EXPRESSION AND COMPOSITION OF OPTIMIZATION-BASED APPLICATIONS FOR SOFTWARE-DEFINED NETWORKING

Victor Heorhiadi

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2017

Approved by:

Michael K. Reiter

Jay Aikat

Theophilus Benson

Anupam Gupta

Vyas Sekar

ABSTRACT

Victor Heorhiadi: Expression and Composition of Optimization-based Applications for Software-Defined Networking
(Under the direction of Michael K. Reiter)

Motivated by the adoption of the Software Defined Networking and its increasing focus on applications for resource management, we propose a novel framework for expressing network optimization applications. Named the SDN Optimization Layer (SOL), the framework and its extensions alleviate the burden of constructing optimization applications by abstracting the low-level details of mathematical optimization techniques such as linear programming. SOL utilizes the *path abstraction* to express a wide variety of network constraints and resource-management logic. We show that the framework is general and efficient enough to support various classes of applications. We extend SOL to support composition of multiple applications in a fair and resource-efficient way. We demonstrate that SOL's composition produces better resource efficiency than previously available composition approaches and is tolerant to network variations. Finally, as a case study, we develop a new application for load balancing network intrusion prevention systems, called SNIPS. We highlight the challenges in developing the SNIPS optimization from the ground up, show SOL's (conceptually) simplified version, and verify that both produce nearly identical solutions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
API	Application Programming Interface
CPU	Central Processing Unit
DC	Data Center
DoS	Denial of Service
DPI	Deep Packet Inspection
GPU	Graphics Processing Unit
IDS	Intrusion Detection System
ILP	Integer Linear Program
IRC	Internet Relay Chat
IPS	Intrusion Prevention System
ISP	Internet Service Provider
LP	Linear Program
MCFP	Multi-Commodity Flow Problem
NFV	Network Function Virtualization
NIDS	Network Intrusion Detection System
NIPS	Network Intrusion Prevention System
ONOS	Open Network Operating System
RAM	Random Access Memory
REST	Representational State Transfer
SA	Simulated Annealing
SDN	Software-Defined Networking
SOL	SDN Optimization Layer
SLA	Service Level Agreement
TCAM	Ternary Content-addressable Memory

TCP	Transmission Control Protocol
TE	Traffic Engineering
UDP	User Datagram Protocol
VM	Virtual Machine

CHAPTER 1: Introduction

As modern networks grow increasingly complex, the demand for fine-grained network control increases. Custom data paths in a datacenter, fault tolerance logic and handling of transport layer protocols are only a few examples that can benefit from better programmability of the network. Unfortunately, traditional network routing protocols, commonly implemented in hardware switches and routers, provide little flexibility and room for experimentation. Software-Defined Networking (SDN) attempts to answer this challenge by moving all routing logic into software, while still maintaining the benefits of fast switching fabric.

The SDN view decouples the network into an "intelligent" control plane, a "naive" data plane, and a standardized communication interface between the two. The control plane (often referred to as simply the controller) maintains a global view of network topology and is well suited to run custom software. The data plane consists of multiple packet forwarding devices (e.g., switches) that only execute the instructions given to them by the controller using a standardized configuration protocol (e.g., OpenFlow [72]). Such decoupled design enables a high degree of network programmability.

SDN has been shown to be an an enabler for network management applications that may otherwise be difficult to realize using previously existing control-plane mechanisms. Recent work has used SDN-based applications to implement network configuration for a range of management tasks, far beyond simple routing. For example, traffic engineering (e.g., [82]) is easily achieved with SDN because of its centralized nature. Service chaining ensures that traffic is dynamically routed through a series of services (e.g., firewalls or network address translation middleboxes). Network function virtualization (NFV) takes this approach further by virtualizing said services and allowing

them to migrate as virtual machines (VMs) to different points in the network. These types of applications have been made more accessible with the help of SDN.

While this body of work has been instrumental in demonstrating the potential benefits of SDN, realizing these benefits still requires significant effort. In particular, at the core of many SDN applications are custom *optimization problems* to tackle various constraints and requirements that arise in practice. For instance, an SDN application might need to account for limited TCAM, link capacities, or middlebox capacities, among other considerations. Developing such formulations involves a non-trivial learning curve, a careful understanding of theoretical and practical issues, and considerable manual effort. Furthermore, when the resulting optimization problems are intractable to solve with state-of-the-art solvers (e.g., CPLEX or Gurobi), heuristic algorithms must be crafted to ensure that new configurations can be generated on timescales demanded by the application as relevant inputs (e.g., traffic matrix entries) change (e.g., [38, 61]). Without a common framework for representing network optimization tasks, reusing key ideas across applications or combining features into a new application is difficult. As such, many efforts reinvent common building blocks, e.g., ensuring that the rules output by the optimizations can fit inside the TCAM, or generating volume-aware load-balancing rules that also maintain flow affinity.

This dissertation aims at eliminating such redundant efforts and includes three major components. First, we present the design of SOL: a novel framework that can express a variety of network optimizations (outlined in Section 1.1 and detailed in Chapter 3). SOL is designed to support a variety of applications while abstracting away many of the challenging optimization details, thereby reducing the effort for creating new applications. Second, we extend SOL to support composition of multiple network management applications that require optimizations. Our goal is to enable fair, efficient, and fast composition of multiple applications without imposing the requirement that applications should be aware of other "neighboring" applications. The resulting system, Chopin, is

outlined in Section 1.2 and detailed in Chapter 4. Finally, to demonstrate the efficacy of SOL and Chopin features, we present a case study where we construct a novel application for managing Network Intrusion Prevention Systems (NIPS), outlined in Section 1.3 and detailed in Chapter 5. The resulting system, SNIPS, leverages an optimization for balancing workloads across intrusion prevention systems.

1.1 SOL: SDN Optimization Layer

Our goal in creating SOL is to raise the level of abstraction for writing SDN-based network optimization applications. There are two natural requirements for such a framework: 1) *generality* to express the requirements for a broad spectrum of SDN applications (e.g., traffic engineering, policy steering, load balancing, and topology management); and 2) *efficiency* to generate (near-) optimal configurations on a timescale that is responsive to application needs. Given the diversity of the application requirements and the trajectory of prior work in developing custom solutions (e.g., [81, 44, 42, 38, 61, 28, 15, 105, 82, 39]), generality and efficiency appear individually difficult, let alone combined. We show that it is indeed possible to achieve both generality and efficiency.

To this end, we introduce SOL in Chapter 3, a framework that enables SDN application developers to express high-level application goals and constraints. Conceptually, SOL is an intermediate layer that sits between the SDN optimization applications and the actual control platform. Application developers can create new network optimization capabilities and express their requirements using the SOL API. SOL then generates configurations that meet these goals, which can be deployed to SDN control platforms.

Key insight of SOL's design is its utilization of the *path abstraction*. Both routing policies and resource allocations are represented as functions of network paths, as opposed to nodes or links. As a result, SOL is a general framework capable of expressing a variety of network management goals. We elaborate on this and other benefits of path abstraction in Chapter 3.

1.2 Chopin: Composition of Multiple Optimization Applications

As SDN gains momentum, it gives rise to increasingly complex network deployments. In particular, deployments with multiple, specialized network management applications. SDN needs the ability to compose applications and their functionality on a single network, considering the growth of application diversity and recent potential emergence of SDN "app stores" [88, 70, 92].

Such composition presents new challenges in the deployment of multiple applications. While policy composition has been studied extensively and many solutions are available [46, 83, 79], optimal resource management remains a hard problem due to a wide range of applications and their demands (e.g., load balancing, power saving, service chaining, etc.) [81, 61, 38].

A high-level network optimization framework, such as SOL, offers a promising alternative for SDN resource management applications. Unfortunately, the original SOL design is not robust on three key dimensions discussed below, precluding it from being used as-is for composition:

- *Fairness:* Multiple applications lead to multi-objective optimizations, i.e., moving away from a single notion of optimality. Hence, the framework must fairly balance multiple optimization criteria.

- *Resource efficiency:* multiple applications introduce variability into the network, since one application's routing decisions impact the decision-making process of co-existing applications by modifying network state. However, optimizations are sensitive to such data uncertainty, possibly causing the solution quality to degrade [7].

- *Responsiveness:* When resource efficiency suffers from traffic dynamics and competing applications, traffic allocations must be recomputed to offset these inefficiencies. Doing so, however, requires the ability to recompute and redeploy quickly, so as to be responsive at the timescales needed for the applications.

To address aforementioned challenges, we propose Chopin[1] in Chapter 4 — a framework that enables robust composition of resource-management SDN applications. Chopin's design provides several attractive features. First, Chopin provides transparent composition. By exposing single-application APIs to the developers, Chopin abstracts composition details from the development process. The abstraction enables application-agnostic development. Second, Chopin supports multiple composition modes based on different fairness metrics (e.g., [34, 54, 3]), providing flexibility for the operator. Finally, Chopin produces near-optimal composition results in short timescales, in order to respond to traffic variations.

1.3 SNIPS: Scalable Network Intrusion Prevention

To demonstrate the process and benefits of creating a new application using SOL, we present a novel approach for managing Network Intrusion Prevention Systems using SDN in Chapter 5.

We choose NIPS as a case study for a number or reasons. First, the ubiquity of such systems; NIPS are an integral part of today's network security infrastructure [111]. Second, NIPS require new scaling approaches to process increasing volumes of network traffic. Third, the complexity of modeling the NIPS scaling problem as an optimization problem serves as a good proving ground for Chopin's capabilities. We present a first-principles optimization to implement SNIPS capabilities, and compare it with the Chopin-enabled optimization. We highlight the benefits of using Chopin to create a complex application such as SNIPS.

Together, SOL, Chopin and SNIPS support the following thesis statement: *An optimization framework that leverages the network path abstraction can support expression and composition of complex network optimizations. Such a framework can compute near-optimal, resource-efficient traffic allocations at network time scales.*

[1] Allusion to Frédéric Chopin, a Polish 19th century classical composer

CHAPTER 2: Background and Prior Work

This chapter provides background on Software Defined Networking and optimization — two fields around which this dissertation revolves. We also summarize closely related work on optimization frameworks and network intrusion prevention.

2.1 Software-Defined Networking

Some of the ideas that form Software-Defined Networking, such as control and data plane separation [103, 59], active [100, 87] and programmable [10] networks, date back to the 1990s. We, however, focus on the most recent incarnation of SDN, as described by Open Networking Foundation [27]. This embodiment gained popularity after a work describing the OpenFlow protocol [63] was published. We highlight that SDN and OpenFlow are not synonymous and can exist separately from each other. However, OpenFlow has become a de-facto standard protocol for switch management in the academic literature.

Of particular relevance to our work are the subsequent improvements to the controllers [78, 32, 8, 71] and applications running atop the controllers, which we describe in detail below.

2.1.1 SDN Applications

At their core, all SDN applications follow a similar control loop: collect information about the network, make routing decisions, and change the network state to reflect those decisions. As an example, Aster*x [35] is an application developed to reactively perform flow load balancing using SDN. B4 [44] and SWAN [42] devise applications that imple-

ment max-min fairness for flow allocation.

Going beyond simple flow management, SDN applications can be used to manage middleboxes and even create energy savings in the datacenter . For example, ElasticTree [38] and Response [105] leverage redundant path structure in modern datacenters to power off switches and/or links at low-demand times to reduce energy costs. Scissors [52] tackles power efficiency by reducing packet header sizes.

SIMPLE [81] utilizes SDN to route traffic through a series of middleboxes according to a global policy, while maintaining acceptable levels of load on switches, network links, and middleboxes. Aplomb [93] is a system that leverages network programmability to offload middlebox processing outside of the network (e.g., a cloud provider).

2.1.2 Higher-layer Abstractions for SDN

In addition to building new and improved applications, other works explore new and improved *ways of making* applications. This includes new programming languages (e.g., [83, 25]), testing and verification tools (e.g, [53]), semantics for network updates (e.g., [84]), compilers for rule generation (e.g., [51]), abstractions for handling control conflicts (e.g., [4]), and APIs for users to express requirements (e.g., [22]).

In this section we focus on systems and languages that allow expression of multiple types of SDN applications. For example, Merlin is a language for network resource management and captures a number of requirements present in resource management applications [97]. Merlin uses regular expressions to express routing policies on a set of packets similar to that of Frenetic framework [25]. In contrast, SOL is designed to does not limit policies to the set of regular languages and introduces new approaches to solving resource-management optimization that are orders of magnitude faster than Merlin.

DEFO is another optimization framework that focuses on traffic engineering and service chaining applications [36]. Unlike SOL, DEFO is not to a general-purpose framework, but rather is designed to support easy management of carrier-grade networks.

DEFO accomplishes using a two-layer architecture and support for networks that are not OpenFlow-enabled via segment routing.

Finally, Maple [108] allows expression of SDN applications using an "algorithmic policy". Conceptually it allows the developer to run a function on each network packet that determines the packet's forwarding path. However, Maple does not take into resource-management constraints.

2.1.3 Composition of Multiple SDN Applications

Composing multiple SDN applications running atop a single controller presents an a new set of challenges. First, one must ensure that there are no conflicts with respect to the routing policies different applications are trying to enforce. Second, the applications are "competing" for a finite set of network resources, each attempting to deploy a solution that optimizes for its own goal.

Works such as Covisor [46] and Frenetic [25, 67] focus on OpenFlow rule composition and packet-level policy chain composition. PGA [79], Statesman [99], and PANE [22] are systems that focus on resolving policy conflicts between applications and enforcing network invariants.

The closest work that focuses on composition of resource-management applications is Corybantic [4] and its successor Athens [5]. These systems utilize voting protocols to generate a global network configuration across applications, and require modified, composition-aware applications. Chopin, in contrast, does not require applications to be aware of each other, provides automatic resource conflict resolution, and allows fair composition of applications.

2.2 Optimization

As we mentioned earlier, SDN enables a centralized control plane to perform all network management decisions. This naturally lends itself to algorithms that produce the

best possible solution — global optimization algorithms. In particular, many problems from the networking domain are modeled using specific subfields of mathematical optimization. In particular, convex optimization, where convex functions are minimized (or maximized) over convex sets. A special case of convex optimization is linear programming, which we desribe below.

2.2.1 Linear Programming (LP)

Linear programming is a special case of mathematical optimization that limits the objective and constraint functions to be linear. This ensures that the problem can be modeled using a system of linear inequalities, which enables fast solution (i.e., solvable in polynomial time). More formally, a linear program in standard form is expressed as follows:

$$\begin{aligned} &\texttt{maximize } f_0(x) \\ &\texttt{Subject to} \\ &\quad f_i(x) \leq b_i \qquad\qquad \text{for } i = 1..m \\ &\quad x \geq \vec{0} \end{aligned}$$

where a vector x denotes the decision variables whose values must be assigned to maximize (or minimize) $f_0(x)$, subject to m constraints $f_i(x)$. Each function $f_0(x) : \mathbb{R}^n \rightarrow \mathbb{R}$ maps the n-dimensional vector x to a real number and b is a vector of constants. Note that if any of the variables in x must take on an integer value, the problem becomes an Integer-Linear Program (ILP) and finding the optimal solution becomes NP-hard [74].

Languages for optimization: Prevalence of linear programming has lead to creation of various solvers and modeling tools. In particular, CPLEX [43] and Gurobi [33] are two well known commercial, general-purpose solvers that incorporate state-of-the-art algorithms for solving linear programs. Multiple modeling frameworks/languages such as

AMPL [26], Mosek [68], PyOpt [77], and PuLP [66] were built for easier modeling and expressing of optimization tasks. However, these tools do not specifically target network optimization.

2.2.2 Network Optimization

Many classic networking problems for which custom algorithms were designed (e.g., maximum flow and shortest path routing problems) can also be expressed as linear programs [1]. In fact, they are simply special cases of a broad class of problems, called network flow problems. As the name implies, the vector of decision variables x refers to the amount of flow routed along a network edge (or path) while optimizing for a given objective function. Of particular interest to computer networking is the multi-commodity flow problem (MCFP) [91]. MCFP computes the best routing for multiple, concurrent commodities on a given network. A commodity has an origin, a destination and a demand, whilst network links have capacities. Naturally, MCFP presents a convenient abstraction for modern IP networks.

Edge and Path formulations: There are two ways of writing network flow problems using linear programs. The most common is a node-edge (also referred to as node-arc or node-link) formulation, where the decision variables are the amount flow traveling on a given edge, and the constraints capture the amount of flow that enters and exits each node. An alternative approach is the path formulation, where the decision variables represent fraction of flow on each path. The two are equivalent, but trade off larger numbers of variables (the path formulation, since the number of paths in a graph is large) versus larger numbers of constraints (the node-edge formulation) [1].

2.2.3 General Optimization Enhancements

Due to the applicability of linear programming (and its more general parent, convex optimization) to many other fields, a number of extensions to both were developed to

better model real-world problems.

Multi-objective optimization: Extensive literature exists on multi-objective optimization (e.g., [98, 17]). The goal of multi-objective optimization is to compute solutions that simultaneously optimize multiple objectives. In many cases, finding a solution that is optimal for all objectives is impossible, and therefore new notions of optimality must be introduced (e.g., pareto optimality, ordered optimization).

Robust optimization: The field of robust optimization [7] develops ways of "protecting" optimizations against uncertain data. More specifically, it deals with the cases where the values used in constraint or objective functions $f_i(x),\ i = 0..m$ are not known but can be bound. In networking this takes on the form of network design validation against failures [14] or (semi-)oblivious routing [6, 56, 58]. Unfortunately, such techniques can be overly conservative and computationally intensive, especially with multiple resources involved in addition to bandwidth.

2.3 Network Security

Since this dissertation presents a case study focused on managing network security appliances (in particular, Intrusion Prevention Systems), we present a cursory overview of literature on NIPS followed by works that explore the interplay between SDN and network security.

2.3.1 Network Intrusion Detection and Prevention

The goal of network intrusion detection and prevention is to flag (and respectively, block) malicious behavior on the network. Coarsely, such systems can be divided into two categories, signature matching (looking for known attacks using pattern matching, e.g., [76, 85]) and anomaly detection [18]. Of interest to us are signature matching systems, since they are commonly deployed.

Signature matching systems face a continuous scaling struggle, due to computationally expensive expression matching, commonly called deep packet inspection (DPI). Unlike network forwarding, which operates on data- and network-layer headers, DPI requires processing packet payloads to guard against application-layer attacks. Such high costs make it difficult for NIPS to keep up with the "line rate" of a network, requiring novel scaling approaches besides faster hardware.

2.3.1.1 Scaling approaches

Traditional NIPS/NIDS scaling: There are several complementary approaches for scaling NIPS, including algorithmic improvements [96], using specialized hardware such as TCAMs (e.g., [112, 64]), FPGAs (e.g., [60]), or GPUs (e.g., [106, 45]).

On-path offloading: Work by Sekar et al. explores on-path offloading [90, 89], where the traffic is either processed or ignored by different NIPS machines along a routing path. They employ a centralized optimization framework to optimally balance the processing responsibilities across a network. One of the benefits of network offloading is the ability to load-balance without access to internals of NIPS software. Thus the load-balancing can be accommodate a variety of NIPS types, including legacy and proprietary solutions.

Off-path offloading: Recent efforts make the case for virtualizing NIPS-like functions [30] and demonstrate the viability of off-path offloading using public cloud providers [93]. Heorhiadi et al. developed a system which focuses on offloading for passive monitoring system, where the traffic is simply replicated to a datacenter or cloud provider [40]. However, this prior work does not model rerouting or the impact on user-perceived latency, making them less applicable to intrusion prevention.

2.3.2 Security Applications of SDN

Recent work has recognized the potential of SDN for security tasks. For example, FRESCO uses SDN to simplify botnet or scan detection [95]. Bohatei [20] is a system that leverages SDN to provide elastic denial of service (DoS) defense. SPIFFY [49] focuses on a specific type of DoS attack caused by link flooding and presents a way to mitigate the attack using SDN. We treat such work as completely orthogonal to contributions proposed in this thesis. Other systems, such as SIMPLE [81] and SoftCell [47], use SDN for steering traffic through a desired sequence of waypoints. Because they also consider optimal resource management, we view them as "client" applications that can benefit from our contributions.

CHAPTER 3: SOL — SDN Optimization Layer [1].

We start by developing a baseline framework that allows expression of different types of network optimizations. The high-level overview is shown in Figure 3.1. Recall that two major challenges in developing a framework such as SOL is creating a general abstraction capable of supporting a variety of applications and generating efficient solutions.

The key insight in SOL to achieve generality is that many network optimization problems can be expressed as *path-based* formulations. Paths are a natural abstraction for application developers to reason about intended network behaviors and to express policy requirements. For example, we can use paths to specify service chaining requirements (e.g., each path includes a firewall and intrusion-detection system, in that order) or redundancy (e.g., each includes two intrusion-prevention systems, in case one fails open). Finally, it is easy to model device (e.g., TCAM space, middlebox CPU) and link resource consumption based on the volume of traffic flowing through paths that traverse that device or link.

The natural question is whether the generality of path-based formulations precludes efficiency. Indeed, if implemented naively, optimization problems expressed over the paths that traffic might travel will introduce efficiency challenges since the number of paths grows exponentially with the network size. Our key insight is that by combining infrequent, offline preprocessing with simple, online *path-selection algorithms* (e.g., shortest paths or random paths), we can achieve near-optimal solutions in practice for all applications we considered. Moreover, SOL is typically far more efficient than solving the optimization problems originally used to express these applications' requirements.

[1]This chapter is excerpted from previously published work [41]

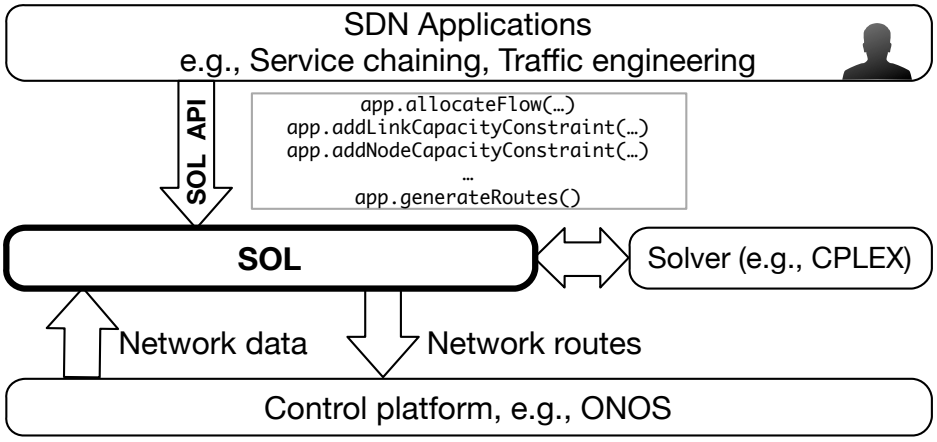


Figure 3.1: Developers use the SOL high-level APIs to specify optimization goals and constraints. SOL generates near-optimal solutions and produces device configurations that are input to the SDN control platform.

We have implemented SOL as a Python-based library that interfaces with ONOS [8] and prototyped numerous SDN applications in SOL, including SIMPLE [81], ElasticTree [38], Panopticon [61].

Our evaluations on a range of topologies show that: 1) SOL outperforms several applications' original optimization algorithms by an order of magnitude or more, and is even competitive with their custom heuristics; 2) SOL scales better than other network management tools like Merlin [97]; 3) SOL substantially reduces the effort required (e.g., in terms of lines of code) for implementing new SDN applications by an order of magnitude; and 4) optional SOL extensions can reduce route churn substantially across reconfigurations with modest impact on optimality.

3.1 Motivation and Challenges

First, we describe representative network applications that could benefit from a framework such as SOL. We highlight the need for careful formulation and algorithm development involved in prior efforts, as well as the diversity of requirements they entail.

