP we generate large, asymmetric topologies to examine the total number of rules installed at an aggregation switch. A typical asymmetric topology contains two layers of switches: N_C core switches and N_A aggregation switches. Each core switch has at most L_C links to the aggregation layer; each aggregation switch has at most L_A links to the core layer. The connection algorithm in [88] is used to interconnect two layers of switches. The result is an asymmetric topology that maximizes bisection bandwidth among aggregation switches. We set $L_C = 64$ and $L_A = 192$ and vary the values of $N_C \in [1, L_A]$ and $N_A = 8, 16, 24, 32$. Figure 4.18(d) compares the number of rules generated by (1) WCMP, (2) Niagara_no_share, where there is no shared default rules and (3) Niagara_shared, where uniform default rules are used. We found that Niagara_share always outperforms WCMP. This figure also shows the rule-saving benefits of shared default rules.

4.9 Conclusion

Niagara advances the state-of-the-art in traffic splitting on switches by demonstrating a new approach that takes a resourceful approach to install carefully optimized flow-rules into hardware switches to closely approximate the desired load distribution and minimize traffic churn during weight changes given the limited rule table capacity.

Chapter 5

Conclusion

Today, managing enterprise networks is complicated, as a result of lacking high-level management abstractions and open configuration to devices. Leveraging the evolving technologies—SDN, we present a novel management system for enterprise networks. The new management abstractions shield operators from low-level details about hosts and switches and enable flexible network policies using commodity devices. In this chapter, we summarize our research contributions, discuss the issues on the deployment of our system and remark on the future work.

5.1 Summary of Contributions

We revisit our design principles and summarize our contributions.

New abstractions. We propose three abstractions for flexible network management: One Big Switch, Attribute-Carrying IP Addresses and One Big Server. All these abstractions relieve the operators from manual low-level management tasks and provide them intuitive yet powerful control on how to handle network traffic. One Big Switch decouples specification of the endpoint policies and the routing policies, shielding operators from reasoning about per-device configuration and dependencies among policies. With Attribute-Carrying IP Addresses, operators can efficiently group hosts with the same attribute infor-

mation and specify flexible network policies for host groups. Finally, One Big Server offers operators precise control on server loads using commodity devices.

Efficient algorithms. We design algorithms to automate the realization of these abstractions. The core challenges of realizing the abstractions are the limited rule-table sizes of switches. The algorithms must deal with the constraints of multiple switches simultaneously and handle frequent updates to policies gracefully. For One Big Switch, our rule placement algorithms intelligently partition endpoint policies to fit into individual switches while respecting the routing policy. For ACIPs, our Alpaca algorithms compute address assignment to efficiently encode host attributes using limited address space and minimize the rules to represent attributes. For One Big Server, our Niagara algorithms generate traffic splitting rules given the target load on servers, the traffic distribution and the available rule space on switches.

Realistic evaluations and prototype. We evaluate the performance of our algorithms using real and synthetic data. Our evaluation demonstrates that the rule placement algorithm can effectively realize real campus access control policies on "One Big Switch" consisting of hundreds of physical switches. Alpaca not only substantially reduces the existing switch configuration rules by up to 68%, but also shows potential reduction by an order of magnitude for futuristic network policies compared to the state of the art. Finally, Niagara scales to tens of thousands of services using a few thousand switch rules.

5.2 Deployment of the Management System

In this section, we discuss benefits of combining Alpaca, Niagara and One Big Switch in a network.

5.2.1 Deploy Niagara and One Big Switch

Niagara generates traffic-splitting rules for a single switch given the load balancing weights of VIPs over backend servers. Our "One Big Switch" algorithms place the rules of the endpoint policy according to the routing policy. Combining both works, we are able to build a powerful virtual hardware load balancer with a very big rule-table, out from a distributed collection of switches with small rule-tables.

We abstract the set of commodity switches, which connect external networks (*e.g.,* Internet) and backend servers, as a single "One Big Switch". On one hand, the routing policy lays down paths between all pairs of endpoints through the commodity switches, taking care of network bandwidth and latency. On the other hand, the high-level load balancing goal, *i.e.,* the splitting weights of VIP requests, is the endpoint policy, defining how packets coming from one endpoint (*e.g.,* the external network) should be modified (*e.g.,* rewritten to a DIP) and distributed across other endpoints (*e.g.,* the backend servers). Niagara algorithms can generate rules for this endpoint policy, which is divided across the underlying commodity switches by running the rule placement algorithm.