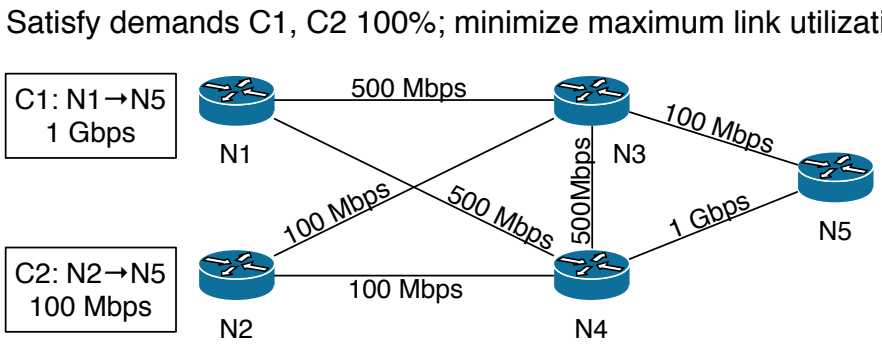


Figure 3.2: Traffic engineering applications

3.1.1 Traffic engineering

Traffic engineering (TE) is a canonical application that was an early driving application for SDN [44, 42]. Figure 3.2 shows an example where traffic classes C1 and C2 need to be routed completely while minimizing the load on the most heavily loaded link. A TE application takes as input traffic demands (e.g., the traffic matrix between WAN sites), a specification of the traffic classes and priorities, and the network topology and link capacities. It determines how to route each class to achieve network-wide objectives, e.g., minimizing network link load [24] or weighted max-min fairness [44, 42].

Challenges: Simple goals like link congestion can be represented and solved via max-flow formulations [1]. However, the expressivity and efficiency quickly breaks down for more complex objectives such as max-min fairness, which multiple research efforts have sought to address [42, 16, 44]. When max-flow like formulations fail, designers invariably revert to "low-level" techniques such as linear programs (LP) or combinatorial algorithms. Neither is ideal—using/tuning LP solvers is painful as they expose a very low-level interface, and combinatorial algorithms require significant theoretical expertise. Finally, translating the algorithm output into actual routing rules requires care to install volume-aware rules to truly reap the benefits of the optimization [109].

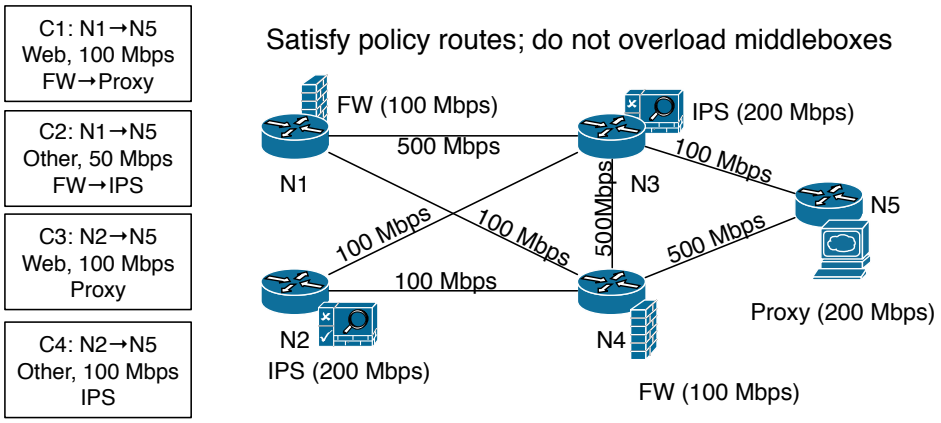


Figure 3.3: Service chaining applications

3.1.2 Service chaining

Networks today rely on a wide variety of middleboxes (e.g., IDS, proxy, firewall) for performance, security, and external compliance capabilities (e.g., [93]). The goal of service chaining is to ensure that each class of traffic is routed through the desired sequence of network functions. For example, in Figure 3.3, class C1 is required to traverse a firewall and proxy in order. Such policy routing rules must be suitably encoded within the available TCAM on SDN switches [81]. Since middleboxes are often compute-intensive, they can get easily overloaded and thus operators would like to balance the load on these appliances [81, 29]. The key inputs to such applications are the service chaining requirements of different classes, traffic demands, and the available middlebox processing resources. The application then sets up the forwarding rules such that the service chaining requirements are met while respecting the switch TCAM and middlebox capacities. Furthermore, as many of these middleboxes are stateful, these rules must ensure flow affinity.

Challenges: Service chaining introduces more complex requirements when compared to TE applications. First, modeling the consumption of switch TCAM introduces discrete components into the optimization, which impacts scalability [81]. Second, such service processing requirements fall outside the scope of existing network flow abstrac-

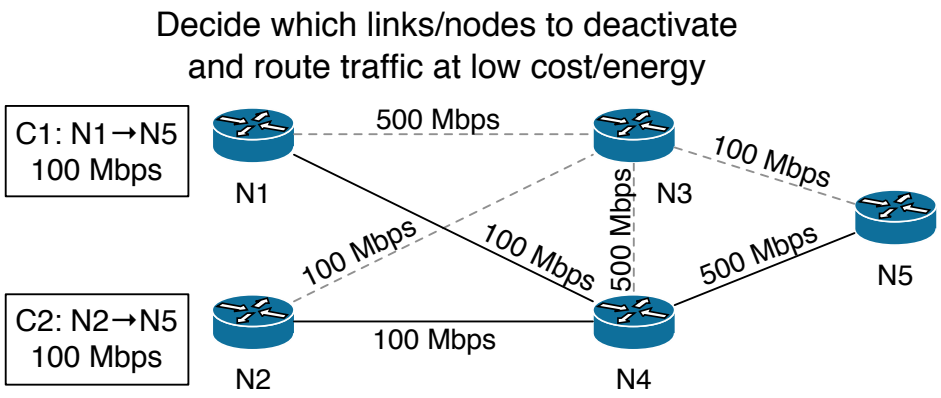


Figure 3.4: Topology reconfiguration applications

tions [15]. Third, service chaining highlights the complexity of combining different requirements; e.g., reasoning about the interaction between the load balancing algorithm and the switch TCAM constraints is non-trivial [47]. Existing service chaining efforts developed custom heuristics [12] or new theoretical extensions [15]. Furthermore, as observed previously, ensuring flow affinity can be quite tricky [40, 39].

3.1.3 Flexible topology management

SDN enables topology modifications that would be difficult to implement with existing control plane capabilities. For instance, ElasticTree [38] and Response [105] use SDN to dynamically switch on/off network links and nodes to make datacenters more energy efficient. In Figure 3.4, these applications might shut down node N3 during periods of low utilization, if classes C1 and C2 can be routed via N4 without significantly impacting end-to-end performance. Topology reconfiguration is especially feasible in rich topologies with multiple paths between every source and destination. Such applications take as input the demand matrix (similar to the TE task) and then compute the nodes and links that should be active and traffic-engineered routes to ensure performance SLAs.

Challenges: The on-off requirement on the switches/links once again introduces discrete constraints, yielding integer-linear optimizations that are theoretically intractable

and difficult to express using max-flow like abstractions. Solving such problems requires significant computation even on small topologies and thus forces developers to design new, heuristic solving strategies; e.g., ElasticTree uses a greedy bin-packing algorithm [38].

3.1.4 Network function virtualization

Prior work has leveraged SDN capabilities to offload or outsource network functions to leverage clusters or clouds [93, 30, 82]. This is especially useful for expensive deep-packet-inspection services (as we show in [39], and Chapter 5). The key decision here is to decide how much of the processing on each path to offload to the remote datacenter — e.g., in Figure 3.5, how much of class C1 traffic should be routed to the datacenter between N4 and N5 for IPS processing, versus processing it at N3. Offloading can increase user-perceived latency and impose additional load on network links. Moreover, some active functions (e.g., WAN optimizers or IPS) induce changes to the observed traffic volumes due to their actions. Thus, optimizing such offloading must take into account the congestion that might be introduced, as well as latency impact and any traffic volume changes induced by such outsourced functions. Further generalizations have considered not only offloading middlebox services but also elastically scaling them [73, 28, 69, 9], exacerbating these issues.

Challenges: Such offloading and elastic scaling opportunities introduce new dimensions to optimization that are difficult to capture. For instance, offloading requires rerouting the traffic and thus optimizations must model the impact on link loads, downstream nodes, and TE objectives. If done naively, this can introduce non-linear dependencies since the actions of downstream nodes depend on control decisions made upstream. The active changes to traffic volumes by some functions (e.g., compression for redundancy elimination or drops by IPS) also introduce non-linear dependencies in the optimization. Finally, elastic scaling introduces a discrete aspect to the problem similar

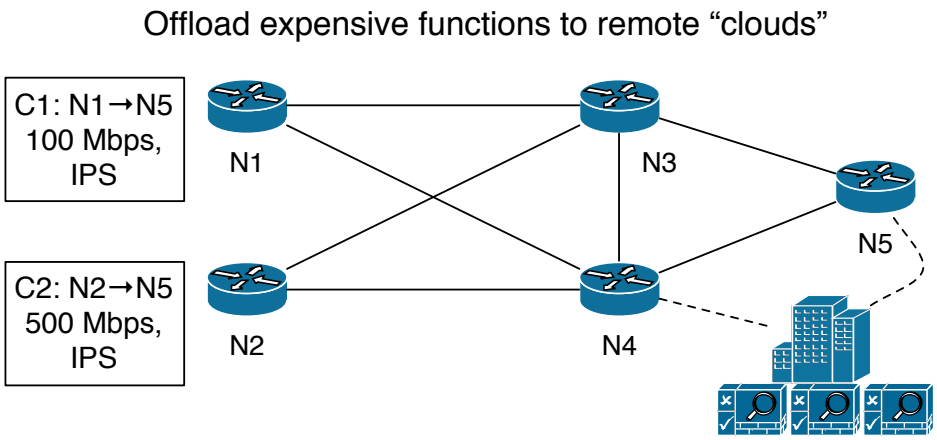


Figure 3.5: Offloading network functions

to the topology modification application, further decaying the problem's tractability.

3.1.5 Motivation for SOL

Drawing on the above discussion, we summarize a few key considerations:

- Network applications have diverse and complex optimization requirements; e.g., service chaining requires us to reason about valid paths while topology modification needs to enable/disable nodes.
- Designers of these applications have to spend significant effort in expressing and debugging these problems using low-level optimization libraries.
- It can take non-trivial expertise to ensure that the problems can be solved fast enough to be relevant for operational timescales, e.g., recomputing TE every few minutes or periodically solving the large integer-linear programs (ILPs) supporting topology reconfiguration (e.g., [38]).

3.2 SOL Overview

Our overarching vision in developing SOL is to raise the level of abstraction in developing new SDN applications and specifically to eliminate some of the black art in

developing SDN-based optimizations, making them more accessible for deployment by network managers. To do so, SOL abstracts away low-level details of optimization solvers and SDN controllers, allowing the developer to focus on the high-level application goals (recall Figure 3.1). SOL takes as inputs the network topology, traffic patterns, and optimization requirements in the SOL API. It then translates these into constraints for optimization solvers such as CPLEX or Gurobi. Finally, SOL interfaces with existing SDN control platforms such as ONOS to install the forwarding rules on the SDN switches. SOL does not require modifications to the existing control or data plane components of the network. Our vision for SOL stands in stark contrast to the state of affairs today, in which a developer faces programming a new SDN optimization either directly for a generic and low-level optimization solver such as CPLEX or using a heuristic algorithm designed by hand, after which she must translate the decision variables of the optimization to device configurations.

Path abstraction: For SOL to be useful and robust, we need a unifying abstraction that can capture the requirements of diverse classes of SDN network optimization applications described in the previous section. SOL is built using *paths* through a network as a core abstraction for expressing network optimization problems. This is contrary to how many optimizations are formulated in the literature — using a more standard edge-centric approach [1]. In our experience, however, an edge-centric approach forces complexity when presented with additional requirements, especially ones that attempt to capture path properties [61, 38].

In contrast, path-based formulations capture these requirements more naturally. For instance, much of the complexity in modeling service chaining or network function offloading applications from Section 3.1 is in capturing the path properties that need to be satisfied. With a path-based abstraction, we can simply define predicates that specify valid paths — e.g., those that include certain waypoints or that avoid a certain node (to anticipate that node's failure). In addition, we can model path-based resource use with

ease. For example, usage of TCAM space in a switch corresponds to a traffic-carrying path traversing that switch (and thus a rule to accommodate that path). Without the path abstraction, modeling such constraints is difficult (cf., [81]). Finally, expressing constraints on nodes and edges does not introduce increased difficulty compared to edge-centric approach.

Scalability: In a pure flow-routing scenario, an edge-based formulation admits simple algorithms that guarantee polynomial-time execution. Path-based formulations, on the other hand, are often dismissed because of their inefficient appearance — after all, in the worst case, the number of paths in the network is exponential in the network size — or due to the complexity of algorithms to solve path based formulations (column-generation, decompositions, etc. [1]). However, in many practical scenarios, the number of valid paths (as defined by the application) is likely to be significantly smaller. Furthermore, multipath routing can provide only so much network diversity before its value diminishes [62]. So, the set of paths that need to be considered is not large.

SOL leverages an off-line path generation step to determine valid paths (step 1 of Figure 3.6). Since for most applications, the set of valid paths is fairly static and does not need to be recomputed every time the optimization is run, we expect this step is infrequent. Next, SOL *selects* a subset of these paths (step 2) using a selection strategy (see Section 3.4) and runs the optimization with only the selected paths as input (step 3), to ensure that the optimization completes quickly. We show in Section 3.7 that this strategy still permits inclusion of sufficiently many paths for the optimization to converge to a (near) optimal value. So, while the efficiency of path-based optimization is a valid theoretical concern, in practice we show that there are practical heuristics to address this issue.

Generating device configurations: SOL translates the decision variables from the SOL optimization to network device configurations to implement appropriate flow routing (step 4 of Figure 3.6). The algorithm utilized in SOL to perform this translation is

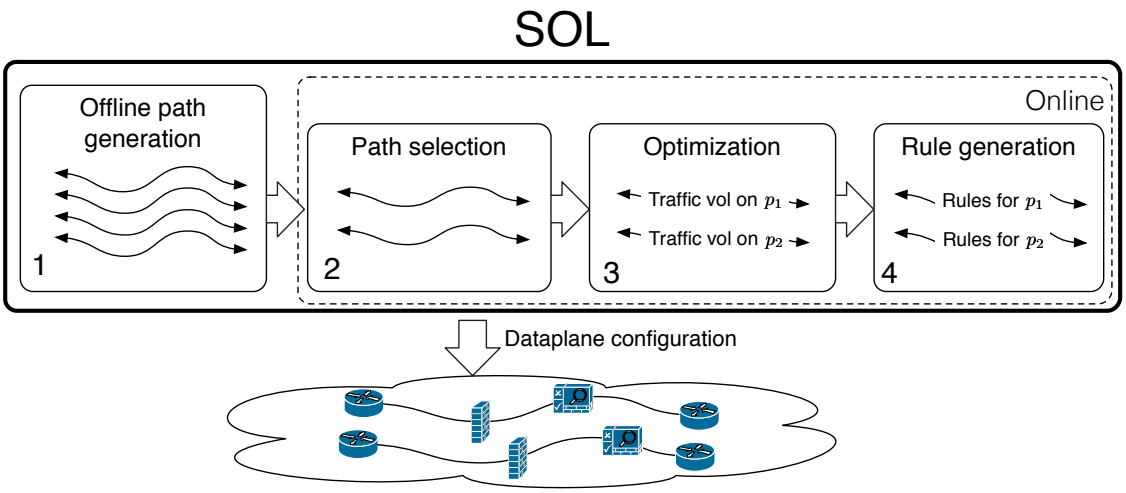


Figure 3.6: SOL architecture, overview of the workflow

based on that in previous work [109, 39]. However, because the optimization is path-based, the algorithm is more straightforward and requires fewer steps.

3.3 SOL Detailed Design

In this section, we present the detailed design of SOL. We focus on the high-level API that the SDN application developer would use to express applications via SOL, and the impact of these API calls on the SOL's internal representation of the optimization problem. Note, however, that the developer "thinks" in terms of the high-level API rather than low-level details of dealing with the solver-level variables, how paths are identified, etc.

A developer begins a new optimization in SOL by instantiating an `opt` object via the `getOptimization` function and then building the optimization using *constraint templates*, which we explain below (summary of the templates if provided in Table 3.3 on page 31).

`nodes`	Set of all nodes, part of the topology
`links`	Set of all links, part of the topology
`classes`	Set of all traffic classes
`paths`(c)	Paths available for class $c \in$ `classes`; output by path-selection stage (Section 3.4)

Table 3.1: Network data input

	Variable	Description
Decision	$x_{c,p}$	Fraction of class-c flows allocated to path $p \in$ `paths`(c); non-integer
	b_p	Is path p used; binary
	b_v	Is node v used; binary
	b_l	Is link l used; binary
	$capvar_v^r$	Capacity allocated for resource r at node v; non-integer
Derived	a_c	Fraction of c's "demand" routed; non-integer
	$load_l^r$	Amount of resource r consumed by flows routed over link l; non-integer
	$load_v^r$	Amount of resource r consumed by flows routed via node v; non-integer

Table 3.2: Variables internal to the optimization

3.3.1 Preliminaries

Data inputs: There are two basic data inputs that the developer needs to provide to any network optimization. First, the network topology is a required input, specified as a graph with `nodes` and `links`. It also contains metadata of node/edge types or properties; e.g., nodes can have designated functions like "switch" or "middlebox". Second, SOL needs a specification of *traffic classes*, where each class c has associated ingress and egress nodes and some expected traffic volume. Each class can (optionally) be associated with a specification of the "processing" required for traffic in this class, e.g., service chaining. Finally, to each traffic class c is associated a set `paths`(c) available to route flows in class c; `paths`(c) is output by a path-selection preprocessing step described in Section 3.4.

Internal variables: SOL internally defines a set of variables summarized in Figure 3.2. We reiterate that the developer does not need to reason about these variables and uses a high-level mental model as discussed earlier. There are two main kinds of

variables:

- *Decision variables* that identify key optimization control decisions. The most fundamental decision variable is $x_{c,p}$, which captures traffic routing decisions and denotes the fraction of flow for a traffic class c that path $p \in \texttt{paths}(c)$ carries. This variable is central to various types of resource management applications as we will see later. To capture topological requirements (e.g., Section 3.1.3), we introduce three binary decision variables b_p, b_v, and b_l that denote whether each path, node or link (respectively) is enabled ($= 1$) or disabled ($= 0$). The variable $capvar_v^r$ is the SOL-assigned allocation of resource-r to node v.

- *Derived variables* are functions defined over the above decision variables that serve as convenient "shorthands". a_c denotes the total fraction of flow for class c that is carried by all paths. The load variables $load_v^r$ and $load_l^r$ model the consumption of resource r on node v and link l, respectively.

3.3.2 Routing requirements

Routing constraints control the allocation of flow in the network. `AllocateFlow` creates the necessary structure for routing the traffic through a set of paths for each traffic class. Some network applications try to satisfy as much of their flow demands as possible (e.g., max-flow) while others (e.g., TE) want to "saturate" demands. For example, a developer of a TE application (Section 3.1.1) would like to route all traffic though the network, and thus she would add the following high-level routing constraint templates to her empty `opt` object:

```
opt.AllocateFlow()
opt.RouteAll()
```

In contrast, a simple max-flow would only need `AllocateFlow` since there is no requirement on saturating demands in that case.

The `EnforceSinglePath`(C) constraint forces a single flow-carrying path per class $c \in C$, preventing flow-splitting and multi-path routing. Finally, `FlowAffinityConstraint`(C_{pair}) ensures that for each pair of traffic classes $c_1, c_2 \in C_{pair}$ equal amount of processing from each class occurs at the same nodes, thus maintaining flow affinity for processing that requires session-level (as opposed to flow-level) granularity.

Internals: `AllocateFlow` ensures that the total traffic flow across all chosen paths for the class c matches the variable a_c.

$$\forall c \in \texttt{classes} : \sum_{p \in \texttt{paths}(c)} x_{c,p} = a_c$$

Similarly, `RouteAll` implies:

$$\forall c \in \texttt{classes} : \ a_c = 1$$

`EnforceSinglePath` ensures that only a single path is used for routing traffic, and is expressed as follows:

$$\forall c \in \texttt{classes} : \sum_{p \in \texttt{paths}(c)} b_p = 1$$

`FlowAffinityConstraint` ensures that the load on each processing node contributed by each path in both forward and reverse directions is equal:

$$\forall c_1, c_2 \in C_{pair} \forall v : \sum_{p \in \texttt{paths}(c_1) : v \in p} x_{c_1,p} = \sum_{p \in \texttt{paths}(c_2) : v \in p} x_{c_2,p}$$

3.3.3 Resource capacity constraints

As we saw in Section 3.1, SDN optimizations have to deal with a variety of capacity constraints for network resources such as link bandwidth, switch rules, and middlebox CPU and memory. SOL allows users to write custom resource management logic

by specifying several "cost" functions, depicted in Figure 3.7. These functions prescribe how to compute the cost of routing traffic through a link, a node, or a given path. SOL provides default implementations of these for common tasks, but allows the user to specify their own logic, as well, as we will show later (Section 3.5).

These cost functions can then be passed into constraint templates. For example, to add a constraint that limits link usage, the user can invoke the template function `LinkCapacityConstraint` with a resource that we are constraining (e.g., 'bandwidth'), a map of links to their capacities,[2] and optionally, a custom `linkCapFn` to compute the cost of traffic on a link.

```
opt.LinkCapacityConstraint ('bandwidth',
    {(1,2): 10**7, (2,3): 10**7},
    defaultLinkFunction)
```

This indicates that bandwidth should not exceed 10 Mbps for links 1-2 and 2-3. Note that the default function is purely for illustration; the developer can write her own `linkCapFn` (recall Figure 3.7).

`NodeCapacityPerPathConstraint` generates constraints on the nodes that do not depend on the traffic, but rather on the routing path. That is, the cost of routing at a node does not depend on the volume or type of traffic being routed; it depends on the path and its properties. The best example of such usage is accounting for the limited rule space on a network switch (e.g., Section 3.1.2). If a path is "active", the rule must be installed on each switch to support the path.

Internals: `LinkCapacityConstraint` and `NodeCapacityConstraint` rely on `linkCapFn` and `nodeCapFn`, respectively, to compute the cost of using a particular resource at a link or node if all of the class-c traffic was routed to it. Internally, the load is multiplied by the $x_{c,p}$ variable to capture the load accurately, then the load is capped by a user-provided $lnCap$ ($ndCap$), which is a mapping of links (nodes) to capacities for a

[2]When capacities should be allocated by the optimization itself, a capacity of `TBA` (meaning To Be Allocated) can be specified, instead.

`linkCapFn`(l, c, p, r): Amount of resource type r consumed if all class-c traffic is allocated to path $p \ni l$ for link l

`nodeCapFn`(v, c, p, r): Amount of resource r consumed if all class-c traffic is allocated to path $p \ni v$ for node v

`nodeBudgetFn`(v): Cost of using node v; required with `BudgetConstraint`

`routingCostFn`(p): Cost of routing along path p; required with `minRoutingCost`

`predicate`(p): Determine whether any given path is valid by returning `True` or `False`

Figure 3.7: Customizable functions

given resource r.

$$\forall l \text{ in } lnCap:$$

$$load_l^r = \sum_c \sum_{p \in \texttt{paths}(c): l \in p} x_{c,p} \times \texttt{linkCapFn}(l, c, p, r)$$

$$load_l^r \leq lnCap[l]$$

Node capacity equations function similarly, and operates on nodes instead of links:

$$\forall v \text{ in } ndCap:$$

$$load_v^r = \sum_c \sum_{p \in \texttt{paths}(c): v \in p} x_{c,p} \times \texttt{nodeCapFn}(v, c, p, r)$$

$$load_v^r \leq ndCap[v]$$

The `NodeCapacityPerPathConstraint` functions a bit differently, as it depends on enabled paths:

$$\forall v \text{ in } ndCap:$$

$$load_v^r = \sum_c \sum_{p \in \texttt{paths}(c): v \in p} b_p \times \texttt{nodeCapFn}(v, c, p, r)$$

$$load_v^r \leq capvar_v^r$$

$$\text{if } ndCap[v] \neq \texttt{TBA} \text{ then } capvar_v^r = ndCap[v]$$

3.3.4 Node/link activation constraints

Next set of constraints, when used, allow developers to logically model the act of *enabling* or *disabling* nodes, links, and paths; e.g., for managing energy or other costs (e.g., Section 3.1.3). We identify two possible modes of interactions between these topology modifiers, and the optimization developer can choose the one that is most suitable for their context. 1) `RequireAllNodesConstraint` captures the property that disabling a node disables all paths that traverse it; and 2) `RequireSomeNodesConstraint` captures the property that enabling a node permits any path traversing it to be enabled, as well. The latter version is suitable when, e.g., a node can still route traffic even if its other (middlebox) functionality is disabled, and so a path containing that node is potentially useful as providing middlebox functions if at least one of its nodes is enabled. There are analogous constraint templates for links. A third constraint template, `PathDisableConstraint`, restricts a path to carry traffic only if it is enabled.

For example, a developer trying to implement the application from Section 3.1.3 can model the requirements for shutting off datacenter nodes by adding the `RequireAllNodesConstraint` and `PathDisableConstraint` templates:

```
opt.RequireAllNodesConstraint (trafficClasses)
opt.PathDisableConstraint (trafficClasses)
```

Other efficiency considerations may enforce a budget on the number of enabled nodes, to model constraints on total power consumption of switches/middleboxes, cost and budget of installing/upgrading particular switches, etc. These are captured via the `BudgetConstraint` template function.

Internals: Internally, these topology modification templates are achieved using the binary variables we introduced earlier. Specifically, the above requirements can be for-

malized as follows:

	$\forall p \in \texttt{paths}(c)$:
`RequireAllNodesConstraint`	$\forall v \in p:\ b_p \leq b_v$
`RequireSomeNodesConstraint`	$b_p \leq \sum_{v \in p} b_v$
`PathDisableConstraint`	$x_{c,p} \leq b_p$

Naturally, similar constraints are constructed for links. Note that `PathDisableConstraint` is crucial to the correctness of the optimization in that it enforces that no traffic traverses a disabled path.

`BudgetConstraint` allows the developer to cap the number of enabled nodes in the topology according to a custom cost function. More formally,

$$\sum_{v \in \texttt{nodes}} b_v \times \texttt{nodeBudgetFn}(v) \leq k$$

In its simplest form, where `nodeBudgetFn` returns 1, this constraint simply allows up to k nodes to be enabled.

3.3.5 Specifying network objectives

The goal of SDN applications is eventually to optimize some network-wide objective, e.g., maximizing the network throughput, balancing load, or minimizing total traffic footprint. Table 3.4 lists the most common objective functions, drawing on the applications considered in Section 3.1. For instance, the developer of a TE application may want to implement the objective of minimizing the maximum link load and thus add the following code snippet:

```
opt.setPredefinedObjective (minMaxLinkLoad, 'bandwidth')
```

Other optimizations (e.g., Section 3.1.4) may need to minimize the total routing

Group	Function	Description
Routing	`AllocateFlow`	Allocate flow in the network
	`RouteAll`	Route all traffic demands
	`EnforceSinglePath` (C)	For each $c \in C$, at most one $p \in$ `paths`(c) is enabled.
Capacities	`LinkCapacityConstraint` (r, $lnCap$, `linkCapFn`)	If l is in $lnCap$, then limit utilization of link resource r on link l to $lnCap[l]$.
	`NodeCapacityConstraint` (r, $ndCap$, `nodeCapFn`)	If v is in $ndCap$, then limit utilization of node resource r on node v to $ndCap[v]$.
	`NodeCapacityPerPathConstraint` (r, $ndCap$, `nodeCapFn`)	If v is in $ndCap$, then limit utilization of node resource r on node v by enabled paths to $ndCap[v]$.
	`CapacityBudgetConstraint` (r, N, $totCap$)	Limit total type-r resources allocated to nodes in $N \subseteq$ `nodes` to $totCap$. Used when SOL is allocating capacities.
Topology control	`RequireAllNodesConstraint` (C)	For each $c \in C$ and each $p \in$ `paths`(c), p can be enabled iff all nodes on p are enabled.
	`RequireSomeNodesConstraint` (C)	For each $c \in C$ and each $p \in$ `paths`(c), p can be enabled iff some node on p is enabled.
	`RequireAllEdgesConstraint` (C)	For each $c \in C$ and each $p \in$ `paths`(c), p can be enabled iff all links on p are enabled.
	`PathDisableConstraint` (C)	For each $c \in C$ and each $p \in$ `paths`(c), p can carry traffic only if it is enabled.
	`BudgetConstraint` (`nodeBudgetFn`, k)	Total cost of enabled nodes, as computed using `nodeBudgetFn`, is at most k.
Objective	`setPredefinedObjective` (name)	Set one of the predefined functions as the objective (see Table 3.4).

Table 3.3: Selected constraint template functions for building optimizations; see Figure 3.7 for `linkCapFn`, `nodeCapFn`, and `nodeBudgetFn`

cost and include a `minRoutingCost` objective. This objective is parameterized with `routingCostFn`(p); i.e., developers can plugin their own cost metrics such as number of hops or link weights. As shown, we also provide a range of natural load-balancing templates.

3.3.6 Minimizing reconfiguration changes

As the network state changes, the optimization can be recomputed to adjust traffic allocations along forwarding paths. Traditionally, there are no guarantees on the similarity of newly computed solution and previous solution. This can result in significant amount of network churn, where traffic is reallocated to different paths unnecessarily. To mitigate network churn, SOL supports an additional constraint, for simplicity dubbed "mindiff". The goal of mindiff is to limit the fraction of traffic that migrates from one path to another with respect to a previous solution.

We express the churn per class using the difference between flow fractions on each path:

$$\begin{aligned} \forall c \quad Churn_c &= \sum_{p \in \texttt{paths}(c)} z \\ \varepsilon &\geq x_{c,p} - \hat{x}_{c,p} \\ \varepsilon &\leq -x_{c,p} + \hat{x}_{c,p} \end{aligned}$$

where $\hat{x}_{c,p}$ is the fraction of traffic on path p for traffic class c in the previous solution. The global network churn is computed as follows:

$$Diff = \frac{1}{|\texttt{classes}|} \sum_{c \in \texttt{classes}} Churn_c$$

and can be either constrained globally (e.g., $Diff \leq .3$, less than 30% of traffic migrates)

`maxAllFlow`	maximize $\sum_{c \in \texttt{classes}} a_c$
`minMaxNodeLoad` (r)	minimize $\max_{v \in \texttt{nodes}} load_v^r$
`minMaxLinkLoad` (r)	minimize $\max_{l \in \texttt{links}} load_l^r$
`minRoutingCost`	$\sum_{c,p} \texttt{routingCostFn}(p) \times x_{c,p}$

Table 3.4: Common objective functions

or added to the objective function to be minimized. If adding to the objective function, an appropriate weight must be specified to avoid sacrificing the primary objective's optimality.

3.3.7 Low-level API

While the SOL API described above is general and expressive enough to capture the diverse requirements of the broad spectrum of applications, we also expose a low-level API that gives more control to the user by giving access to the SOL internal variables. Advanced users can use this API for further customization.

For instance, API calls enable the names of the internal variables in Figure 3.2 to be retrieved and their values determined. Similarly, using the `defineVar` ($name$, $coeffs$, lb, ub) function, the user can create a new variable with name $name$, specify numeric lower and upper bounds (lb and ub), and equate it to a linear combination of any other existing variables as specified by $coeffs$, a map from variable names to numeric coefficients. This is a useful primitive when specifying complex objectives. SOL also allows setting a custom objective function that is a linear combination of any existing variables, allowing for multi-objective optimization. This is done using the `setObjective` ($coeffs$, dir) function call, which accepts a mapping $coeffs$ of variable names to their coefficients. The binary input dir indicates whether the objective should be minimized or maximized.

3.4 Path generation and selection

Given these constraint templates, the remaining question is how we populate the path set `paths`(c) for each traffic class c to meet two requirements. First, each $p \in$ `paths`(c) should satisfy the desired policy specification for the class c. Second, `paths`(c) should contain paths for each class c that make the formulation tractable and yet yield near-optimal results. We describe how we address each concern next.

Generation: First, to populate the paths, SOL does an offline enumeration of all simple (i.e., no loops) paths per class.[3] Given this set, we filter out the paths that do not satisfy the user-defined predicate `predicate`, i.e., where `predicate`(p) =`True` only if p is a valid path. Note that we can generalize this to allow different predicates per class, but do not do so for ease of explanation.

In practice, we implement the predicate as a flexible Python callable function rather than constrain ourselves to specific notions of path validity (e.g., regular expressions as in prior work [97]). Using this predicate gives the user flexibility to capture a range of possible requirements. Examples include waypoint enforcement (forcing traffic through a series of middleboxes in order); enforcing redundant processing (e.g., through multiple IDS, in case one fails open); and limiting network latency by mandating shorter paths.

Selection: Using all valid paths per class may be inefficient since the number of paths grows exponentially with the size of the network, meaning that the LP/ILP that SOL generates will quickly become too large to solve in reasonable time. SOL thus provides path selection algorithms that choose a subset of valid paths (number of paths denoted as `selectNumber`) that are still likely to yield near-optimal results in practice. Specifically, two natural methods work well across the spectrum of applications we have considered: (1) shortest paths for latency-sensitive applications (`selectStrategy` = `shortest`) or (2) random paths for applications involving load

[3]This is to simplify the forwarding rules without resorting to tunneling or packet tagging [81].

```
SIMPLE_predicate = functools.partial(waypointMboxPredicate, order=('fw','ids'))
def SIMPLE_NodeCapFunc(node,tc,path,resource,nodeCaps):
  if resource=='cpu' and node in nodeCaps['cpu']:
    return tc.volFlows*tc.cpuCost/nodeCaps[resource][node]
capFunc = functools.partial(SIMPLE_NodeCapFunc, nodeCaps=nodeCaps)

def SIMPLE_TCAMFunc(node, tc, path, resource):
  return 1
# Path generation, typically run once in a precomputation phase
opt = getOptimization()
pptc = generatePathsPerTrafficClass(topo, trafficClasses, SIMPLE_predicate, 10, 1000,
     functools.partial(useMboxModifier, chainLength=2))
# Allocate traffic to paths
pptc = chooserand(pptc, 5)
opt.addDecisionVariables(pptc)
opt.addBinaryVariables(pptc, topo, ['path','node'])
opt.addAllocateFlowConstraint(pptc)
opt.addRouteAllConstraint(pptc)
opt.addLinkCapacityConstraint(pptc, 'bandwidth', linkCaps, defaultLinkFuncNoNormalize)
opt.addNodeCapacityConstraint(pptc, 'cpu',      {node: 1 for node in topo.nodes() if 'fw' or
     'ids' in topo.getServiceTypes(node)}, capFunc)
opt.addNodeCapacityPerPathConstraint(pptc, 'tcam', nodeCaps['tcam'], SIMPLE_TCAMFunc)
opt.setPredefinedObjective('minmaxnodeload','cpu')
opt.solve()
obj = opt.getSolvedObjective()
pathFractions = opt.getPathFractions(pptc)
c = controller()
c.pushRoutes(c.getRoutes(pathFractions))
```

Figure 3.8: Code to express SIMPLE [81] in SOL

balancing (`selectStrategy` = `random`). SOL is flexible to incorporate other selection strategies, e.g., picking paths with minimal node overlap for fault tolerance. We find `random` works well for many applications that require load balancing. We conjecture this is because choosing random paths on sufficiently rich topologies yields a high degree of edge-disjointedness among the chosen paths, yielding sufficient degrees of freedom for balancing loads.

3.5 Examples

Next, we show end-to-end examples to highlight the ease of using the SOL APIs to write existing and novel SDN network optimizations. These examples are actual Python code that can be run, not just pseudocode. By comparison, the code is significantly higher-level and more readable than the equivalent CPLEX code would be, as it does not need to deal with large numbers of underlying variables and constraints.

Service chaining (Section 3.1.2): As a concrete instance of the service chaining example, we consider SIMPLE [81]. SIMPLE involves the following requirements: route all traffic through the network, enforce the service chain (e.g., "firewall followed by IDS") policy for all traffic, load balance across middleboxes, and do so while respecting CPU, TCAM, and bandwidth requirements. Figure 3.8 shows how the SIMPLE optimization can be written in ≈ 25 lines of code. This listing assumes that topology and traffic classes have been set up, in the `topo` and `trafficClasses` objects, respectively.

The first part of the figure shows function definitions and the path generation step, which would typically be performed once as a precomputation step. We start by defining a path predicate (line 1) for basic enforcement through middleboxes by using the SOL-provided function with the middlebox order. The next few lines (lines 2–4) show a custom node capacity function to normalize the CPU load between 0 and 1. This computes the processing cost per traffic class (number of flows times CPU cost) normalized by the current node's capacity. Similarly, the TCAM capacity function captures that each path consumes a single rule per switch (line 7). The user gets the optimization object (line 10), and generates the paths (line 11), obtaining the "paths per traffic class" (`pptc`) object. The path generation algorithm is parameterized with the custom `SIMPLE_predicate`, a limit on path length of 10 nodes, and a limit on the number of paths per class of 1000. It is also instructed to evaluate every possible use of two middleboxes on a routing path for inclusion as a distinct path in the output.

The remaining lines show what would be executed whenever a new allocation of traffic to paths is desired. Line 13 selects 5 random paths per traffic class; lines 14–20 add the routing and capacity constraints. We use the default link capacity function for bandwidth constraints, and our own functions for CPU and TCAM capacity. Because the CPU capacity function normalizes the load, the capacity of each node is now 1 (line 19). The program selects a predefined objective to minimize the CPU load (line 21) and calls the solver (line 22). Finally, the program gets the results and interacts with the

SDN controller to automatically install the rules (line 26).

ElasticTree [38]: We only show the most important differences between ElasticTree and SIMPLE. There is no requirement on paths, and so `nullPredicate` is used for path generation. We use link binary variables (see line 1 below) and the node/link activation constraints (lines 2–4). Finally, we use the low-level API (recall Section 3.3.7) to define power consumption for switches and links (lines 5, 6, wherein "`opt.bn(node)`" and "`opt.be(u, v)`" retrieve the names of variables $b_{\texttt{node}}$ and $b_{(\texttt{u},\texttt{v})}$ from Figure 3.2, respectively) and use these variables to define a custom objective function (line 7).

```
1 opt.addBinaryVariables(pptc,topo,['path','node','edge'])
2 opt.addRequireAllNodesConstraint(pptc)
3 opt.addRequireAllEdgesConstraint(pptc)
4 opt.addPathDisableConstraint(pptc)
5 opt.defineVar('SwitchPower', {opt.bn(node):switchPower[node] for node in topo.nodes()})
6 opt.defineVar('LinkPower', {opt.be(u, v): linkPower[(u, v)] for u, v in topo.links()})
7 opt.setObjectiveCoeff({'SwitchPower': .75, 'LinkPower': .25}, 'min')
```

3.6 Implementation

Developer interface: We currently provide a Python API for SDN optimization that is an extended version of the interface described in Section 3.3.

Invoking solvers: We use CPLEX (via its existing Python API) as our underlying solver. This choice largely reflects our familiarity with the tool, and we could substitute CPLEX with other solvers like Gurobi. SOL offers APIs to exploit solver capabilities to use a previously computed solution and incrementally find a new solution. This approach is typically faster than starting from scratch and so is useful for faster reconfigurations. SOL also allows hard-limiting of the optimization runtime, albeit affecting the optimality of the solution.

Path generation: Path generation is an inherently parallelizable process; we simply launch separate Python processes for different traffic classes. We currently support two path selection algorithms: `random` and `shortest`. It is easy to add more algorithms as

new applications emerge.

Rule generation and control interface: We implement applications for ONOS [8] and use custom REST API to allow remote batch installation of the relevant rules. We generate the rules based on the optimization output, using network prefix splitting to implement the fractional responsibilities represented by the $x_{c,p}$ variables. This step is similar to prior work that map fractional processing and forwarding responsibilities onto network flows (e.g., [109, 50]). The only difference is that in case of flow affinity constraints, the IP prefix splitting is done on the source addresses for the "forward" traffic class, and on the destination addresses for the "reverse" traffic class. For middleboxes attached to a single switch using a single port, packet tagging can be used to avoid loops. With ONOS, we leverage *path intents* [8]: while not required, it facilitates easier integration.

Minimizing reconfiguration changes: Networks are in flux during reconfigurations with potential performance or consistency implications, and thus it is desirable to minimize unnecessary configuration changes. SOL supports constraints that bound (or minimize) the logical distance between a previous solution and the new solution. Following the optimization constraints, the rule generation logic can employ techiques described in previous work [50] to help minimize the number of flows that have to be assigned a new route.

3.7 Evaluation

In this section we show that SOL

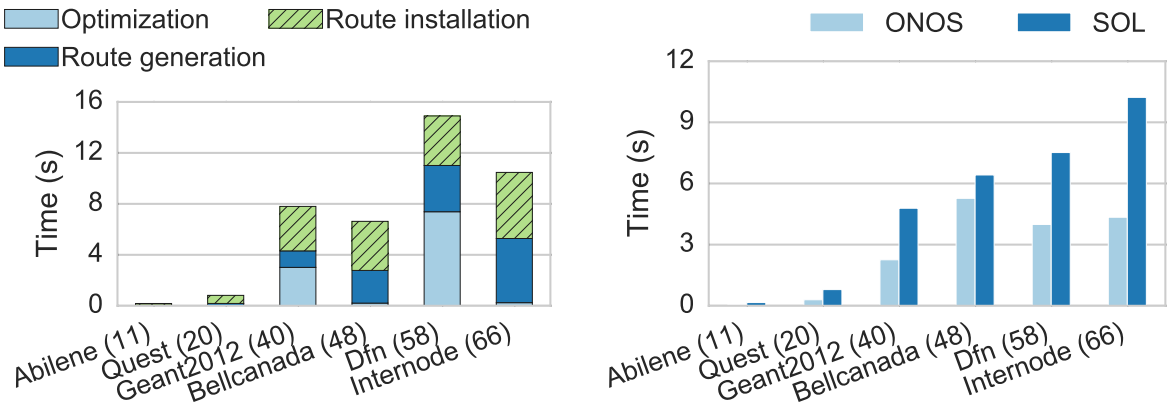
- performs well with the ONOS controller;
- computes optimal solutions for published applications order(s) of magnitude faster than their original optimizations; allows to minimize traffic churn
- is either faster or has richer functionality than state-of-the-art related work;

- significantly reduces development effort in comparison to manually coding optimization applications; and
- scales well, because it computes near-optimal solutions using few paths per traffic class.

Setup: We evaluate the effect of using SOL to implement three existing SDN applications: ElasticTree [38], SIMPLE [81], and Panopticon [61]. For each application, we implemented the original formulation presented in prior work or obtained the original source code. We refer to these as "original" formulations (and solutions). We chose topologies of various sizes from the TopologyZoo dataset [55]; when indicating a topology, we generally include the number of nodes in the topology in parentheses, e.g., "Abilene (11)" for the 11-node Abilene topology. For ElasticTree, we also constructed FatTree topologies of various sizes [2]. We synthetically generated traffic matrices using a uniform traffic matrix for the FatTree networks and a gravity-based model [86] for the TopologyZoo topologies. We used randomly sampled values from a log-normal distribution as "populations" for the gravity model. Unless otherwise specified, we used 5 paths per traffic class when running SOL. All times below refer to computation on computers with 2.4GHz cores and 128GB of RAM. For deployment benchmarks, we used the default Mininet [65] virtual machine to emulate topologies.

3.7.1 Deployment benchmarks

We setup a variety of emulated networks using Mininet and ONOS. We measured time for SOL to run the optimization for a traffic engineering goal and compute and install network routes. Figure 3.9a shows the times to perform each step. SOL exhibits low optimization and route generation times, making route installation the most time-consuming part of the configuration process. This bottleneck is caused by the number of rules that must be installed and by the controller platform. Figure 3.9b shows the time



(a) Time for SOL to configure the network using the ONOS controller for a traffic engineering application.

(b) Route generation & installation time of SOL traffic engineering app vs. ONOS all-pair shortest paths

Figure 3.9: Deployment benchmarks using the ONOS controller

to generate and install routes for a traffic engineering application using SOL, in contrast to installing shortest path routes using methods available in ONOS. The difference is insignificant, and exists due to the additional optimization time and because of the multiple paths per source-destination pair in the SOL case.

3.7.2 Optimality and scalability

Comparing to optimal: Next, we examine how well SOL's results match original solutions, which are optimal (by definition). In all cases except ElasticTree, SOL finds the optimal solution. Due to complexity of ElasticTree's optimization, SOL suffers a 10% optimality gap: the relative error in the objective value computed by SOL (i.e., relative to the true optimal objective value).

SOL solution times are at least one order of magnitude faster than solving the original formulations, and are often two or even three orders of magnitude faster. Figure 3.10 shows run times to find original solutions. The runtime was capped at 30 min (1800 s), after which the execution was aborted. Several original formulations did not complete in that time, such as SIMPLE for topologies Bellcanada and larger, and Panopticon for Ion and larger. The topologies for which original solutions could not be found are indi-

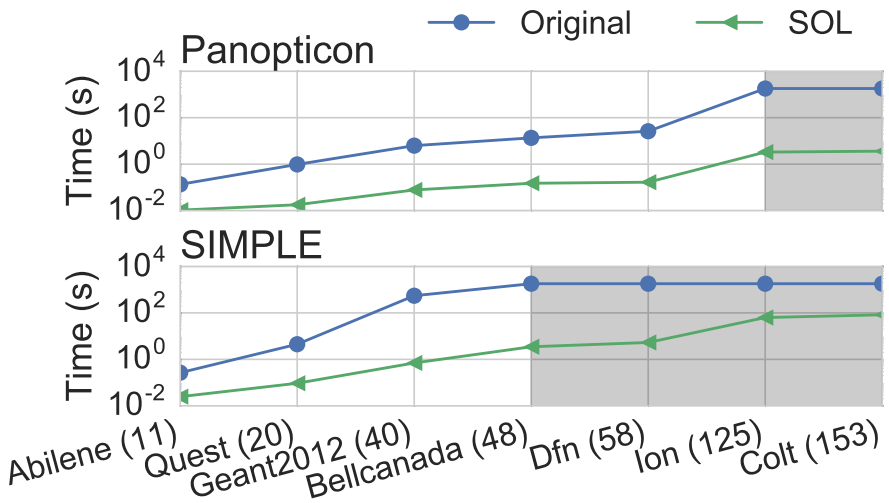
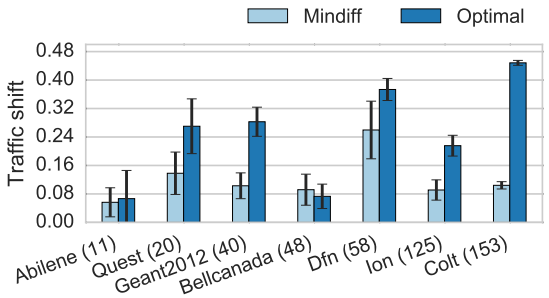


Figure 3.10: Optimization runtime of SOL and original formulations; gray regions show where original formulation could not be solved within 30 mins

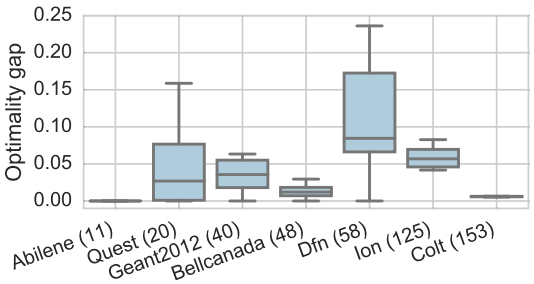
cated in the gray regions in Figure 3.10.