The rule placement algorithm guarantees that the resulting "One Big Switch" has a much larger rule capacity than any of the hardware switches. Meanwhile, with the increasing rule capacity, Niagara is able to balance service requests more accurately over the backend servers. These two works together present a design of an elastic load balancer architecture using commodity components. The design offers operators flexible choices from handling a small number of services to hosting a large number of services with hundreds of switches.

5.2.2 Deploy Alpaca and One Big Switch

Alpaca manages IP address assignment to hosts. The address assignment is computed such that address of hosts with the same attribute can be aggregated into a small number

of address patterns (*e.g.,* wildcard matches). These patterns are used to "compile" a policy defined on host attributes to a set of "match-and-action" switch rules.

While Alpaca can be deployed alone, instructing DHCP server and working with drivers to install rules for individual switches (Figure 3.1), it can collaborate with the "One Big Switch" to provide more powerful network control. Given a high-level endpoint policy (*e.g.,* access control) defined on host attributes, Alpaca can compute an address assignment so as to minimize the resulting switch rules to realize the policy. These switch rules define the end-to-end actions applied to packets, and can be distributed to the underlying physical switches using the rule placement algorithms.

Initially, Alpaca is designed to minimize the number of patterns to represent host attributes, regardless of how these attributes are used by different policies. When individual switches have their own policies (*i.e.,* per-switch configuration), each of which may be defined on different sets of host attributes, it is hard for Alpaca to generate the minimum number of patterns for all attributes or to generate patterns such that each resulting rule-set fits into the corresponding rule table. With the One Big Switch abstraction, the optimization goal of Alpaca is simplified. The endpoint policy is global. Alpaca only needs to optimize the address assignment with regard to the single network-wide policy. The rule placement algorithm will take care of distributing the optimized resulting endpoint rules to individual switches.

5.3 Concluding Remarks

Today, most SDN research works focuses on data-center networks and private backbone networks, where the network owners also control the endpoints and innovate the management systems. Unlike these companies which are specialized in managing their own networks, most enterprises do not innovate (or develop) their own network management systems and therefore very limited to the prior complicated management approaches. Fur-

thermore, enterprise networks connect many diverse hosts that are not fully controlled by the enterprise (*e.g.,* personal laptops, mobile phones and printers). Hence, operators have more desire to enforce diverse and flexible policies.

This thesis presents a novel management system for enterprise networks. It leverages the architecture of Software Defined Networks and revisits how operators should manage an enterprise network. We present a series of management abstractions for operators to define diverse network policies, such as access control and load balancing. Our system automates the enforcement of the network-wide policies on commodity hardware switches.

We notice that our system only deals with a limited types of network policies such as access control, routing and load balancing. A complete system for managing enterprise networks should also support (i) traffic monitoring which collects live packets to analyze traffic patterns, record usage and detect anomaly, (ii) advanced QoS which provides better performance to critical traffic and rate limits hosts' traffic to avoid congestion, and (iii) debugging tools which help operators to quickly localize faulty devices and troubleshoots the performance issues. A promising future research direction is to build a framework that incorporates these functionality for the enterprise network management.

Bibliography

[1] GLIF 2014 demos. http://www.glif.is/meetings/2014/demos.

[2] POX. http://www.noxrepo.org/pox/about-pox/.

[3] Production quality, multilayer open virtual switch. http://openvswitch.org/.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

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

[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 *SIGCOMM*, 2014.

[7] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *ANCS*, 2012.

[8] Michiel Appelman and Maikel DE Boer. Performance analysis of OpenFlow hardware. Technical report, University of Amsterdam, 2012. http://www.delaat.net/rp/2011-2012/p18/report.pdf.

[9] David L. Applegate, Gruia Calinescu, David S. Johnson, Howard Karloff, Katrina Ligett, and Jia Wang. Compressing rectilinear pictures and minimizing access control lists. In *ACM-SIAM SODA*, 2007.