Comparing to specialized heuristics: We found that SOL performs fairly well even compared to specialized heuristics. Specifically, we compared the performance of SOL to the custom heuristic for SIMPLE, obtained from its authors. The runtime of SOL is comparable to that of the SIMPLE heuristic algorithm, with a performance gap of at most 3 seconds on the largest topologies we considered (up 58 nodes, namely the "Dfn" topology). We believe the benefit of not having to design custom heuristics outweighs this performance gap.

Responding to traffic changes: We explore the benefits of the reconfiguration minimization capabilities of SOL, for simplicity dubbed "mindiff." We first computed an optimal solution for a traffic engineering application; then, a random permutation of the traffic matrix triggered the re-computation with mindiff enabled. When computing the new solution, we assigned $4\times$ greater priority to the TE objective than the mindiff objective. Figure 3.11a shows that with mindiff enabled, up to an additional 35% of *total* traffic stays on its original paths across reconfigurations, versus being reassigned to new paths by the optimal solution. Naturally, SOL sacrifices some optimality in the original TE objective (shown in Figure 3.11b).



(a) Fraction of traffic reassigned to different paths with and without "mindiff"



(b) Optimality gap when using "mindiff"

Figure 3.11: Traffic shift and optimality gap when using reconfiguration minimization capabilities of SOL

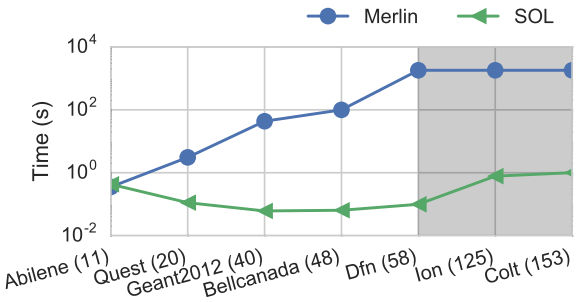
3.7.3 Comparison to Merlin and DEFO

Merlin [97] tackles problems of network resource management similar to SOL. While the goals and formulations of Merlin and SOL are quite different, we use this comparison to highlight the generality of SOL and the power of its path abstraction. Specifically, Merlin uses a more heavyweight optimization that is always an ILP and operates on a graph that is substantially larger than the physical network. We implemented the example application taken from the Merlin paper using both SOL and Merlin. Figure 3.12a shows that SOL outperforms Merlin by two or more orders of magnitude.

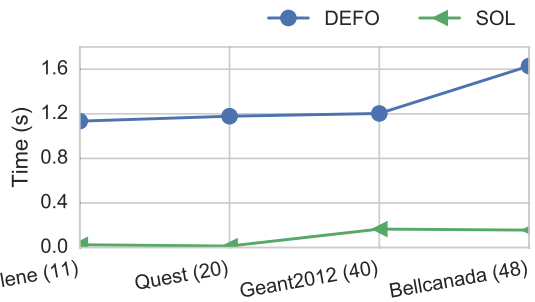
DEFO [36] is an optimization framework that aims to simplify traffic engineering [36]. We obtained the DEFO authors' implementation and compared the optimization times of DEFO and SOL on a simple traffic engineering application. DEFO and SOL exhibit comparable runtimes (see Figure 3.12b). However, DEFO lacks the ability to express more complex applications and objectives and to filter paths by arbitrary predicates.

3.7.4 Developer benefits

SOL is a much simpler framework for encoding SDN optimization tasks, versus developing custom solutions by hand. In an effort to demonstrate this simplicity some-



(a) Optimization runtimes of SOL and Merlin; gray region indicates where Merlin did not complete in 30 mins



(b) Optimization runtimes of SOL and DEFO, for a traffic engineering application

Figure 3.12: Runtime of SOL vs. state-of-the-art optimization frameworks

what quantitatively, Table 3.5 shows the number of lines of code (LOC) in our SOL implementations of various applications ("SOL lines of code"), and the ratio of the LOC of the original formulations to the LOC for our SOL implementations ("Estimated improvement"). We acknowledge that lines-of-code comparisons are inexact, but we do not know of other ways of comparing "development effort" without conducting user studies.

Name	SOL lines of code	Estimated improvement
ElasticTree	16	21.8×
Panopticon	13	25.7×
SIMPLE	21	18.6×

Table 3.5: Development effort benefits provided by SOL

We believe that the improvements in Table 3.5 are conservative. First, producing original formulations is a much more complex and delicate process than writing SOL code. We primarily attribute this difference to needing to account for CPLEX (or other solvers, e.g., [33, 68]) particulars at all; with SOL, these particulars are completely hidden from the developer. Second, SOL translates its optimization results to device configurations, whereas this functionality is not even included in our scripts for producing original formulations. Producing device configurations from original solutions would require designing an extra algorithm to map the variables in each formulation to rele-

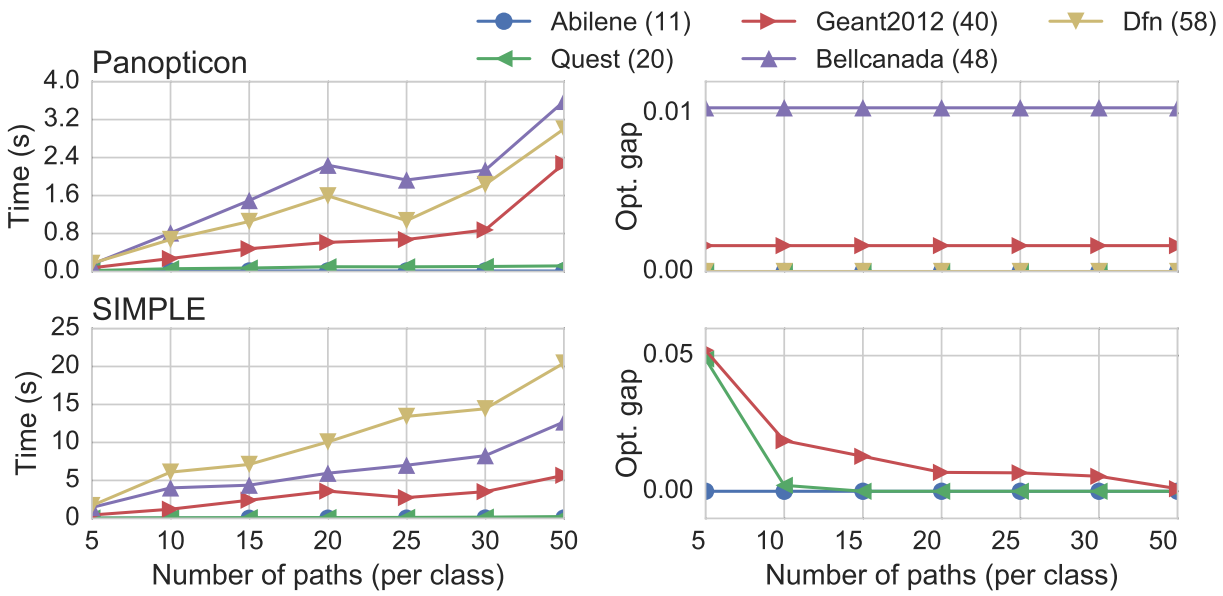


Figure 3.13: Runtime and optimality gap as function of number of paths; optimality is achieved in most cases with as few as 5 paths per class

vant device configurations.

3.7.5 Sensitivity

SOL solutions require the specification of both the number (`selectNumber`) and type (`shortest` or `random`) of paths to select per traffic class. In this section, we quantify how sensitive SOL is to these parameters.

Number of Paths: Figure 3.13 shows the SOL's runtime and optimality gap as a function of the number of paths per class for two applications: SIMPLE and Panopticon. Unsurprisingly, with a larger number of paths, SOL's runtime increases. However, this is not a significant concern, since we find optimal solutions at `selectNumber` as low as 5. These numbers are representative of all applications and topologies we have considered.

Path selection strategy: We evaluated different selection strategies across topologies and applications. Our results were consistent with our experiences more generally that most problems lend themselves to a fairly obvious path selection strategy: those with need for load balancing are likely to benefit from `random` and those that are latency-

sensitive benefit from `shortest`. If in doubt, however, both strategies can be attempted.

Path selection and generation costs: Since path selection is part of the optimization cycle, we ensure that path selection times are small, ranging from 0.1 to 3 seconds across topologies. Path selection is preceded by a path generation phase that enumerates the simple paths per class. Path generation is moderately costly for large topologies, e.g., taking <300 s for the largest presented topology, when parallelized to 60 threads. However, we emphasize that path generation can be relegated to an offline precomputation phase that is only performed once.

CHAPTER 4: Robust Composition of Multiple Optimizations

In this chapter we build upon SOL's foundation to enable efficient composition of multiple network optimization applications. One of the limitations of SOL as presented in Chapter 3 is its focus on deployment of a single application only. As we will show in Section 4.1, naive extension to support multiple applications leads to suboptimal outcomes. To provide robust composition of multiple SDN applications, we present the design (sections 4.2 and 4.3) and evaluation (Section 4.5) of Chopin, a system for composing multiple network management applications.

Today, most such resource management applications are expressed as standalone applications written in low-level optimization tools or using custom solvers. As such, these optimizations assume full control of the network and are constructed without regard for other applications, their resources, or traffic demands. Furthermore, optimizations (as presented to the solver) retain little to no semantic information about the network, resources, and intent of the optimization. Since composing optimizations at this level is impractical, previous work has avoided direct composition by taking "black box" approaches to composition: by partitioning resources between applications via topology virtualization [94, 57], or by requiring optimizations to be aware of other applications [4, 5]. The former is prone to producing suboptimal results, while the latter adds to developer burden, especially given the diverse set of applications imminent in the SDN ecosystem [92, 88].

To achieve these features, Chopin builds on top of SOL, utilizing its path-based optimization abstraction. To ensure that the resulting solution is optimal and fair, Chopin composes multiple applications into a *unified optimization*. To gain robustness while maintaining responsiveness to traffic changes, Chopin introduces an offline *coordinated*

path selection, a step that prunes the set of available paths for scalability, while maintaining efficiency in the online optimization. We further improve the tractability of coordinated path selection using traffic matrix clustering and relaxed path search.

We have implemented a Chopin prototype using Python and a Chopin service in Java for the ONOS controller. We show that using the unified optimization and coordinated path selection Chopin achieves better optimality than naive approaches by as much as 10%. Chopin also outperforms black-box composition based on voting mechanisms in optimality by a factor of 2 and runtime by as much as an order of magnitude. Finally, we analyze Chopin's scalability improvements by showing that with traffic matrix clustering and relaxed path search we achieve an order of magnitude speedup while sacrificing $\approx 1\%$ in optimality.

4.1 Background and Motivation

In this section we show example use cases of composing multiple resource-management applications and highlight the shortcomings of existing work. If two applications run their respective optimizations and attempt to implement the solution naively using an SDN controller, they run the risk of interfering with each other and overloading network resources. Hence, a composition step is necessary to facilitate proper resource sharing. We broadly divide the existing composition approaches into two classes, white-box and black-box, based on how they treat the optimizations. We highlight the limitations of both, by examining individual techniques and their shortcomings.

As a concrete example, consider Figure 4.1. Suppose the administrator desires to install two different applications: one to balance the load of web traffic on network links, and another to ensure that SSH traffic traverses a firewall and, subject to this constraint, travels minimal-latency paths. For clarity we examine only traffic traveling between nodes N_1 and N_5; it is easy to see that the optimal solution is for App_1 to use path N_1-

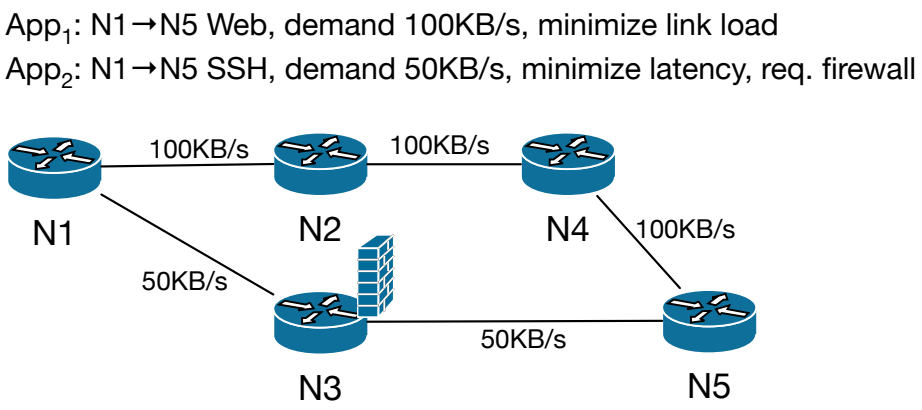


Figure 4.1: Composition scenario: two applications to be deployed and their optimizations to be applied to Web and SSH traffic (respectively).

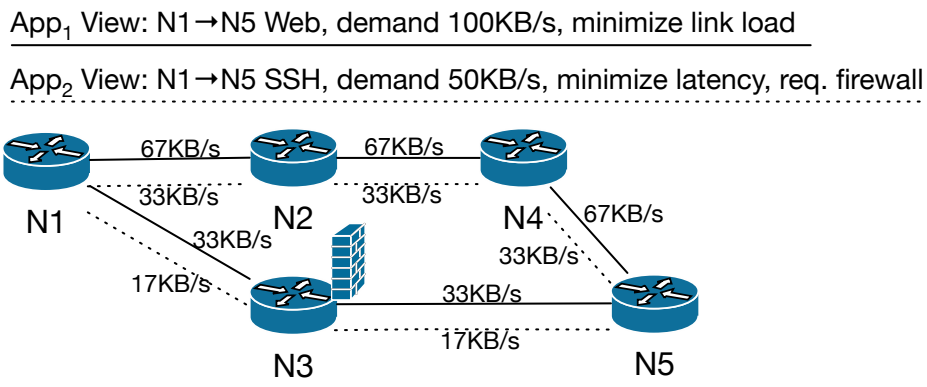


Figure 4.2: Static resource allocation provides an over-constrained view of links, resulting in a failure to enforce the firewall policy and sub-optimal latency for SSH traffic.

N_2-N_4-N_5 and for App_2 to use path N_1-N_3-N_5.

Black box composition: We classify an approach as "black box" if the optimizations are executed separately and only their inputs are modified to produce a correct result. Two common techniques are static resource allocation and ordered optimization [17].

Static allocation divides the resources, and each application is presented with a "view" of a topology based on those allocations. The allocations are computed proportionally to application priority (e.g., the amount of traffic that belongs to the application). For example, in Figure 4.2, resources are divided proportionally by traffic volume, where App_1 perceives links to have $\frac{2}{3}$ of their physical capacity, while App_2 perceives

links with $\frac{1}{3}$ of their capacity. When the optimizations are executed, due to link constraints, $\frac{1}{3}$ of SSH traffic is forced to take the path N_1-N_2-N_4-N_5, which lacks a firewall and is longer than N_1-N_3-N_5 — resulting in a failed policy enforcement and suboptimal latency.

Ordered optimization solves problems sequentially. After the first optimization is executed, the capacity of the network is adjusted by subtracting the resources consumed and the next optimization is run using the network with residual capacities. In our example, after App_1 is run, due to link load-balancing, residual capacity of the network is identical to that of App_2 view in Figure 4.2: links with capacity $17\mathrm{KB/s}$ and $33\mathrm{KB/s}$. This capacity is insufficient to correctly route the SSH traffic. While simply re-ordering the applications can alleviate this problem, for larger number of applications exploring all possible orderings to find the best solutions is impractical.

Voting schemes make improvements to the ordered optimization strategy. Examples include systems such as Corybantic [4] and Athens [5] which make applications *aware* of the other applications and allow them to vote on each others resource-management proposals to negotiate a fairer solution. We strive to solve the composition problem without imposing this awareness requirement on application developers. Additionally, as we will show in Section 4.5, voting approaches can also produce resource-inefficient results.

White box composition: We classify approaches as "white box" if the multiple applications are used to construct another, integrated optimization. Constructing a single optimization eliminates any discrete ordering of applications or explicit negotiation between them, resulting in more degrees of freedom when making routing decisions.

Manually re-designing the application optimization(s) for composition is a powerful method, but requires a non-trivial amount of effort and expertise, making this approach difficult to scale. It is also prone to errors, making it unsuitable in a production environment where applications are expected to work out-of-the-box.

Low-level composition merges the optimization encodings as they would be given

```
Minimize
  y_cpu
Subject To
 R0: x_4_0_0 + x_4_1_0 - a_4_0 = 0
 R1: x_4_0_1 + x_4_1_1 - a_4_1 = 0
 ...
 R10: x_1_0_0 + x_1_1_0 - a_1_0 = 0
 R12: - 0.00074 x_4_0_0
   - 0.00074 x_4_1_0
   - 7.581e-04 x_5_0_0
   + y_cpu_0 >= 0
 R14: y_cpu - y_cpu_0 >= 0
```

Figure 4.3: Excerpt from an optimization source, as given to the Gurobi solver. Lack of semantic information complicates composition and making decisions about resource management.

to the solver (e.g., Gurobi). However, the low-level code (example shown in Figure 4.3) has little semantic information about the topology, traffic, or network resources. For example, it is unclear what quantity the variable `x_4_0_0` represents, nor can it be easily mapped to a variable in another application's optimization. Furthermore, different types of constraints have different resolution policies. For example constraint `R1` is a flow conservation constraint and must not be arithmetically combined with others, while `R12` is a load-computation constraint and must be "added" to load-computation constraints from other applications. Naively merging low-level optimizations has no clear meaning.

High-level composition leverages network optimization frameworks such as SOL [41], Merlin [97], and Maple [108]. Composing applications written in these frameworks is viable because these frameworks retain semantic information about the optimization that could permit the automatic reconciliation of multiple applications' specifications, making it an app-agnostic and potentially robust approach.

Committing to a composition strategy leveraging high-level frameworks leaves multiple options for how to calculate the composition, however. In order to ensure that applications can be responsive to traffic changes and other events, frameworks like SOL offload a significant portion of the computation to an offline step, where the online part

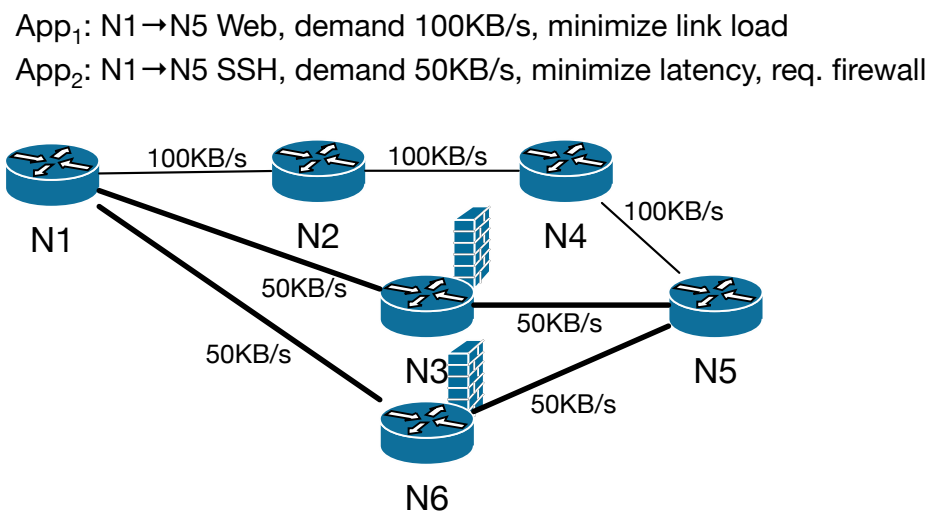


Figure 4.4: Uncoordinated path selection can result in no available solution. Both applications choose shortest paths (in bold), sufficient per application, but lack capacity in a composition scenario.

of the problem is "trimmed down" to be solvable quickly.

Failing to account for composition in the offline part of the computation can result in a final solution that is resource-inefficient. To see this, consider SOL's offline step, which involves selecting a subset of network paths over which the online optimization will be performed. For example, both SOL and Merlin support, and suggest, computing shortest paths offline and then performing online optimization only over these paths. However, in a multi-app scenario, this can lead to infeasible or resource-inefficient solutions. Consider a network in Figure 4.4: two applications are required to choose two paths, and they pick the shortest available. While sufficient for each application on its own, the total capacity of the paths is too low to carry traffic from both applications. A better solution would consider the resource demands of each application in the path selection step.

Summary: A summary of automated approaches and their features is given in Table 4.1. Black-box approaches either produce resource-inefficient results or, in the case of voting, require developers to be application-aware. (Voting also lacks responsiveness, as we show in Section 4.5.) We are aware of no low-level white-box composition tech-

	Approach	App-unaware	Fair	Responsive	Resource-efficient
Black-box	Static allocation	●	●	●	○
	Ordered	●	○	●	◔
	Voting	○	●	○	●
White-box	Low-level	○	○	○	◔
	High-level uncoordinated	●	●	●	○
	Chopin	●	●	●	●

Table 4.1: Automated composition approaches and desired features. Filled circle indicates satisfactory result. Unfilled circle indicates unsatisfactory result or unknown implementation.

niques, nor do we know how to implement composition at a low level, and so we have indicated this hypothetical alternative as unsatisfactory across the board. Existing high-level frameworks provide the semantic information needed to compose applications' objectives fairly (as we do here), but their lack of coordination during preprocessing (which is needed for responsiveness) can lead to resource-inefficiency or even infeasibility, as discussed above.

4.2 Overview

We draw a distinction between the application developer and the network operator. Developers write their optimizations by specifying the traffic and resources they desire to manage and their optimization goals. Our overarching goal is to provide fair, resource-efficient, and responsive compositions of resource management applications while maintaining APIs that do not expose composition to the developer. The operator, in contrast, configures the composition of applications (e.g., specifying what fairness metric to use) and global network constraints (such as maximum utilization of each resource). Chopin combines demands from the developers and the operator, constructs and solves a unified optimization (using a linear programming solver), and produces a

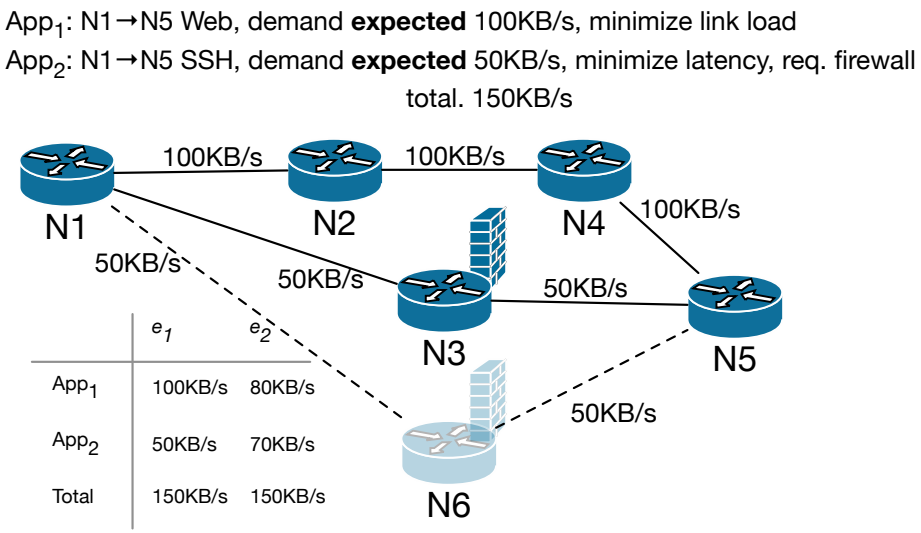


Figure 4.5: Traffic shift from time epoch e_1 to e_2 causes a policy violation if path N_1-N_6-N_5 is not chosen, despite the total volume of traffic remaining the same.

solution that can be deployed using an SDN controller.

4.2.1 High-level approach

In light of the discussion in Section 5.1, we begin by choosing the high-level, white-box approach. We adopt the path abstraction proposed in the previous chapter 3 as a general, unifying "language" upon which the unified optimization is constructed. We ensure that paths are selected in a coordinated manner, specific to the set of applications being composed, by introducing an offline, coordinated path-selection step.

Unfortunately, multiple applications exacerbate the challenges associated with offline path selection. Specifically, they add another level of complexity to the optimization, since an application's routing decisions are reflected in the network state, which in turn drives the decision making of co-located applications. Consider the network in Figure 4.5, where a traffic volume shift occurs between time epochs e_1 and e_2. Despite the total volume of traffic being the same, App_2 cannot route traffic according to policy as path N_1-N_2-N_4-N_5 does not conform to the policy and path N_1-N_6-N_5 was not selected during preprocessing. Multi-app path selection needs to account for such potential traf-

fic shifts between applications to avoid infeasibility pitfalls.

To remedy this, we update the coordinated selection to choose paths that are tolerant to traffic variations. If the traffic pattern shifts at a later time, only small flow allocation adjustments will be necessary, without the need to re-select the paths. Performing coordinated path selection is computationally difficult due to the large number of paths in the optimization the and high dimensionality of the traffic matrix. We introduce two heuristic scalability improvements (described in Section 4.3.3): traffic matrix clustering and relaxed path search, aimed at maintaining the tractability of the offline coordinated path-selection problem.

4.2.2 Workflow

Application development: The developers express their optimizations using a declarative application model. They specify the type of traffic their application manages, an objective function, and how the traffic consumes network resources. For example, an application for minimizing latency of SSH traffic (recall Section 5.1) would specify the following:

Traffic All SSH traffic

Objective Minimize latency

Constraints Route all traffic

Resource costs Bandwidth: 1KB per flow

Policy Path contains firewall

Since Chopin adopts a path-based optimization model, the declarative model can support a variety of network management applications expressible using paths (e.g., [61, 38, 81, 39, 93]). Resource consumption per flow (e.g., for modeling bandwidth usage), per path (e.g., for modeling TCAM constraints) and other types of path-based

constraints can be expressed in Chopin, though this is not the focus of our work. We refer the reader to work by Heorhiadi et al. [41].

Offline coordinated path selection: The operator collects the applications she wishes to deploy and generates network paths that conform to the applications' policies (step ❶ in Figure 4.6). We assume that policy conflicts between applications can be resolved prior to running Chopin (e.g., with a tool like PGA [79]). She specifies global resource utilization limits, and the type of fairness with which to combine the applications. She then generates a collection of traffic matrices, one per *epoch* (step ❷), which is used as an input to the path-selection process. The temporal variability of the traffic matrix across epochs dictates the robustness of the solution and can be generated from past observations or using synthetic models (e.g., [101]). The coordinated path selection (step ❸) selects a set of paths for each application, by composing the applications and choosing the paths that produce best results across the per-epoch traffic matrices. Paths are saved in the path store for use in the online deployment phase.

Online deployment: After preprocessing, the operator proceeds to deploy the applications (step ❹). Chopin constructs an optimization suited to current network demands and reflecting the desired fairness metric. When running the optimization, Chopin will retrieve appropriate paths from the path store and use them as input to the optimization. The solution is converted to network rules and can be deployed via network controller (step ❺) using techniques described in previous work [109].

4.3 Detailed Design

In this section we detail the workings of selected steps in the workflow of Figure 4.6. We focus on the steps that represent our primary technical innovations, namely step ❹ (described in Section 4.3.2) and then step ❸ (described in Section 4.3.3). We begin in Section 4.3.1 with defining terminology that facilitates these discussions.

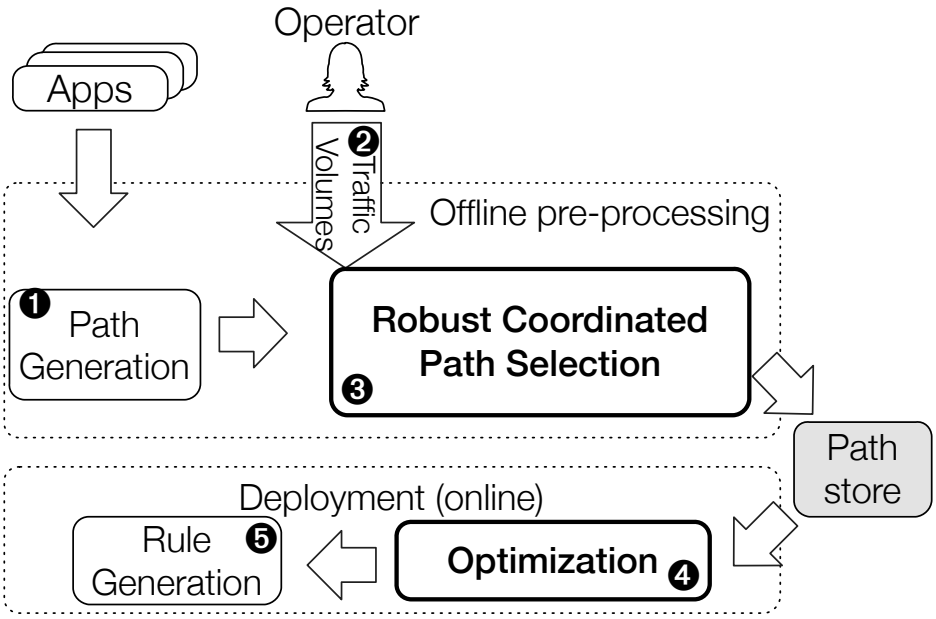


Figure 4.6: End-to-end operator workflow. The operator collects applications, generates a traffic matrix, performs robust path selection, and deploys the applications. Modules containing main Chopin contributions highlighted in bold.

4.3.1 Preliminaries

Composition is performed by combining individual elements of the applications in a systematic way. Each application declares its traffic classes and policy, which are used to generate valid network paths. Traffic is routed along these paths, consuming network resources as specified by applications' resource costs. Resource consumption, in turn, is reflected in the applications' objective functions. Precise notation follows below.

Traffic classes: A *traffic class* c is a subset of all traffic arriving at a designated ingress node $c.\mathsf{in}$, exiting at a designated egress node $c.\mathsf{out}$, and matching the specification $c.\mathsf{flowspec}$ (e.g., specified by IP 5-tuple). Each class c also has an associated volume estimate in number of flows per *time epoch* e, denoted $c.\mathsf{vol}[e]$. We assume that traffic classes do not overlap, i.e., if $\mathbb{C}$ is the set of all traffic classes, then for any $c_1, c_2 \in \mathbb{C}$, $c_1 \cap c_2 = \emptyset$. (Non-overlapping classes can be ensured by simply decomposing traffic into sufficiently

fine-granted classes, e.g., [79].) A *traffic matrix* TM_e for epoch e holds the value

$$\sum_{\substack{c \in \mathbb{C} \,:\, c.\mathsf{in} = in \\ \wedge\, c.\mathsf{out} = out}} c.\mathsf{vol}[e]$$

at location $TM_e[in, out]$.

Applications: For our purposes, an application App is specified as a set of traffic classes $App.\mathsf{classes} \subseteq \mathbb{C}$ that it manages; a set of permissible paths $App.\mathsf{paths}[in, out]$ for carrying traffic classes $c \in App.\mathsf{classes}$ such that $c.\mathsf{in} = in$ and $c.\mathsf{out} = out$; an average per-flow amount $App.\mathsf{cost}[r]$ of resource r consumed by traffic associated with this application; an objective function $App.\mathsf{obj}$ specified in terms of those resource costs and the network topology (e.g., maximizing flow, minimizing resource load); and various constraints that characterize allowable allocations of the traffic in $App.\mathsf{classes}$ to the network. The set $App.\mathsf{paths}[in, out]$ is generated in step ❶ of Figure 4.6 to contain the paths that satisfy a predicate specified by the app developer (as a function that Chopin evaluates on candidate paths, as in SOL). Each node N on path $p \in App.\mathsf{paths}[in, out]$ has a fixed resource-r *capacity* $N.\mathsf{cap}[r]$, specified in the same units as $App.\mathsf{cost}[r]$. Similarly, each link L on path $p \in App.\mathsf{paths}[in, out]$ has a fixed resource-r *capacity* $L.\mathsf{cap}[r]$.

It is convenient to define the per-flow cost for resource r associated with traffic class c to be

$$c.\mathsf{cost}[r] = \max_{App:\ c \in App.\mathsf{classes}} App.\mathsf{cost}[r]$$

and the allowable paths for c to be

$$c.\mathsf{paths} = \bigcap_{App:\ c \in App.\mathsf{classes}} App.\mathsf{paths}[c.\mathsf{in}, c.\mathsf{out}]$$

which we assume to be nonempty.

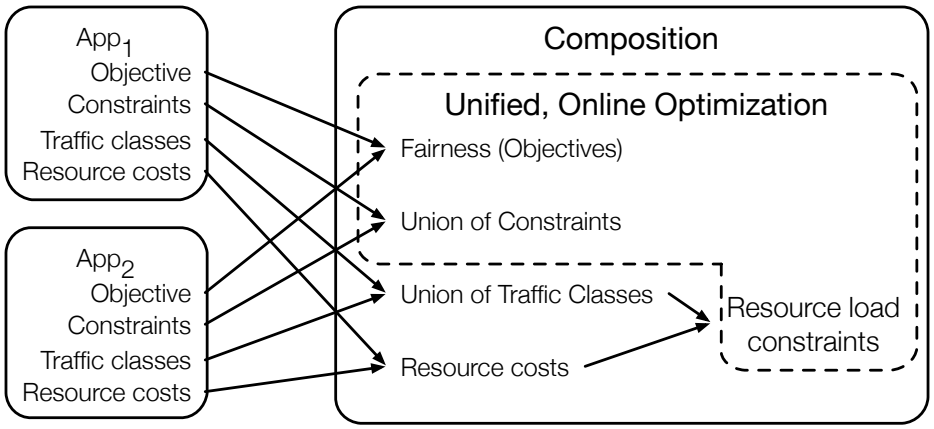


Figure 4.7: Conceptual composition of two applications: unified optimization provides resource efficiency. Fairness is applied to the application objectives. Traffic classes and resource costs are used to compute resource load constraints.

4.3.2 Online, Unified Optimization

Chopin achieves fair and resource-efficient composition by creating a single unified optimization, which allows simultaneous optimization over multiple criteria. For example, decision making for middlebox load balancing and link load balancing occurs simultaneously. Figure 4.7 provides a conceptual view for the composition process: a single online optimization is constructed from the application building blocks provided using the declarative model. A fairness measure is applied to the applications' objective functions. Application-specific constraints (such as flow conservation constraints) are combined unmodified, while resource load constraints are computed from the traffic classes and resource costs provided by each application. Figure 4.8 describes the mathematical underpinnings of the optimization. We emphasize that for this subsection, e is a constant, and $\mathbb{E}$ denotes the singleton set $\mathbb{E} = \{e\}$. This is relaxed in Section 4.3.3.

Resource load and objectives: To standardize resource consumption, all resource-r loads and objectives must be normalized to a standard range of $[0, 1]$, using the resource-r capacity $N.\mathsf{cap}[r]$ per node N and $L.\mathsf{cap}[r]$ per link L. Load on resources is expressed per traffic class using the network paths available for that traffic class. (We describe how to compute available paths in Section 4.3.3.) For example, for resources relevant

maximize

$$\mathfrak{Objective} = \sum_i w_i \times App_i.\mathsf{obj}[\mathbb{E}] \tag{4.1}$$

subject to, for all $e \in \mathbb{E}$,

$$\cdots$$

$$NLoad_N^c[r,e] = \sum_{\substack{p \in c.\mathsf{paths}: \\ N \in p}} x_{c,p,e} \frac{c.\mathsf{cost}[r] \times c.\mathsf{vol}[e]}{N.\mathsf{cap}[r]} \tag{4.2}$$

$$0 \leq x_{c,p,e} \leq 1 \tag{4.3}$$

$$NLoad_N[r,e] = \sum_{c \in \mathbb{C}} NLoad_N^c[r,e] \tag{4.4}$$

$$NLoad[r,e] = \max_N NLoad_N[r,e] \tag{4.5}$$

$$NLoad[r,e] \leq NLimit[r] \tag{4.6}$$

$$\cdots$$

$$\sum_{c \in \mathbb{C}} \sum_{p \in c.\mathsf{paths}} b_{c,p} \leq |\mathbb{C}| \times NumPaths \tag{4.7}$$

$$0 \leq x_{c,p,e} \leq b_{c,p} \tag{4.8}$$

$$b_{c,p} \in \{0,1\} \tag{4.9}$$

Figure 4.8: Core components of the linear programming formulation of the unified optimization. An example resource load computation is described in Equation 4.2–Equation 4.6, where $\mathbb{E}$ is a singleton set (containing the current epoch index) in the online optimization and $\mathbb{E} = \{1, \ldots, NumEpochs\}$ in the offline path selection. Offline path selection also adds Equation 4.7–Equation 4.9.

to nodes, we define the resource-r load $NLoad_N^c[r,e]$ induced by traffic class c on node N during epoch e by Equation 4.2, where $N \in p$ represents that node N lies on path p and where $x_{c,p,e}$ is a variable representing the fraction of flows of traffic class c routed on path p during epoch e. Then, we define the load on a resource r at node N as the sum of loads imposed by all traffic classes (Equation 4.4), and require these loads for all nodes to be at most $NLimit[r]$ (Equation 4.6), an operator-specified constant. Links are treated similarly.

Chopin supports a number of predefined objective functions to maximize. Note that because objectives are normalized, any $\min$ optimization can be converted to a $\max$ opti-

mization by using $1 - App.\mathsf{obj}$ as the new $App.\mathsf{obj}$. For example, a maximization objective that minimizes the load on node resource r is

$$App.\mathsf{obj}[\mathbb{E}] = 1 - \max_{N} \sum_{c \in App.\mathsf{classes}} NLoad_N^c[r, e] \tag{4.10}$$

The combined optimization objective for the composed applications is computed according to a specified fairness metric (see below) and maximized, subject to the constraints of all of the applications.

Fairness metrics: To ensure that no single application dominates the solution, Chopin is capable of supporting a variety of fairness metrics (also known as welfare functions). Two natural ways of enforcing fairness are at the objective level and at the resource level. We opt for applying the fairness metrics to the applications' objectives, for two reasons. First, the objectives allow for a unified way of enforcing fairness across applications. Second, mandating fair *use* of resources can result in non-linear equations with respect to $x_{c,p,e}$ variables, thus sacrificing many of the scalability benefits of linear programming optimizations.

For the objectives, most linear functions can be directly incorporated into the optimization. For example, weighted combination of objective functions results in a utilitarian solution, shown in Equation 4.1, where w_i is a weight assigned to each application.

Another common linear metric is maximizing the minimum objective (i.e., Rawlsian difference principle [34]):

$$\texttt{maximize}\ \mathfrak{Objective} = \min_i App_i.\mathsf{obj}[\mathbb{E}] \tag{4.11}$$

At the price of higher computational cost (due to their quadratic nature), the relative mean deviation and variance functions [3] can be supported. As a special case, Chopin supports proportional fairness [54] — a commonly used fairness metric. Proportional

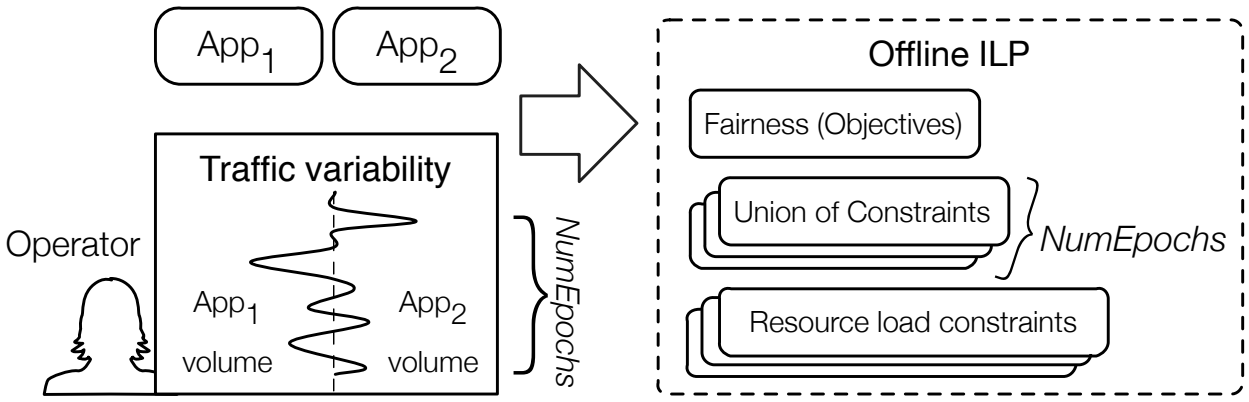


Figure 4.9: Offline coordinated path selection provides robustness by constructing a unified optimization with sets of constraints for each epoch, ensuring resource-efficiency and fairness in each epoch.

fairness is defined as:

$$\texttt{maximize}\ \mathfrak{Objective} = \sum_i \log App_i.\text{obj}[\mathbb{E}] \tag{4.12}$$

Since a $\log$ function cannot be directly incorporated into a linear program, Chopin implements a piece-wise linear approximation, based on work by Camponogara et al. [11].

4.3.3 Offline, Coordinated Path Selection

A key innovation in Chopin is selecting paths in an offline phase to ensure that the available paths are (i) rich enough to offer adequate capacity to support all applications but also (ii) few enough to permit the online, unified optimization above to be solved fast enough to ensure responsiveness on network timescales. For this purpose, we leverage the unified optimization described in Section 4.3.2 to construct a path selection integer linear program (ILP). The resulting ILP chooses paths capable of achieving resource-efficiency under traffic variations (as specified by the operator) by creating a set of constraints per traffic matrix epoch (overview shown in Figure 4.9).

Formally, this is achieved by augmenting the unified optimization with additional constraints, shown as Equation 4.7–Equation 4.9 in Figure 4.8, and broadening the opti-

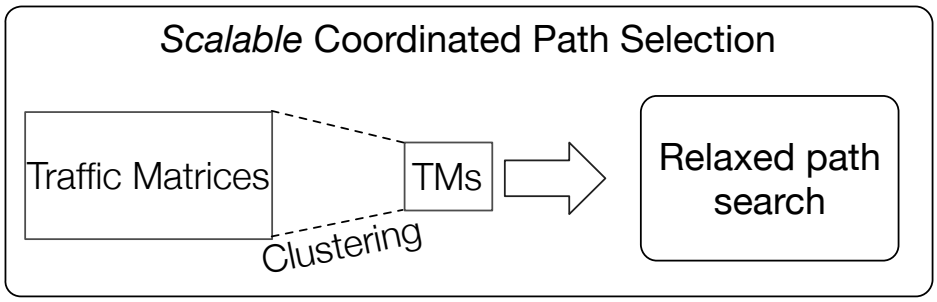


Figure 4.10: Overview of the two-step scalability improvement of coordinated path selection: traffic matrix is clustered and used as input to the offline optimization, where the ILP is replaced by a series of optimizations using relaxed path search.

mization to maximize per-application objectives *across epochs* $\mathbb{E} = \{1, \ldots, NumEpochs\}$, i.e., where

$$App.\mathsf{obj}[\mathbb{E}] = \frac{1}{NumEpochs} \sum_{e \in \mathbb{E}} App.\mathsf{obj}[\{e\}]$$

(see Equation 4.1). Equation 4.7 specifies a global limit on the number of paths used. The cap is computed using a baseline of $NumPaths$ paths per traffic class, although the final number of paths per traffic class can deviate from $NumPaths$ to achieve better results. Equation 4.8 ensures that only chosen paths are allowed to carry flow.

Tractability: The resulting ILP presents tradeoffs between resource-efficiency and scalability. A larger number of epochs provides a solution more accommodating to traffic variations and thus typically yielding better resource-efficiency, at the cost of runtime and memory needed to perform offline path selection. Similarly, an increase in the network size and so the number of paths (and thus number of binary $b_{c,p}$ variables) renders computing a true-optimal solution intractable. To address these challenges, we propose two scalability improvements, clustering and relaxed path search, shown in Figure 4.10 and described below.

Clustering speedup: To reduce the problem size, we must reduce the number of traffic matrices (epochs), yet do so without significantly reducing the variability they represent. We exploit the fact that network traffic volumes (and their synthetic models) exhibit patterns that we can preserve if we employ a *clustering* technique. Hence,

we cluster traffic classes across epochs based on their volumes. Specifically, if $\langle c_1, c_2, \ldots \rangle$ is a fixed ordering of the traffic classes, then we cluster the vectors $\{\langle c_1.\mathsf{vol}[e], c_2.\mathsf{vol}[e], c_3.\mathsf{vol}[e], \ldots \rangle\}_{e \in \mathbb{E}}$. We have experimented with a number of clustering techniques and find that k-means clustering [37] is the most scalable approach. However, the output of k-means clustering is a set of centroids that represent the *average* traffic volumes for each cluster, causing path selection to not consider the worst-case scenario. This is acceptable in some cases, however for added robustness, we also implement a hierarchical clustering algorithm with Ward's linkage [110]. Ward's algorithms allows us to group traffic volumes based on their similarity into clusters, leaving us with the possibility of applying a custom reduction function (e.g., max). Meaning, that given $NumClusters$ groups of traffic classes, we can use the worst-case volumes from each group, not the mean.

Relaxed path search: Unfortunately, even with the clustering described above, solving the ILP remains a challenge. Increases in the number of paths (due to topology size or number of traffic classes) quickly makes computing a solution impractical due to time and memory consumption. To combat this, we introduce an iterative path search approach that does not require solving an ILP.

The intuition behind iterative search is that routing with multiple traffic paths per traffic class is of limited value for topologies with sufficiently many traffic classes. This has been analytically shown for routing problems with a single resource and objective function [1, 62] and we observe similar behavior empirically for multi-application, multi-epoch optimizations. Therefore, we exploit the diminishing returns of adding more paths to the optimization by iteratively increasing number of available paths per traffic class until the objective value no longer improves. At the end, we choose a union of flow-carrying paths across epochs as the set of available paths for the online stage.

The success of iterative path search hinges on the heuristic logic responsible for choosing paths to be added in the next iteration. A natural approach is to adopt a greedy heuristic (e.g., based on path length or edge-disjointness). However, we find

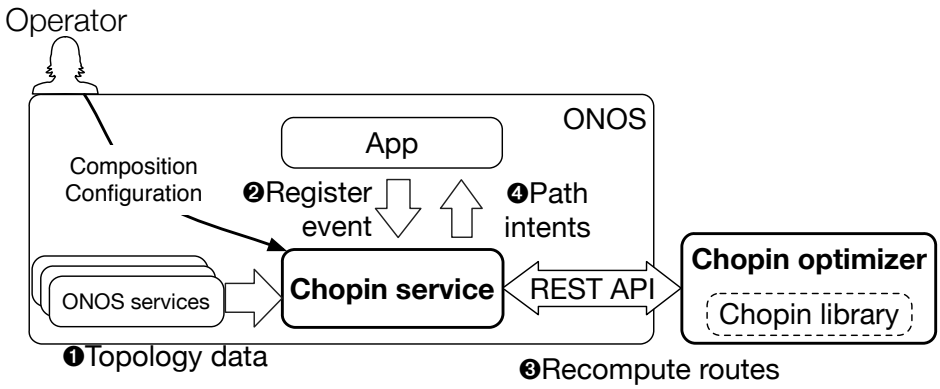


Figure 4.11: Integration between Chopin and ONOS. ONOS applications register with the Chopin service which triggers a computation using the Chopin optimizer. The solution is converted to path intents and returned to the application.

that this heuristic does not perform as expected in the multi-application scenario. Our view is that multi-objective optimization necessitates a multi-criteria heuristic. Therefore, our heuristic scores the paths based on length and "resource-richness". That is, it favors shorter paths that maximally augment the resources available to the application (based on the applications' objective functions). This biases the selection towards lower latency and link consumption (as bandwidth is a shared resource among all applications), and yet provides sufficient freedom to load balance the applications' resources of interest.