[10] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[11] Theophilus Benson, Aditya Akella, and David A Maltz. Mining policies from enterprise network configuration. In *IMC*, 2009.

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

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

[14] 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*, 2014.

[15] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Ferndando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, 2013.

[16] Broadcom. High capacity StrataXGS Trident II Ethernet switch booktitle. `http://www.broadcom.com/products/Switching/Data-Center/BCM56850-booktitle`.

[17] Kenneth Calvert, Matthew B. Doar, Ascom Nexion, and Ellen W. Zegura. Modeling Internet topology. *IEEE Communications Magazine*, 1997.

[18] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-packet load-balanced, low-latency routing for Clos-based data center networks. In *CoNEXT*, 2013.

[19] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *ToN*, 2009.

[20] Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *PRESTO*, 2010.

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

[22] Sean Donovan and Nick Feamster. NetAssay: Providing new monitoring primitives for network operators. In *ACM HotNets*, 2014.

[23] Richard Draves, Christopher King, Srinivasan Venkatachary, and Brian Zill. Constructing optimal IP routing tables. In *INFOCOM*, 1999.

[24] David Erickson. The Beacon OpenFlow controller. In *HotSDN*, 2013.

[25] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *NSDI*, 2014.

[26] FlowScale. `http://www.openflowhub.org/display/FlowScale`.

[27] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ICFP*, 2011.

[28] Rohan Gandhi, Hongqiang Liu, Yu Hu, Guohan Lu, Jitu Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM*, 2014.

[29] Aaron Gember, Aditya Akella, Ashok Anand, Theophilus Benson, and Robert Grandl. Stratos: Virtual Middleboxes as First-Class Entities. Technical Report TR1771, University of Wisconsin-Madison, 2012.

[30] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, 2009.

[31] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 2008.

[32] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *SIGCOMM*, 1999.

[33] Nikhil Handigol, Mario Flajslik, Srini Seetharaman, Ramesh Johari, and Nick McKeown. Aster*x: Load-balancing as a network primitive. In *ACLD*, 2010.

[34] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick Mckeown. Reproducible network experiments using container based emulation. In *CoNEXT*, 2012.

[35] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.

[36] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.

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

[38] Danny Yuxing Huang, Kenneth Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *HotSDN*, 2013.

[39] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *CCS*, 2000.

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

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

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

[43] Abdul Kabbani, Balajee Vamanan, Jahangir Hasan, and Fabien Duchene. FlowBender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *CoNEXT*, 2014.

[44] Srikanth Kandula, Dina Katabi, Shan Sinha, and Arthur W. Berger. Flare: Responsive Load Balancing Without Packet Reordering. In *SIGCOMM CCR*, 2007.

[45] Yossi Kanizo, David Hay, and Isaac Keslassy. Palette: Distributing tables in Software-Defined Networks. In *IEEE INFOCOMM Mini-conference*, 2013.

[46] Naga Katta, Jennifer Rexford, and David Walker. Infinite CacheFlow in Software-Defined Networks. Technical Report TR-966-13, Princeton University, 2013.

[47] Kirill Kogan, Sergey I. Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. Exploiting order independence for scalable and expressive packet classification. *ToN*, 2015.

[48] Aggelos Lazaris, Daniel Tahara, Xin Huang, Erran Li, Andreas Voellmy, Y Richard Yang, and Minlan Yu. Tango: Simplifying SDN Control with Automatic Switch Property Inference, Abstraction, and Optimization. In *CoNEXT*, 2014.

[49] Yadi Ma and Suman Banerjee. A smart pre-classifier to reduce power consumption of TCAMs for multi-dimensional packet classification. In *SIGCOMM*, 2012.

[50] Rick McGeer and Praveen Yalagandula. Minimizing rulesets for TCAM implementation. In *INFOCOM*, 2009.

[51] 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*, 2008.

[52] Chad R. Meiners, Alex X. Liu, and Eric Torng. Topological transformation approaches to optimizing TCAM-based packet classification systems. In *SIGMETRICS*, 2009.

[53] Chad R. Meiners, Alex X. Liu, and Eric Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *ToN*, 2010.

[54] Chad R. Meiners, Alex X. Liu, and Eric Torng. BitWeaving: A non-prefix approach to compressing packet classifiers in TCAMs. *ToN*, 2012.

[55] Chad R. Meiners, Alex X. Liu, Eric Torng, and Jignesh Patel. Split: Optimizing space, power, and throughput for TCAM-based classification. In *ANCS*, 2011.

[56] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software Defined Networks. In *NSDI*, 2013.

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

[58] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. vCRIB: Virtualized rule management in the cloud. In *NSDI*, 2013.

[59] Erik Nordström, David Shue, Prem Gopalan, Rob Kiefer, Matvey Arye, Steven Ko, Jennifer Rexford, and Michael J. Freedman. Serval: An end-host stack for service-centric networking. In *NSDI*, 2012.

[60] O. Rottenstreich and I. Keslassy. On the code length of TCAM coding schemes. In *IEEE ISIT*, 2010.

[61] Recep Ozdag. Intel®Ethernet Switch FM6000 Series-Software Defined Networking. *Intel Corporation*, 2012.

[62] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[63] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

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

[65] Ori Rottenstreich, Amit Berman, Yuval Cassuto, and Isaac Keslassy. Compression for fixed-width memories. In *IEEE ISIT*, 2013.

[66] Ori Rottenstreich, Isaac Keslassy, Avinatan Hassidim, Haim Kaplan, and Ely Porat. Optimal In/Out TCAM encodings of ranges. *ToN*, 2015.

[67] Ori Rottenstreich, Marat Radan, Yuval Cassuto, Isaac Keslassy, Carmi Arad, Tal Mizrahi, Yoram Revah, and Avinatan Hassidim. Compressing forwarding tables for datacenter scalability. *IEEE JSAC*, 2014.

[68] Ori Rottenstreich and J'anos Tapolcai. Lossy compression of packet classifiers. In *ANCS*, 2015.

[69] SciPass. `http://globalnoc.iu.edu/sdn/scipass.html`.

[70] Siddhartha Sen, David Shue, Sunghwan Ihm, and Michael J. Freedman. Scalable, optimal flow routing in datacenters via local link balancing. In *CoNEXT*, 2013.

[71] Scott Shenker. The future of networking and the past of protocols, 2011. Talk at Open Networking Summit.

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

[73] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. PAST: Scalable ethernet for data centers. In *CoNEXT*, 2012.

[74] Yu-Wei Eric Sung, Sanjay G. Rao, Geoffrey G. Xie, and David A. Maltz. Towards systematic design of enterprise networks. In *CoNEXT*, 2008.

[75] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 2003.

[76] David E. Taylor and Jonathan S. Turner. ClassBench: A packet classification benchmark. In *INFOCOM*, 2004.

[77] D. Thaler and C. Hopps. Multipath issues in unicast and multicast next-hop selection. RFC 2991, 2000.

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

[79] V.V. Vazirani. *Approximation Algorithms*. Springer, 2004.

[80] Andreas Voellmy and Paul Hudak. Nettle: Functional reactive programming of OpenFlow networks. In *PADL*, 2011.

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

[82] Rihua Wei, Yang Xu, and H. Jonathan Chao. Block permutations in boolean space to minimize TCAM for packet classification. In *INFOCOM*, 2012.

[83] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. In *NSDI*, 2013.

[84] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. In *SIGCOMM*, 2010.

[85] Minlan Yu, Jennifer Rexford, Xin Sun, Sanjay G. Rao, and Nick Feamster. A survey of virtual LAN usage in campus networks. *IEEE Communications Magazine*, 2011.

[86] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. FIREMAN: A toolkit for firewall modeling and analysis. In *IEEE Symposium on Security and Privacy*, 2006.

[87] Pamela Zave and Jennifer Rexford. The design space of network mobility. In *Recent Advances in Networking. ACM SIGCOMM*, 2013.

[88] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted cost multipathing for improved fairness in data centers. In *EuroSys*, 2014.

[89] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Haitao Zheng, and Y.Ben Zhao. Packet-level telemetry in large datacenter networks. In *SIGCOMM*, 2015.