Note that a related approach is to use a more boiler-plate search strategy, e.g., simulated annealing (SA) [104]. With SA, a fixed number of different paths is evaluated in each iteration, until the objective value can no longer be improved. While valid, in our experience this approach takes more iterations to converge, due to its randomized nature and not exploiting the diminishing returns property.

4.4 Implementation

Chopin Library: The Chopin library is built using Python and can be used for composing optimizations and generating solutions as described in Section 4.3. The library

requires a linear programming solver; our prototype uses Gurobi [33].

Chopin Optimizer: Atop the library we built the Chopin optimizer, a standalone component capable of receiving composition requests from an SDN controller (or other applications). The optimizer exposes an HTTP REST API, allowing the integrations to be built in a multitude of languages and runtimes.

Integration with ONOS: For our prototype, we implement a Chopin service in the ONOS controller. Figure 4.11 depicts an architectural view of the ONOS component and its interaction with the Chopin optimizer. The Chopin service is deployed inside ONOS and receives network data (e.g., states of devices and links) from other ONOS services (step ❶). A newly deployed application registers with the service and provides its optimization requirements (step ❷). This starts the re-computation process, which utilizes the REST API to communicate with the optimizer to request the composition of all applications registered up to this point (step ❸). The Chopin service parses the solution received from the optimizer, generates appropriate intents and returns them the application(s) (step ❹). The service also allows the administrator to specify global network constraints that will act across applications. This architecture ensures that the Chopin service conforms to ONOS' event-driven nature and maintains applications' unawareness of other applications.

4.5 Evaluation

In this section, we evaluate Chopin using emulated and trace-driven simulations. Specifically, we describe the following results:

- end-to-end validation using the ONOS controller (Figure 4.12)
- resource-efficiency improvements over static allocation and voting approaches (Figure 4.13)
- resource-efficiency benefits over uncoordinated path selection (Figure 4.14)

- low impact of traffic volume clustering on final solution (Figure 4.15)
- impact of different fairness metrics have on the solution (Figure 4.16)
- runtime improvements in scalability of path selection due to the clustering and annealing techniques (Section 4.5.2).

Setup: We chose topologies of various sizes from the TopologyZoo dataset [55]; when indicating a topology, we generally include the number of nodes in the topology in parentheses, e.g., "Abilene (11)" for the 11-node Abilene topology. We also constructed FatTree topologies of various sizes [2]. We refer to these as "kX" where X denotes the arity of the FatTree, as defined in prior work. We synthetically generated traffic matrices using a modulated gravity model [86] and introduced a temporal variation between applications' traffic volumes using a Dirichlet distribution [48] across a 100 epochs. We choose the Dirichlet distribution because it generates a worst-case variability in traffic shifts between applications (e.g., 0/100% to 100/0% split between two applications) while maintaining fixed traffic volume across epochs. We also performed tests with other variability models (e.g., [101]) and observed similar results.

Unless otherwise specified, we used two canonical applications—a traffic engineering application that minimizes link load and a service chaining application that minimizes middlebox load. The applications had no overlapping traffic classes and were composed applications using a utilitarian fairness metric, with the weights equaling the fraction of traffic that belonged to the application. The service chaining application required a chain of two middleboxes – a firewall and an IDS. Times below refer to computation on computers with 2.4GHz cores and 128GB of RAM, except deployment benchmarks, where we used Mininet [65] in a virtual machine to emulate the topologies.

End-to-end validation: We setup different topologies using the Mininet emulator and ONOS controller. We deployed up to four applications (specifically, four copies of the traffic-engineering application with disjoint traffic classes) on each topology. Fig-

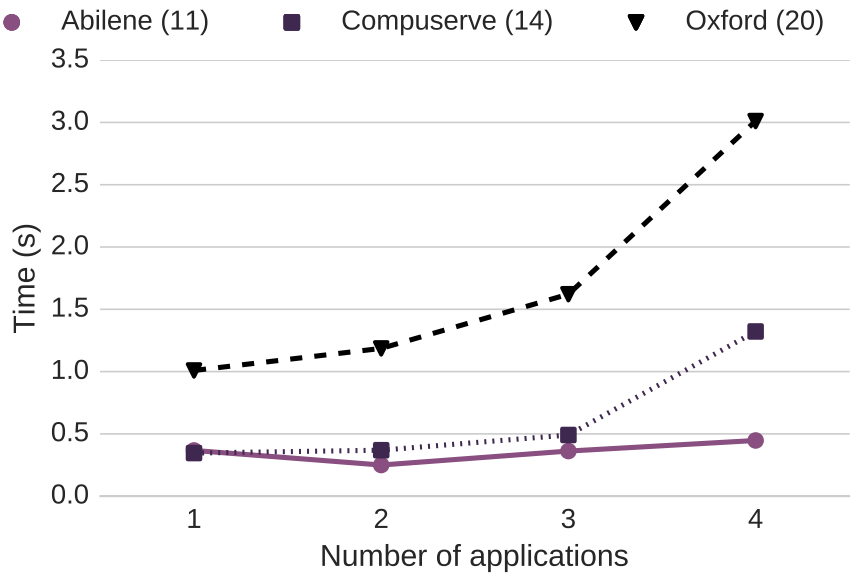


Figure 4.12: Time to deploy multiple applications as a function of number of applications. Includes time for online optimization and computing and installing ONOS path intents.

ure 4.12 shows the worst-case time to deploy the applications, meaning each new application triggered a full re-computation for all traffic classes for all applications. Even the worst-case deployment (i.e., configuring the network from scratch) maintains network time-scale responsiveness.

4.5.1 Resource-efficiency, Fairness and Responsiveness

Next, we use trace-driven simulations to evaluate the benefits of using Chopin for resource-efficiency, fairness, and responsiveness and compare it to black-box approaches. We also evaluate the potential benefits of using Chopin's coordinated path selection approach compared to prior work [41].

Resource-efficiency vs. black-box approaches: We compare Chopin to black-box approaches: naive static allocation and Athens [5] — arguably the closest practical work in this space. We created a simulator that implements the Athens voting protocols and modified our applications to be aware of all other applications present. We considered a set of paths and their flow allocations to be a "proposal" that is submitted for voting.

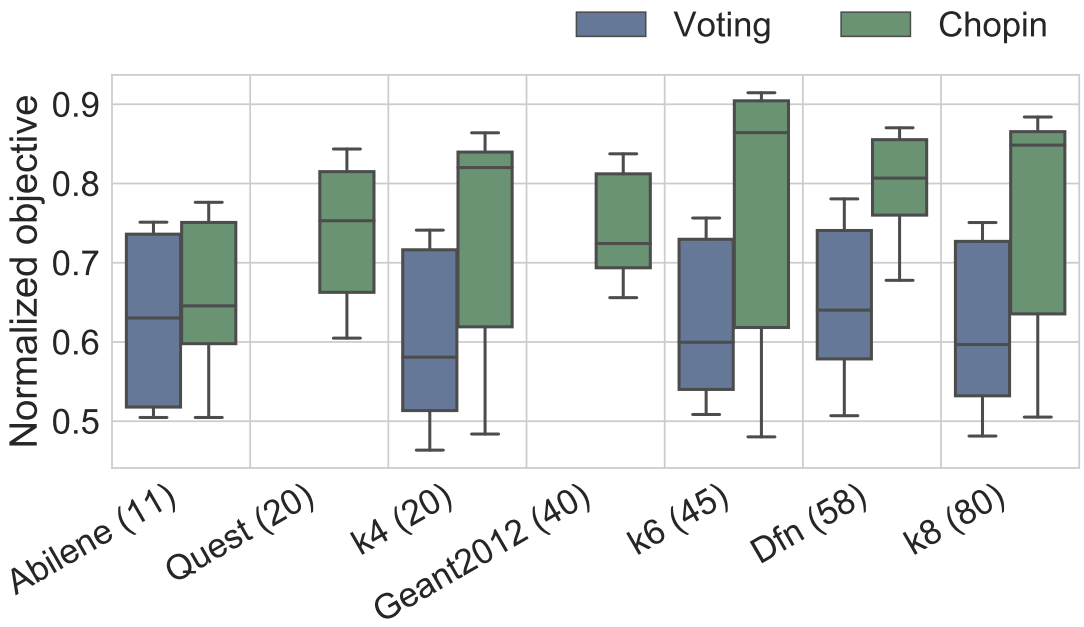


Figure 4.13: Optimality comparison between Chopin and Athens-like voting framework.

Each application was allowed to submit a proposal it considered best for its own objective function. Figure 4.13 shows the objective function (higher is better) for each topology and different composition approaches. Each box in the figure represents a composition strategy executed 100 times (i.e., across 100 epochs), with boxes covering objective values between the 25th and 75th percentiles and whiskers extending to min and max values. Chopin outperforms other approaches by as much as 60%, and naive voting fails to converge in some cases (e.g., Quest and Geant2012 topologies).

Benefits of coordinated path selection: We compared solutions obtained using shortest paths and solutions produced by Chopin. Figure 4.14 shows the relative improvement over the baseline objective computed using shortest paths. Bars represent the improvement averaged across 100 epochs, with error bars indicating the standard deviation. For all topologies, Chopin produces a more resource-efficient solution than the naive shortest-path selection strategy.

Traffic estimation sensitivity: We also evaluated the impact of traffic estimation errors on traffic matrix clustering and paths selection. To do so, we generated traffic ma-

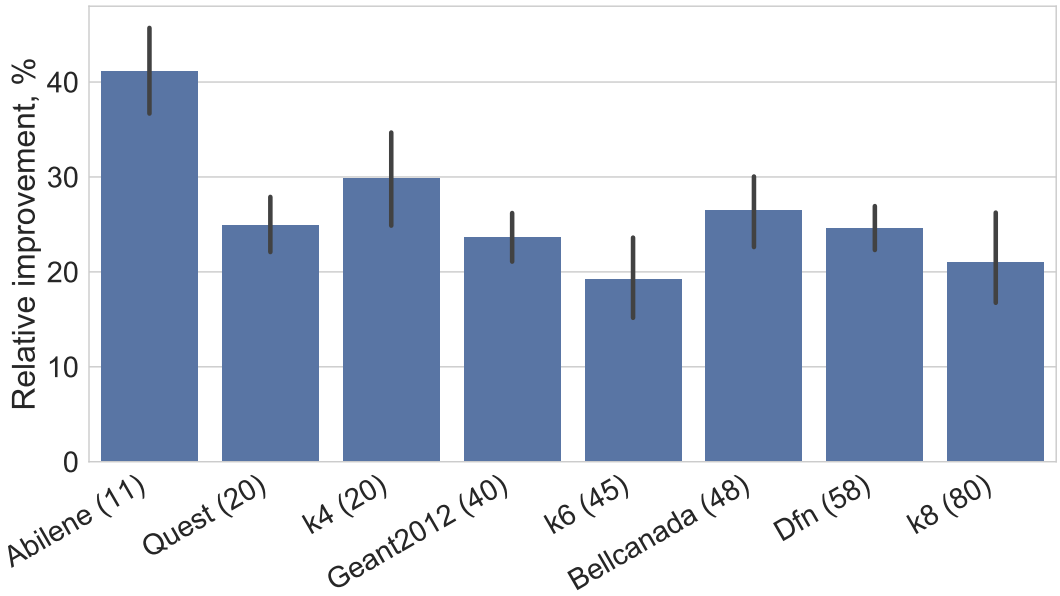


Figure 4.14: Relative improvement in objective function when using Chopin as opposed to static allocation methods with shortest paths per application. Averaged across 100 epochs.

trices for different topologies and used them to perform paths selection as described in Section 4.3. We then introduced noise to the traffic matrices by changing the volumes of each traffic class across different epochs. The noise was relative to the mean traffic volume of a traffic class and was sampled from a truncated normal distribution (with $\mu = 0$ and and $\sigma = 0.2$). Figure 4.15 depicts the relative error of using Chopin's preselected paths compared to the optimal solution (using all paths for each epoch, perfect knowledge or the TM) for topologies where optimal solution could be computed in reasonable time. The boxplots show median, 1st and 3rd quartiles, with whiskers extending to $1.5\times$ the interquartile range. This indicates that in most cases the error is quite small, $\leq 2\%$, with some exceptions.

Impact of fairness metrics on objectives: To explore the effect of different fairness metrics on the objective functions of individual applications we composed two applications using two fairness metrics: weighted and max-min fairness (see Section 4.3.2). Figure 4.16 shows objectives of two applications across different topologies and fairness

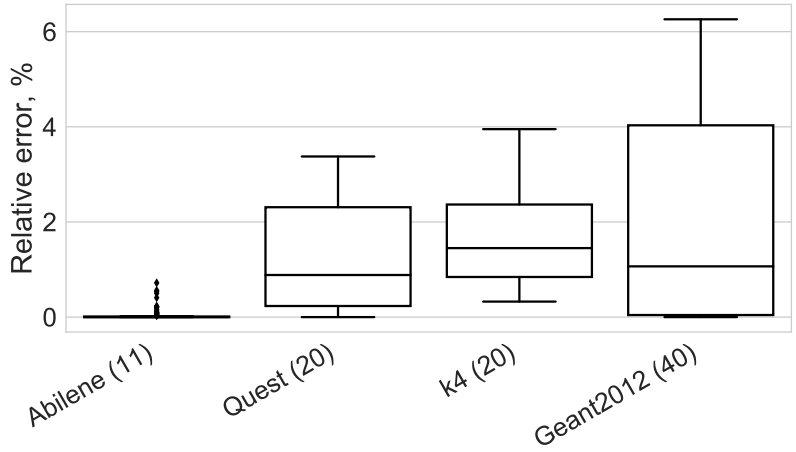


Figure 4.15: Relative error of the objective function in the presence of traffic estimation errors, compared to an optimal solution (i.e., using all paths).

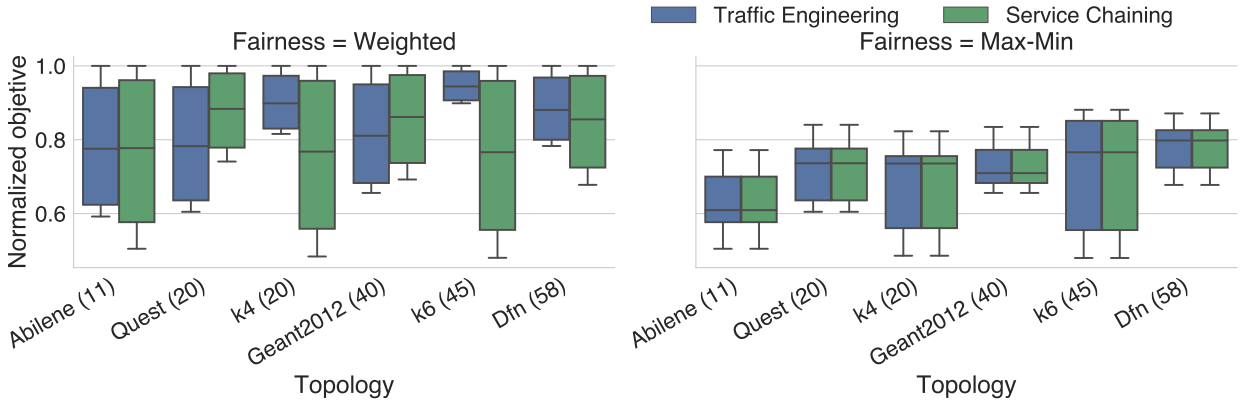


Figure 4.16: Impact of chosen fairness metric on the objective function of each application

metrics. Max-min fairness is arguably the "most fair", ensuring equal objectives but not achieving the best load balancing for either of the applications. However weighted fairness maximizes the global objective, but does so at a cost of application inequality (e.g., favoring the link load balancing on the k4, k6, and Dfn topologies). This result highlights that Chopin is flexible and gives the operator the ability to customize the solution, be it to achieve overall network resource-efficiency or fairness across applications' objectives.

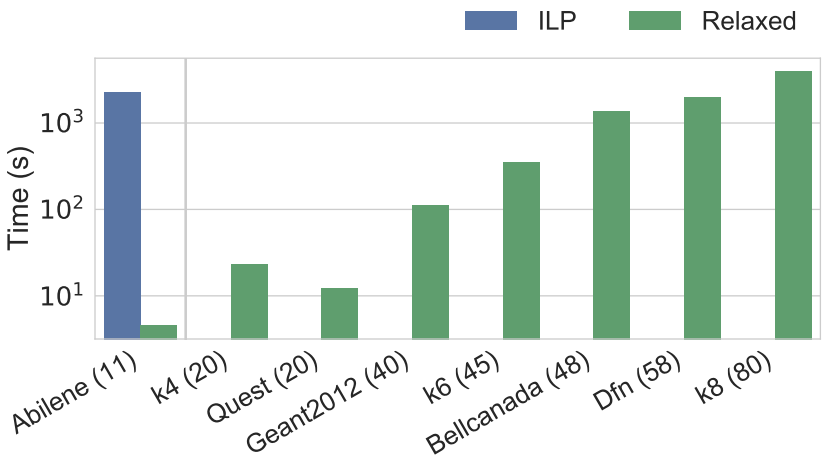


Figure 4.17: Runtime comparison of the optimal ILP path selection and relaxed selection. Relaxed paths selection is orders of magnitude faster.

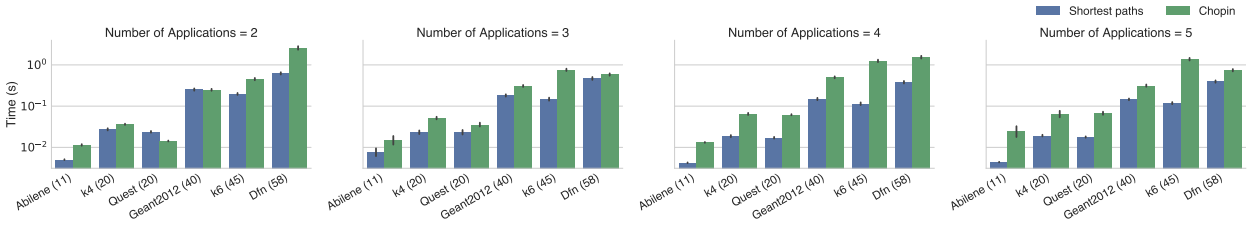


Figure 4.18: Mean time to execute a single-epoch optimization. Chopin scales similarly to using SOL in conjunction with static allocation across topologies and numbers of applications.

4.5.2 Scalability

Path selection benchmarks: To show runtime benefits of Chopin's path selection, we compare using the optimal ILP described in Section 4.3 and relaxed path selection. Not using the ILP allows orders of magnitude faster offline path selection (Figure 4.17) whereas the ILP is difficult to compute for all but the smallest topology.

Online optimization benchmarks: Finally we demonstrate that Chopin's online component is also scalable. We composed different combinations of applications (traffic engineering, service chaining, latency minimization), up to 5 total, using Chopin and static allocation with SOL single-application optimization framework. Both setups used the same number of paths, 5 per traffic class. Figure 4.18 depicts the mean time (across 100 epochs) to construct and solve the unified optimization (in case of Chopin) or series

of optimizations (in case of static allocation) across a number of topologies, for different numbers of applications. The runtimes are similar, and follow the same patterns across topologies and numbers of applications. However, Chopin achieves better optimality (recall Figure 4.14).

CHAPTER 5: Scalable Network Intrusion Prevention Using Chopin [1]

NIPS deployments face a constant battle to handle increasing volumes and processing requirements. Today, network operators have few options to tackle NIPS overload — overprovisioning, dropping traffic, or reducing fidelity of the analysis. Unfortunately, none of these options are attractive in practice. Thus, NIPS scaling has been, and continues to be, an active area of research in the intrusion detection community with several efforts on developing better hardware and algorithms (e.g., [96, 112, 102, 107]). While these efforts are valuable, they require significant capital costs and face deployment delays as networks have 3 to 5 year hardware refresh cycles.

A promising alternative to expensive and delayed hardware upgrades is to *offload* packet processing to locations with spare compute capacity. Specifically, recent work has considered two types of offloading opportunities:

- *On-path offloading* exploits the natural replication of a packet on its route to distribute processing load [90, 89].
- *Off-path offloading* utilizes dedicated clusters or cloud providers to exploit the economies of scale and elastic scaling opportunities [93, 40].

Such offloading opportunities are appealing as they flexibly use existing network hardware and provide the ability to dynamically scale the deployment. Unfortunately, current solutions either explicitly focus on passive monitoring applications such as flow monitors and NIDS [90, 40] and ignore NIPS-induced effects, e.g., on traffic volumes [89, 81, 93]. Specifically, we observe three new challenges in NIPS offloading that fall outside the scope of these prior solutions:

[1] Portions of this chapter have appeared in previously published work [39].

- **Interaction with traffic engineering:** Offloading NIPS to a datacenter means that we are effectively *rerouting* the traffic. This may affect network congestion and other traffic engineering objectives.

- **Impact on latency:** NIPS lie on the *critical forwarding path* of traffic. Delays introduced by overloaded NIPS or the additional latency induced by offloading can thus affect the *latency* for user applications.

- **Traffic volume changes:** NIPS *actively* change the traffic volume routed through the network. Thus, the load on a NIPS node is dependent on the processing actions of the upstream nodes along the packet forwarding path.

To address these challenges and deliver the benefits of offloading to NIPS deployments, we present the SNIPS (scalable NIPS) system. Furthermore, we present two ways to develop the SNIPS optimization:

1. A first-principles approach (described in Section 5.3.1) to capture the above effects and balance the tradeoffs across scalability, latency increase, and network congestion. We show that it is feasible to capture these complex requirements and effects through a linear programming (LP) formulation that is amenable to fast computation using off-the-shelf solvers.

2. A Chopin version of the SNIPS application (Section 5.3.2) that leverages the high-level APIs and composition features of Chopin. We show that it is feasible to achieve nearly identical results to that of a custom formulation without the intricate knowledge of the LP formulation.

Finally, the evaluation section (Section 5.5), presents both the results obtained from a first-principles approach and Chopin approach. For the original formulation, we show that computation takes $\leq$2 seconds for a variety of real topologies, enabling SNIPS to react in near-real-time to network dynamics. The SNIPS-Chopin approach produces

nearly identical results while also achieving faster optimization runtimes (by nearly order of magnitude) and greatly simplifying the SDN deployment.

5.1 Motivation and Challenges

We begin by briefly describing the idea of offloading for scaling *passive monitoring* solutions. Then, we highlight the challenges in using this idea for NIPS deployments that arise as a result of NIPS-specific aspects: NIPS *actively* modify the traffic volume and NIPS placement impacts the *end-to-end latency*.

5.1.1 Case for offloading

Avoiding overload is an important part of NIPS management. Some NIPS processing is computationally intensive, and under high traffic loads, CPU resources become scarce. Modern NIPS offer two options for reacting to overload: dropping packets or suspending expensive analysis modules. Neither is an attractive option. For example, Snort by default drops packets when receiving more traffic than it can process — in tests in our lab, Snort dropped up to 30% of traffic when subjected to more traffic than it had CPU to analyze — which can adversely impact end-user performance (especially for TCP traffic). Suspending analysis modules decreases detection coverage. In fact, this behavior under overload can be used to evade NIPS [75]. As such, network operators today have few choices but to provision their NIDS/NIPS to handle maximum load. For example, they can upgrade their NIPS nodes with specialized hardware accelerators (e.g., using TCAM, GPUs, or custom ASICs). While this is a valid (if expensive) option, practical management constraints restrict network appliance upgrades to a 3–5 year cycle.

A practical alternative to avoid packet drops or loss in detection coverage is by exploiting opportunities for *offloading* the processing. Specifically, prior work has exploited this idea in the context of passive monitoring in two ways: 1) *on-path* offloading to other

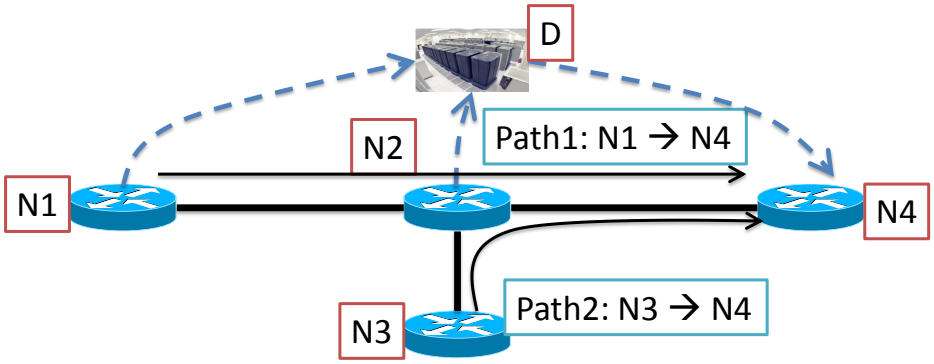


Figure 5.1: An example of the on- and off-path offloading opportunities that have been proposed in prior work for passive monitoring solutions.

monitoring nodes on the routing path [90, 89] and 2) *off-path* offloading by replicating traffic to a remote datacenter [93, 40].

To make these more concrete, consider the example network in Figure 5.1 with 4 nodes N1–N4, with traffic flowing on two end-to-end paths P1:N1 $\rightarrow$ N4 and P2:N3 $\rightarrow$ N4.[2] In a traditional deployment, each packet is processed at its *ingress* on each path: N1 monitors traffic on P1 and N3 monitors traffic on P2. An increase in the load on P1 or P2 can cause drops or detection misses

With on-path offloading, we can balance the processing load across the path (i.e., N1, N2, and N4 for P1 and N2, N3, and N4 for P2) to use spare capacity at N2 and N4 [90, 89]. This idea can be generalized to use processing capacity at off-path locations; e.g., N1 and N2 can offload some of their load to the datacenter; e.g., a NIDS cluster [102] or cloud-bursting via public clouds [93].

5.1.2 Challenges in offloading NIPS

Our goal is to extend the benefits of offloading to NIPS deployments. Unlike passive monitoring solutions, however, NIPS need to be *inline* on the forwarding path and they *actively drop* traffic. This introduces new dimensions for both on-path and off-path

[2]For brevity, in this section we use an abstract notion of a "node" that includes both the NIDS/NIPS functionality and the switching/routing function.

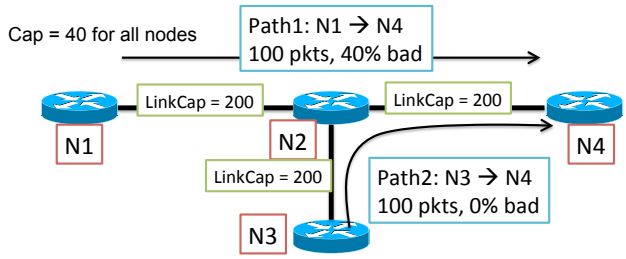


Figure 5.2: Need to model the impact of inline traffic modifications.

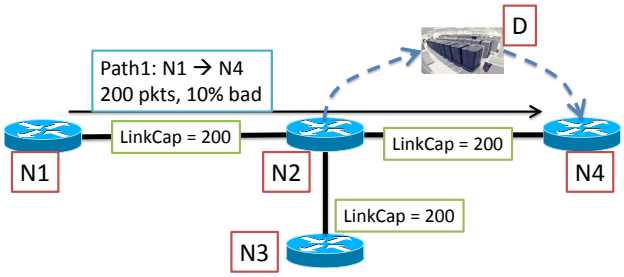


Figure 5.3: Impact of rerouting to remote locations.

offloading that falls outside the scope of the aforementioned prior work.

Suppose we have a network administrator who wants to distribute the processing load across the different nodes to: 1) operate within the provisioned capacity of each node; 2) meet traffic engineering objectives with respect to link loads (e.g., ensure that no link is loaded to more than 30%); 3) minimize increased latency due to rerouting; 4) ensures that the unwanted traffic is dropped as close to the origin as possible subject to 1), 2), and 3). We extend the example topology from earlier to highlight the key challenges that arise in meeting these goals

NIPS change traffic patterns: In Figure 5.2, each NIPS N1–N4 can process 40 packets and each link has a capacity to carry 200 packets. Suppose P1 and P2 carry a total of 100 packets and the volume of unwanted traffic on P1 is 40%; i.e., if we had no NIPS resource constraints, we would drop 40% of the traffic on P1. In order to meet the NIPS load balancing and traffic engineering objectives, we need to model the effects of the traffic being dropped by each NIPS node. If we simply use the formulations for passive monitoring systems and ignore the traffic drop rate, we may incorrectly infer that there is no feasible solution—the total offered load of 200 packets exceeds the total NIPS capacity (160). Because P1 drops 40 packets, there is actually a feasible solution.

Rerouting: Next, let us consider the impact of off-path offloading to a datacenter. Here, we see a key difference between NIDS and NIPS offloading. With NIDS, we *replicate* traffic to the datacenter D. With NIPS, however, we need to actively *reroute* the traf-

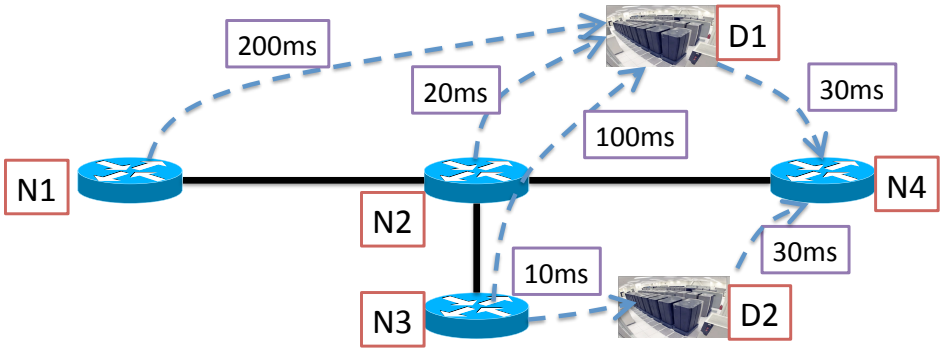


Figure 5.4: Need to carefully select offload locations in order to account for the latency for user connections

fic. In Figure 5.3, the traffic on P1 exceeds the total NIPS capacity even after accounting for the drop rate. In this case, we need to reroute a fraction of the traffic on P1 to the datacenter from N2. If we were replicating the traffic, then the load on the link N2-N4 would be unaltered. With rerouting, however, we are reducing the load on N2-N4 and introducing additional load on the links between N2 and D (and also between D and N4). This has implications for traffic engineering as we need to account for the impact of rerouting on link loads.

Latency addition due to offloading: NIDS do not actively impact user-perceived performance. By virtue of being on the critical forwarding path, however, NIPS offloading to remote locations introduces extra latency to and from the datacenter(s). In Figure 5.4, naively offloading traffic from N1 to D1 or from N3 to D1 can add hundreds of milliseconds of additional latency. Because the latency is critical for interactive and web applications (e.g., [31]), we need systematic ways to model the impact of rerouting to minimize the impact on user experience.

Conflict with early dropping: Naive offloading may also increase the *footprint* of unwanted traffic as traffic that could have been dropped may consume extra network resources before it is eventually dropped. Naturally, operators would like to minimize this impact. Let us extend the previous scenario to case where the link loads are low, and D1 and D2 have significantly higher capacity than the on-path NIPS. From a pure load per-

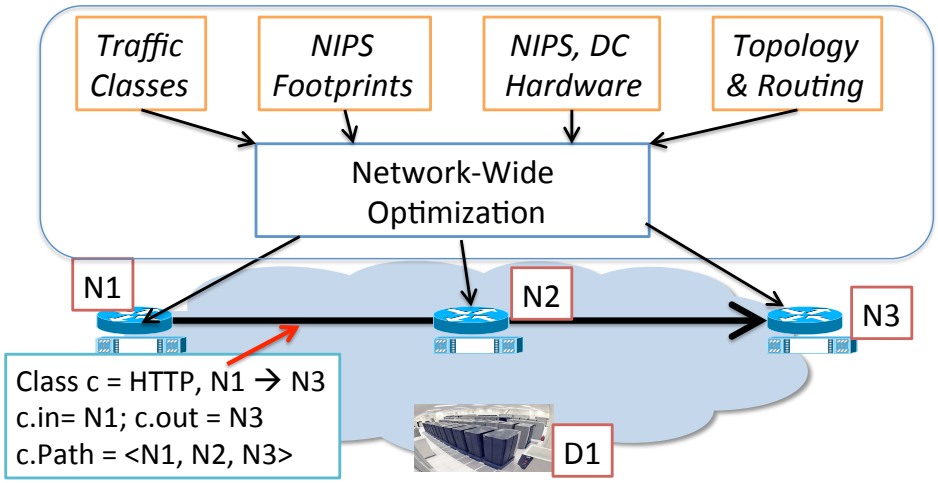


Figure 5.5: Overview of the SNIPS architecture for NIPS offloading

spective, we might want to offload most of the traffic to D1 and D2. However, this is in conflict with the goal of dropping unwanted traffic early.

Together, these examples motivate the need for a systematic way to capture NIPS-specific aspects in offloading including: (1) changes to traffic patterns due to NIPS actions; (2) accounting for the impact of rerouting in network load; (3) modeling the impact of off-path offloading on latency for users; and (4) balancing the tension between load balancing and dropping unwanted traffic early.

5.2 SNIPS System Overview

In order to address the challenges from the previous section, we present the design of the SNIPS system. Figure 5.5 shows a high-level view of the system. The design of SNIPS is general and can be applied to several contexts: enterprise networks, datacenter networks, and ISPs, though the most common use-case (e.g., as considered by past network security literature) is typically for enterprise networks.

We envision a logically centralized *controller* that manages the NIPS deployment as shown, analogous to many recent network management efforts (e.g., [13]). Network administrators specify high-level objectives such as bounds on acceptable link congestion

or user-perceived latency. The controller runs a network-wide optimization and translates these high-level goals into physical data plane configurations.

This network-wide optimization is run periodically (e.g., every 5 minutes) or triggered by routing or traffic changes to adapt to network dynamics. To this end, it uses information about the current traffic patterns and routing policies using data feeds that are routinely collected for other network management tasks [21]. Based on these inputs, the controller runs the optimization procedures (described later) to assign NIPS processing responsibilities. We begin by describing the main inputs to this NIPS controller.

- **Traffic classes:** Each *traffic class* is identified by a specific application-level port (e.g., HTTP, IRC) and network ingress and egress nodes. Each class is associated with some type of NIPS analysis that the network administrator wants to run. We use the variable c to identify a specific class. We use $c.in$ and $c.out$ to denote the ingress and egress nodes for this traffic class; in particular, we assume that a traffic class has exactly one of each. For example, in Figure 5.5 we have a class c consisting of HTTP traffic entering at $\mathsf{c}.in = \mathsf{N1}$ and exiting at $\mathsf{c}.out = \mathsf{N3}$. Let $S(c)$ and $B(c)$ denote the (expected) volume of traffic in terms of the number of sessions and bytes, respectively. We use $Match(c)$ to denote the expected rate of unwanted traffic (which, for simplicity, we assume to be the same in sessions or bytes) on the class c, which can be estimated from summary statistics exported by the NIPS.

- **Topology and Routing:** The path traversed by traffic in a given class (before any rerouting due to offloading) is denoted by $c.path$. For clarity, we assume that the routing in the network is symmetric; i.e., the path $c.path = Path(c.in, c.out)$ is identical to the reverse of the path $Path(c.out, c.in)$. In our example, $\mathsf{c}.path = \langle \mathsf{N1}, \mathsf{N2}, \mathsf{N3} \rangle$. Our framework could be generalized to incorporate asymmetric routing as well. For simplicity, we restrict the presentation of our framework to assume symmetric routing.

We use the notation $N_j \in Path(src, dst)$ to denote that the NIPS node N_j is *on the routing path* between the source node src and the destination node dst. In our example, this means that $\mathsf{N1}, \mathsf{N2}, \mathsf{N3} \in Path(\mathsf{N1}, \mathsf{N3})$. Note that some nodes (e.g., a dedicated cluster such as D1 in Figure 5.5) are off-path; i.e., these do not observe traffic unless we explicitly re-route traffic to them. Similarly, we use the notation $l \in Path(src, dst)$ to denote that the link l is on the path $Path(src, dst)$. We use $|Path(src, dst)|$ to denote the latency along a path $Path(src, dst)$. While our framework is agnostic to the units in which latency is measured, we choose hop-count for simplicity.

- **Resource footprints:** Each class c may be subject to different types of NIPS analysis. For example, HTTP sessions may be analyzed by a payload signature engine and through web firewall rules. We model the cost of running the NIPS for each class on a specific *resource* r (e.g., CPU cycles, memory) in terms of the expected per-session resource footprint F_c^r, in units suitable for that resource (F_c^r for *Footprint* on r). These values can be obtained either via NIPS vendors' datasheets or estimated using offline benchmarks [19].

- **Hardware capabilities:** Each NIPS hardware device N_j is characterized by its resource capacity Cap_j^r in units suitable for the resource r. In the general case, we assume that hardware capabilities may be different because of upgraded hardware running alongside legacy equipment.

We observe that each of these inputs (or the instrumentation required to obtain them) is already available in most network management systems. For instance, most centralized network management systems today keep a network information base (NIB) that has the current topology, traffic patterns, and routing policies [21]. Similarly, the hardware capabilities and resource footprints of the different traffic classes can be obtained with simple *offline* benchmarking tools [19]. Note that our assumption on the availability

of these inputs is in line with existing work in the network management literature. The only additional input that SNIPS needs is $Match(c)$, which is the expected drop rate for the NIPS functions. These can be estimated using historical logs reported by the NIPS; anecdotal evidence from network administrators suggests that the match rates are typically quite stable [80]. Furthermore, SNIPS can provide significant benefits even with coarse estimates. In this respect, our guiding principle is to err on the conservative side; e.g., we prefer to overestimate resource footprints and underestimate the drop rates.

Note that SNIPS does not compromise the security of the network relative to a traditional ingress-based NIPS deployment. That is, any malicious traffic that would be dropped by an ingress NIPS will also be dropped in SNIPS; this drop may simply occur elsewhere in the network as we will see.

Given this setup, we describe the optimization formulations for balancing the trade-off between the load on the NIPS nodes and the latency and congestion introduced by offloading.

5.3 SNIPS Optimization

We first describe how to construct a custom SNIPS optimization using linear programming, followed by a Chopin application to achieve the same functionality.

5.3.1 First-principles SNIPS Optimization

Given the inputs from the previous section, our goal is to optimally distribute the NIPS processing through the network. To this end, we present a linear programming (LP) formulation. While LP-based solutions are commonly used in traffic engineering [90, 23], NIPS introduce new dimensions that make this model significantly different and more challenging compared to prior work [90, 40]. Specifically, rerouting and active manipulation make it challenging to systematically capture the effective link and

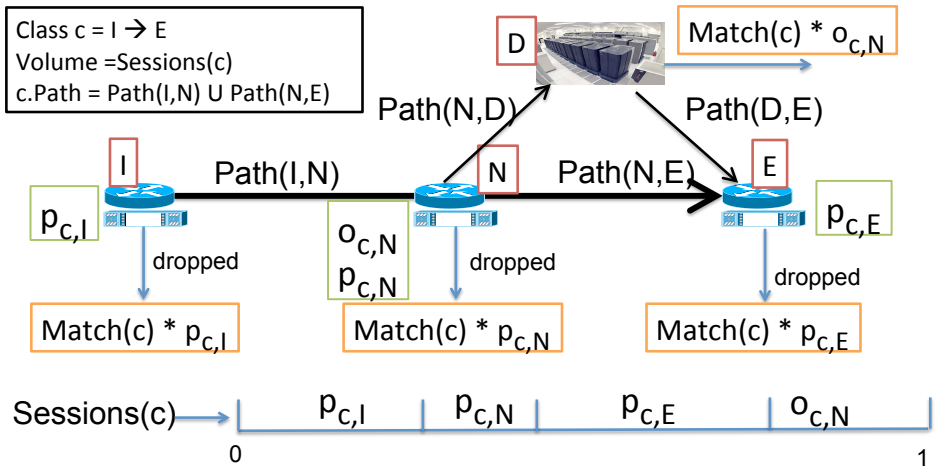


Figure 5.6: An example to highlight the key concepts in our formulation and show modeling of the additional latency due to rerouting.

NIPS loads using the optimization models from prior work, and thus we need a first-principles approach to model the NIPS-specific aspects.

Our formulation introduces decision variables that capture the notion of *processing and offloading fractions*. These variables, defined for each node along a routing path, control the number of flows processed at each node. Let $p_{c,j}$ denote the fraction of traffic on class c that the router N_j processes locally and let $o_{c,j,d}$ denote the fraction of traffic on class c that the NIPS node N_j offloads to the datacenter d. For clarity of presentation, we assume there is a single datacenter d and thus drop the d subscript; it is easy to generalize this formulation to multiple datacenters, though we omit the details here due to space considerations.

Intuitively, we can imagine the set of traffic sessions belonging to class c entering the network (i.e., before any drops or rerouting) as being divided into non-overlapping buckets, e.g., either using hashing or dividing the traffic across prefix ranges [109, 90, 102]. The fractions $p_{c,j}$ and $o_{c,j}$ represent the length of these buckets as shown in Figure 5.6.

Figure 4.8 shows the optimization framework we use to systematically balance the trade-offs involved in NIPS offloading. We illustrate the key aspects of this formulation

Minimize: $(1-\alpha-\beta) \times NLdCost + \alpha \times HopsUnwanted + \beta \times LatencyInc$, subject to:

$$\forall c : \sum_{N_j \in c.path} p_{c,j} + o_{c,j} = 1 \quad (5.1)$$

$$\forall r, j : NLd_{j,r} = \sum_{c:N_j \in c.path} p_{c,j} \times S(c) \times F_c^r \quad (5.2)$$

$$\forall r, j : NLd_{j,r} \leq Cap_j^r \quad (5.3)$$

$$\forall r : \sum_{c} \sum_{N_j \in c.path} o_{c,j} \times S(c) \times F_c^r \leq DCap^r \quad (5.4)$$

$$\forall r, j : NLdCost \geq NLd_{j,r} \quad (5.5)$$

$$\forall l : BG_l = \sum_{c:l \in c.path} B(c) \quad (5.6)$$

$$\forall l : LLd_l \leq MaxLLd \times LCap_l \quad (5.7)$$

$$LatencyInc = \sum_{c} \sum_{N_j \in c.path} o_{c,j} \times S(c) \times \left(\begin{array}{r} |Path(N_j, d)| + |Path(d, c.out)| \\ -|Path(N_j, c.out)| \end{array} \right) \quad (5.8)$$

$$\begin{aligned} HopsUnwanted = & \sum_{c} \sum_{N_j \in c.path} p_{c,j} \times S(c) \times Match(c) \times |Path(c.in, N_j)| \\ & + \sum_{c} \sum_{N_j \in c.path} o_{c,j} \times S(c) \times Match(c) \times \left(\begin{array}{r} |Path(c.in, N_j)| \\ +|Path(N_j, d)| \end{array} \right) \end{aligned} \quad (5.9)$$

$$\begin{aligned} \forall l : LLd_l = & BG_l \quad + \sum_{c} \sum_{\substack{N_j : N_j \in c.path \\ \wedge\ l \in Path(N_j, d)}} o_{c,j} \times B(c) \\ & + \sum_{c:l \in Path(d, c.out)} \sum_{N_j \in c.path} o_{c,j} \times B(c) \times (1 - Match(c)) \\ & - \sum_{c} \sum_{N_j \prec_c l} o_{c,j} \times B(c) - \sum_{c} \sum_{N_j \prec_c l} p_{c,j} \times B(c) \times Match(c) \end{aligned} \quad (5.10)$$

Figure 5.7: Formulation for balancing the scaling, latency, and footprint of unwanted traffic in network-wide NIPS offloading.

using the example topology in Figure 5.6 with a single class c of traffic flowing between the ingress I and egress E. This toy topology has a single data center D and traffic being offloaded to D from a given node N.

Goals: As discussed earlier, NIPS offloading introduces several new dimensions: (1) ensure that the NIPS hardware is not overloaded; (2) keep all the links at reasonable loads to avoid unnecessary network congestion; (3) add minimal amount of extra latency for user applications; and (4) minimize the network footprint of unwanted traffic. Of these, we model (2) as a *constraint* and model the remaining factors as a multi-

criterion objective.[3]

Note that these objectives could possibly be in conflict and thus we need to systematically model the trade-offs between these objectives. For instance, if are not worried about the latency impact, then the optimal solution is to always offload traffic to the datacenter. To this end, we model our objective function as a weighted combination of factors (1), (3), and (4). Our goal here is to devise a general framework rather than mandate specific values of the weights. We discuss some natural guidelines for selecting these weights in Section 5.5.

Coverage (Equation 5.1): Given the process and offload variables, we need to ensure that every session in each class is processed somewhere in the network. Equation 5.1 captures this coverage requirement and ensures that for each class c the traffic is analyzed by some NIPS on that path or offloaded to the datacenter. In our example, this means that $p_{\mathsf{c,I}}$, $p_{\mathsf{c,N}}$, $p_{\mathsf{c,E}}$, and $o_{\mathsf{c,N}}$ should sum up to 1.

Resource Load (Equation 5.2–Equation 5.5): Recall that F_c^r is the per-session processing cost of running the NIPS analysis for traffic on class c. Given these values, we model the load on a node as the product of the processing fraction $p_{c,j}$, the traffic volume along these classes and the resource footprint F_c^r. That is, the load on node N_j due to traffic processed on c is $S(c) \times p_{c,j} \times F_c^r$. Since our goal is to have all nodes operating within their capacity, we add the constraint in Equation 5.3 to ensure that no node exceeds the provisioned capacity. The load on the datacenter depends on the total traffic offloaded to it, which is determined by the $o_{c,j}$ values, i.e., $o_{\mathsf{c,N}}$ in our example of Figure 5.6. Again, this must be less than the capacity of the datacenter, as shown in Equation 5.4. Furthermore, since we want to minimize resource load, Equation 5.5 captures the maximum resource consumption across all nodes (except the datacenter).[4]

[3]The choice of modeling some requirement as a strict constraint vs. objective may differ across deployments; as such, our framework is quite flexible. We use strict bounds on the link loads to avoid congestion.

[4]At first glance, it may appear that this processing load model does not account for reduction in processing load due to traffic being dropped upstream. Recall, however, that $p_{c,j}$ and $o_{c,j}$ are defined as fractions of original traffic that enters the network. Thus, traffic dropped upstream will not impact the

Latency penalty due to rerouting (Equation 5.8): Offloading means that traffic takes a detour from its normal path to the datacenter (and then to the egress). Thus, we need to compute the latency penalty caused by such rerouting. For any given node N_j, the original path $c.path$ can be treated as the logical concatenation of the path $Path(in, N_j)$ from ingress in to node N_j and the path $Path(N_j, out)$ from N_j to the egress out. When we offload to the datacenter, the additional cost is the latency from this node to the datacenter and datacenter to the egress. However, since this traffic does not traverse the path from N_j to the egress, we can subtract out that latency. In Figure 5.6, the original latency is $|Path(\mathsf{I}, \mathsf{N})| + |Path(\mathsf{N}, \mathsf{E})|$; the offloaded traffic incurs a latency of $|Path(\mathsf{I}, \mathsf{N})| + |Path(\mathsf{N}, \mathsf{D})| + |Path(\mathsf{D}, \mathsf{E})|$ which results in a latency increase of $|Path(\mathsf{N}, \mathsf{D})| + |Path(\mathsf{D}, \mathsf{E})| - |Path(\mathsf{N}, \mathsf{E})|$. This models the latency increase for a given class; the accumulated latency across all traffic is simply the sum over all classes (Equation 5.8).

Unwanted footprint (Equation 5.9): Ideally, we want to drop unwanted traffic as early as possible to avoid unnecessarily carrying such traffic. To capture this, we compute the total "network footprint" occupied by unwanted traffic. Recall that the amount of unwanted traffic on class c is $Match(c) \times B(c)$. If the traffic is processed locally at router N_j, then the network distance traversed by the unwanted traffic is simply $|Path(c.in, N_j)|$. If the traffic is offloaded to the datacenter by N_j, however, then the network footprint incurred will be $|Path(c.in, N_j)| + |Path(N_j, d)|$. Given a reasonable bucketing function, we can assume that unwanted traffic will get mapped uniformly across the different logical buckets corresponding to the process and offload variables. In our example, the volume of unwanted traffic dropped at N is simply $Match(\mathsf{c}) \times B(\mathsf{c}) \times p_{\mathsf{c},\mathsf{N}}$. Given this, we can compute the network footprint of the unwanted traffic as a combination of the locally processed and offloaded fractions as shown in Equation 5.9.

Due to processing coverage constraint, we can guarantee that SNIPS provides the

processing load model.

same the security functionality as provided by a traditional ingress NIPS deployment. That is, any malicious traffic that should be dropped will be dropped *somewhere* under SNIPS. (And conversely, no legitimate traffic will be dropped.)

Link Load (Equation 5.6, Equation 5.7, Equation 5.10): Last, we come to the trickiest part of the formulation — modeling the link loads. To model the link load, we start by considering the baseline volume that a link will see if there were no traffic being dropped and if there were no offloading. This is the background traffic that is normally being routed. Starting with this baseline, we notice that NIPS offloading introduces both positive and negative components to link loads.

First, rerouting can induce additional load on a given link if it lies on a path between a router and the datacenter; either on the forward path to the datacenter or the return path from the data center to the egress. These are the additional positive contributions shown in Equation 5.10. In our example, any link that lies on the path $Path(\mathsf{N}, \mathsf{D})$ will see additional load proportional to the offload value $o_{\mathsf{c},\mathsf{N}}$. Similarly, any link on the path from the data center will see additional induced load proportional to $o_{\mathsf{c},\mathsf{N}} \times (1 - Match(\mathsf{c}))$ because some of the traffic will be dropped.

NIPS actions and offloading can also reduce the load on some links. In our example, the load on the link N-E is lower because some of the traffic has been offloaded from N; this is captured by the first negative term in Equation 5.10. There is also some traffic dropped by the NIPS processing at the upstream nodes. That is, the load on link N-E will be lowered by an amount proportional to $(p_{\mathsf{c},\mathsf{I}} + p_{\mathsf{c},\mathsf{N}}) \times Match(\mathsf{c})$. We capture this effect with the second negative term in Equation 5.10 where we use the notation $N_j \prec_c l$ to capture routers that are upstream of l along the path $c.path$.

Together, we have the link load on each link expressed as a combination of three factors: (1) baseline background load; (2) new positive contributions if the link lies on the path to/from the datacenter, and (3) negative contributions due to traffic dropped upstream and traffic being rerouted to the data center. Our constraint is to ensure that no

link is overloaded beyond a certain fraction of its capacity; this is a typical traffic engineering goal to ensure that there is only a moderate level of congestion at any time.

Solution: Note that our objective function and all the constraints are *linear* functions of the decision variables. Thus, we can leverage commodity linear programming (LP) solvers such as CPLEX to efficiently solve this constrained optimization problem. In Section 5.4 we discuss how we map the output of the optimization (fractional $p_{c,j}$ and $o_{c,j}$ assignments) into *data plane* configurations to load balance and offload the traffic.

We note that this basic formulation can be extended in many ways. For instance, administrators may want different types of guarantees on NIPS failures: fail-open (i.e., allow some bad traffic), fail-safe (i.e., no false negatives but allow some benign traffic to be dropped), or strictly correct. SNIPS can be extended to support such policies; e.g., modeling redundant NIPS or setting up forwarding rules to allow traffic to pass through.

5.3.2 SNIPS Optimization using Chopin

The previous section exemplifies the non-trival effort required to correctly model SNIPS requirements using an optimization. We now show how SNIPS can be implemented using Chopin. When describing the Chopin version we focus on the high-level goals of the system and demonstrate how they can be easily expressed in Chopin.[5] We highlight that with Chopin many intricate details of the original-SNIPS linear program become unnecessary or abstracted away. We construct the updated SNIPS application by diving it into three major components: load balancing, latency, and unwanted footprint minimization. Then we utilize the composition capabilities of Chopin to find the optimal solution.

[5]The API used in this chapter uses different naming conventions from that described in Chapter 3 due to evolution of the codebase. However it is functionally equivalent.

5.3.2.1 Predicates and paths

First, we must enforce the required IPS policy by constructing a path predicate that marks the path to be valid if it passes through an IPS. Alternatively, we express this as the following statement: *at least one node (middlebox) in the path is capable of IPS functionality*. In code, this is expressed as follows:

```
1  def ips_predicate(path, topology):
2    return any(['ips' in topo.get_service_types(n) for n in path.mboxes()])
```

5.3.2.2 Load balancing and latency calculations

We divide SNIPS into three separate applications, and utilize the composition capabilities of Chopin to compose them (recall Chapter 4). First, we define the load-balancing application as follows (appropriately provisioned topology and paths per traffic class (pptc) are assumed):

```
1 b = AppBuilder()
2 loadapp = b.name('snips_load').pptc(pptc) ↲
3   .add_constr(Constraint.ROUTE_ALL) ↲
4   .add_constr(Constraint.MBOX_AFFINITY, tc_pairs) ↲
5   .objective(Objective.MIN_NODE_LOAD, 'cpu') ↲
6   .add_resource('cpu', cpu_func, NODES) ↲
7   .add_resource('bandwidth', bw_func, LINKS).build()
```

This ensures that all traffic is routed (Constraint.ROUTE_ALL), defines the cost for how CPU and bandwidth is consumed and the objective function of minimizing the CPU load. Additionally, we enforce middlebox processing affinity for traffic class pairs `tc_pairs` (recall the stateful processing of bi-directional sessions challenge from Section 5.4.1.1).

The key to the correct optimization is modeling the anticipated malicious traffic drop

and its effect on bandwidth consumption. This logic is expressed in the `bw_func`, which is defined as follows:

```
def bw_load_with_drop(tc, path, link, drop_rates, cost):
    u, v = link
    nodes = list(path.nodes())
    # malicious traffic drop point
    mbox = path.mboxes()[0]
    # load after the middlebox (and thus drop)
    if nodes.index(u) >= nodes.index(mbox):
        return tc.volFlows * (1 - drop_rates[mbox]) * cost
    else: # load before the middlebox (no drop)
        return tc.volFlows * cost

# Curry the function with appropriate drop rate and cost parameters
bw_func = functools.partial(bw_load_with_drop,
  drop_rates=defaultdict(lambda: .1), cost=1)
```

Note that `bw_load_with_drop` contains the logic previously captured by Equation 5.6, Equation 5.7, and Equation 5.10. Furthermore, it does so in a more straightforward manner, where for any given network path, the link load imposed by the traffic can be split into two cases: before the drop and after the drop.

Since all the applications operate on the same traffic classes, our subsequent latency and unwanted applications can be even simpler, and only contain objective functions with appropriate cost functions. Latency application will minimize the overall latency:

```
b = AppBuilder()
b.name('snips_latency').pptc(pptc).objective(Objective.MIN_LATENCY)
latencyapp = b.build()
```

The unwanted traffic footprint is a variant of latency computation with a custom cost

function, where the cost of a path is equal to the number of hops until the unwanted traffic is processed (and dropped), which in our case, is the location of the first IPS middlebox in the path:

```
1 def unwanted_func(path, malicious_fraction=.1):
2     ips = path.mboxes()[0]
3     return list(path.nodes()).index(ips) * malicious_fraction

5 b = AppBuilder()
6 b.name('snips_unwanted').pptc(pptc).objective(Objective.MIN_LATENCY,
      cost_func=unwanted_func)
7 unwanted = b.build()
```

The resulting applications are composed using the Chopin's capabilities to produce the solution. Detailed quantitative comparison is presented in Section 5.5.

5.3.2.3 Advantages of Chopin optimization

First, the Chopin optimization removes an implicit assumption that each traffic class has a single routing path. This allows a more general solution, given sufficiently large number of selected paths. Second, the cost logic is easier to comprehend (and also adjust). In the first-principles approach the developer must manually handle processing and offload fractions and be explicitly aware of the datacenter capacity and load. In the Chopin optimization, there is only one processing fraction per path and the optimization automatically chooses best routing. Additionally, complex re-routing logic is not necessary, as the traffic will be routed on the best path possible automatically.

5.4 Implementation Using SDN

In this section, we describe how to implement SNIPS using SDN. The controller installs rules on the switches using an open API such as OpenFlow [72] to specify forwarding actions for different flow match patterns. The flow match patterns are exact or wildcard expressions over packet header fields. The ability to programmatically set up forwarding actions enables a *network-layer* solution for NIPS offloading that does not require NIPS modifications and can thus work with legacy/proprietary NIPS hardware.

5.4.1 First-principles approach

We want to set up forwarding rules to steer traffic to the different NIPSes. That is, given the $p_{c,j}$ and $o_{c,j}$ values, we need to ensure that each NIPS receives the designated amount of traffic. In order to decouple the formulation from the implementation, our goal is to translate *any* configuration into a correct set of forwarding rules.

As discussed in Section 5.2, each traffic class c is identified by application-level ports and network ingress/egress. Enterprise networks typically use structured address assignments; e.g., each site may be given a dedicated IP subnet. Thus, in our prototype we identify the class using the IP addresses (and TCP/UDP port numbers). Note that we do not constrain the addressing structure; the only requirement is that hosts at different locations are assigned addresses from non-overlapping IP prefix ranges and that these assignments are known.

For clarity, we assume that each NIPS is connected to a single SDN-enabled switch. In the context of our formulation, each abstract node N_j can be viewed as consisting of a SDN switch S_j connected to the NIPS $NIPS_j$.[6]

[6]For "inline" NIPS deployments, the forwarding rules need to be on the switch immediately upstream of the NIPS and the NIPS needs to be configured to act in "bypass" mode to allow the remaining traffic to pass through untouched.

5.4.1.1 Challenges in using SDN

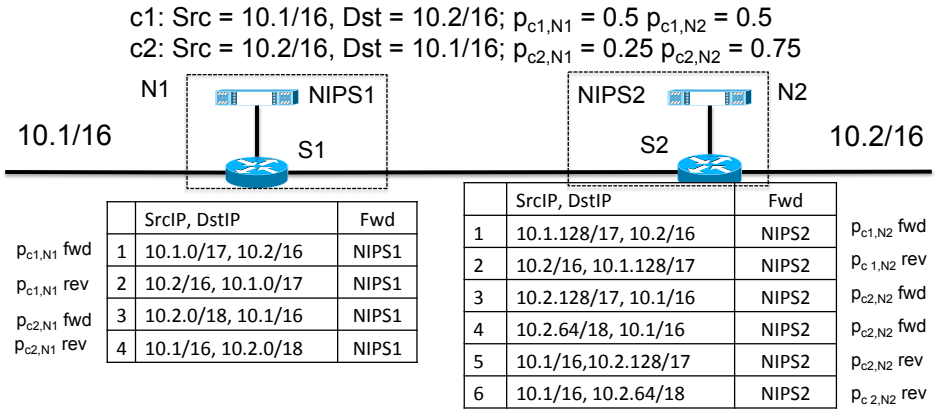
While SDN is indeed an enabler, there are three practical challenges that arise in our context. We do not claim that these are fundamental limitations of SDN. Rather, SNIPS induces new requirements outside the scope of traditional SDN/OpenFlow applications [13] and prior SDN use cases [109, 81].

Stateful processing: NIPS are *stateful* and must observe both forward and reverse flows of a TCP/UDP session for correct operation. In order to pin a session to a specific node, prior solutions for NIDS load balancing use bidirectional hash functions [40, 102]. However, such capabilities do not exist in OpenFlow and we need to explicitly ensure stateful semantics.

To see why this is a problem, consider the example in Figure 5.8 with class c1 (c1.in=S1 and c1.out = S2) with $p_{\text{c1,NIPS1}}$=$p_{\text{c1,NIPS2}}$=0.5. Suppose hosts with gateways S1 and S2 are assigned IP addresses from prefix ranges $Prefix_1$=10.1/16 and $Prefix_2$=10.2/16 respectively. Then, we set up forwarding rules so that packets with src = 10.1.0/17, dst=10.2/16 are directed to NIPS NIPS1 and those with src=10.1.128/17, dst=10.2/16 are directed to NIPS2 as shown in the top half of Figure 5.8. Thus, the volume of traffic each NIPS processed matches the SNIPS optimization. Note that we need two rules, one for each direction of traffic. [7]

There is, however, a subtle problem. Consider a different class c2 whose c2.in = S2 and c2.out = S1. Suppose $p_{\text{c2,NIPS1}} = 0.25$ and $p_{\text{c2,NIPS2}} = 0.75$. Without loss of generality, let the split be src = 10.2.0/18, dst = 10.1/16 for NIPS1 and rest to NIPS2 as shown in bottom half of Figure 5.8. Unfortunately, these new rules will create conflict. Consider a bidirectional session src = 10.1.0.1, dst = 10.2.0.1. This session will match two sets of rules; e.g., the forward flow of this session matches rule 1 on S1 while the reverse flow matches rule 4 (a reverse rule for c2) on S2. Such ambiguity could violate the stateful

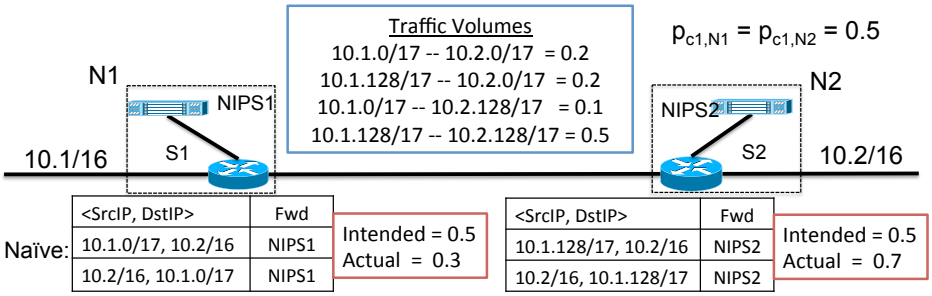
[7] For clarity, the example only shows forwarding rules relevant to NIPS; there are other basic routing rules that are not shown.



	SrcIP, DstIP	Fwd
1	10.1.0/17, 10.2/16	NIPS1
2	10.2/16, 10.1.0/17	NIPS1
3	10.2.0/18, 10.1/16	NIPS1
4	10.1/16, 10.2.0/18	NIPS1

	SrcIP, DstIP	Fwd
1	10.1.128/17, 10.2/16	NIPS2
2	10.2/16, 10.1.128/17	NIPS2
3	10.2.128/17, 10.1/16	NIPS2
4	10.2.64/18, 10.1/16	NIPS2
5	10.1/16,10.2.128/17	NIPS2
6	10.1/16, 10.2.64/18	NIPS2

Figure 5.8: Potentially conflicting rules with bidirectional forwarding rules for stateful processing. The solution in this case is to logically merge these conflicting classes.



<SrcIP, DstIP>	Fwd
10.1.0/17, 10.2/16	NIPS1
10.2/16, 10.1.0/17	NIPS1

<SrcIP, DstIP>	Fwd
10.1.128/17, 10.2/16	NIPS2
10.2/16, 10.1.128/17	NIPS2

Figure 5.9: NIPS loads could be violated with a non-uniform distribution of traffic across different prefix subranges. The solution in this case is a weighted volume-aware split.

processing requirement if the forward and reverse directions of a session are directed to different NIPS.

Skewed volume distribution: While class merging ensures stateful processing, using prefix-based partitions may not ensure that the load on the NIPS matches the optimization result. To see why, consider Figure 5.9 with a single class and two NIPS, NIPS1 and NIPS2, with an equal split. The straw man solution steers traffic between 10.1.0/17–10.2/16 to NIPS1 and the remaining (10.1.128/17–10.2/16) to NIPS2. While this splits the prefix space equally, the actual load may be skewed if the volume is distributed as shown. The actual load on the NIPS nodes will be 0.3 and 0.7 instead of the intended 0.5:0.5. This non-uniform distribution could happen for several reasons; e.g., hotspots of activity or unassigned regions of the address space.

Potential routing loops: Finally, there is a corner case if the same switch is on the path to/from the data center. Consider the route: $\langle in, \ldots, S_{offload}, \ldots, S_i, S_j, \ldots, S_d, d, S_d, \ldots, S_i, S_j, \ldots, out \rangle$. With flow-based forwarding rules, S_j cannot decide if a packet needs to be sent toward the datacenter d or toward egress out. (Note that this is not a problem for S_d itself; it can use the input interface on which the packet arrived to determine the forwarding action.)

We could potentially address some of these issues by modifying the optimization (e.g., choose a loop-free offload point for (2) or rewrite the optimization with respect to merged classes for (1).) Our goal is to decouple the formulation from the implementation path. That is, we want to provide a correct SDN-based realization of SNIPS without making assumptions about the structure of the optimization solution or routing strategies.

5.4.1.2 Our approach

Next, we discuss our approaches to address the above challenges. At a high-level, our solution builds on and extends concurrent ideas in the SDN literature [81, 47, 109]. However, to the best of our understanding, these current solutions do not handle conflicts due to stateful processing or issues of load imbalance across prefixes.

Class merging for stateful processing: Fortunately, there is a simple yet effective solution to avoid such ambiguity. We identify such conflicting classes—i.e., classes c_1 and c_2 with $c_1.in = c_2.out$ and vice versa[8]—and logically *merge* them. We create a merged class c' whose $p_{c',j}$ and $o_{c',j}$ are (weighted) combinations of the original responsibilities so that the load on each NIPS $NIPS_j$ matches the intended loads. Specifically, if the resource footprints $F^r_{c_1}$ and $F^r_{c_2}$ are the same for each resource r, then it suffices to set $p_{c',j} = \frac{S(c_1)\times p_{c_1,j} + S(c_2)\times p_{c_2,j}}{S(c_1)+S(c_2)}$. In Figure 5.8, if the volumes for c1 and c2 are equal, the effec-

[8]If the classes correspond to different well-known application ports, then we can use the port fields to disambiguate the classes. In the worst case, they may share some sets of application ports and so we could have sessions whose port numbers overlap.

tive fractions are $p_{c',\mathsf{NIPS1}} = \frac{0.5+0.25}{2}$ and $p_{c',\mathsf{NIPS2}} = \frac{0.5+0.75}{2}$. We can similarly compute the effective offload values as well. If the resource footprints $F^r_{c_1}$ and $F^r_{c_2}$ are not the same for each resource r, however, then an appropriate combination can be computed using an additional optimization.

Volume-aware partitioning: A natural solution to this problem is to account for the volumes contributed by different prefix ranges. While this problem is theoretically hard (being reducible to knapsack-style problems), we use a simple heuristic described below that performs well in practice, and is quite efficient.

Let $PrefixPair_c$ denote the IP subnet pairs for the (merged) class c. That is, if $c.in$ is the set $Prefix_{in}$ and $c.out$ is the set $Prefix_{out}$, then $PrefixPair_c$ is the cross product of $Prefix_{in}$ and $Prefix_{out}$. We partition $PrefixPair_c$ into non-overlapping blocks $PrefAtom_{c,1}$ ... $PrefAtom_{c,n}$. For instance, if each block is a $/24 \times /24$ subnet and the original $PrefixPair$ is a $/16 \times /16$, then the number of blocks is $n = \frac{2^{16} \times 2^{16}}{2^8 \times 2^8} = 65536$. Let $S(k)$ be the volume of traffic in the k^{th} block.[9] Then, the fractional weight for each block is $w_k = \frac{S(k)}{\sum_{k'} S(k')}$.

We discretize the weights so that each block has weight either δ or zero, for some suitable $0 < \delta < 1$. For any given δ, we choose a suitable partitioning granularity so that the error due to this discretization is minimal. Next, given the $p_{c,j}$ and $o_{c,j}$ assignments, we run a pre-processing step where we also "round" each fractional value to be an integral multiple of δ.

Given these rounded fractions, we start from the first assignment variable (some $p_{c,j}$ or $o_{c,j}$) and block $PrefAtom_{c,1}$. We *assign* the current block to the current fractional variable until the variable's demand is satisfied; i.e., if the current variable, say $p_{c,j}$, has the value 2δ, then it is assigned two non-zero blocks. The only requirement for this procedure to be correct is that each variable value is satisfied by an integral number of blocks; this is true because each weight is 0 or δ and each variable value is an integral multiple

[9]These can be generated from flow monitoring reports or statistics exported by the OpenFlow switches themselves.

of δ. With this assignment, the volume of traffic meets the intended $p_{c,j}$ and $o_{c,j}$ values (modulo rounding errors).

Handling loops using packet tagging: To handle loops, we use *packet tags* similar to prior work [81, 47]. Intuitively, we need the switches on the path from the datacenter to the egress to be able determine that a packet has already been forwarded. Because switches are stateless, we add tags so that the packet itself carries the relevant "state" information. To this end, we add an OpenFlow rule at S_d to set a *tag bit* to packets that are entering *from* the datacenter. Downstream switches on the path to out use this bit (in conjunction with other packet header fields) to determine the correct forwarding action. In the above path, S_j will forward packets with tag bit 0 toward d and packets with bit 1 toward out.

Given these building blocks we translate the LP solution into an SDN configuration in three steps:

1. Identify conflicting classes and merge them.

2. Use a weighted scheme to partition the prefix space for each (merged) class so that the volume matches the load intended by the optimization solution.

3. Check for possible routing loops in offloaded paths and add corresponding tag addition rules on the switches.

We implement these as custom modules in the POX SDN controller [78]. We choose POX mostly due to our familiarity; these extensions can be easily ported to other platforms. One additional concern is how packets are handled during SNIPS rule updates to ensure stateful processing. To address this we can borrow known techniques from the SDN literature [84].

5.4.2 Deployment with Chopin

Deployment is also greatly simplified since Chopin rule generation engine (recall Section 3.6) contains logic for volume-aware traffic splitting, flow affinity, and middlebox loop handling. Hence Chopin deployment does not require re-inventing custom SDN deployment logic.

In addition, SNIPS can benefit from the robustness features of Chopin's composition (recall Chapter 4), such as tolerance to traffic variations, not captured in the original SNIPS formulation.

5.5 Evaluation

We split the evaluation section into two parts:

1. comparison of original SNIPS and Chopin-based optimizations; followed by
2. evaluation of the SNIPS benefits with respect to previous work.

Setup: We use realistic network topologies from the TopologyZoo dataset [55]. Due to the absence of public traffic data, we use a *gravity model* to generate the traffic matrix specifying the volume of traffic between every pair of network nodes for the AS-level topologies. For simplicity, we consider only one application-level class and assume there is a single datacenter located at the node that observes the largest volume of traffic.

We configure the node and link capacities as follows. We assume a baseline *ingress* deployment (without offloading or on-path distribution) where all NIPS processing occurs at the ingress of each end-to-end path. Then, we compute the maximum load across all ingress NIPS and set the capacity of each NIPS to this value and the datacenter capacity to be $10\times$ this node capacity. For link capacities, we simulate the effect of routing traffic without any offloading or NIPS-induced packet drops, and compute the maximum volume observed on the link. Then, we configure the link capacities such that the maximum loaded link is at $\approx 35\%$ load.

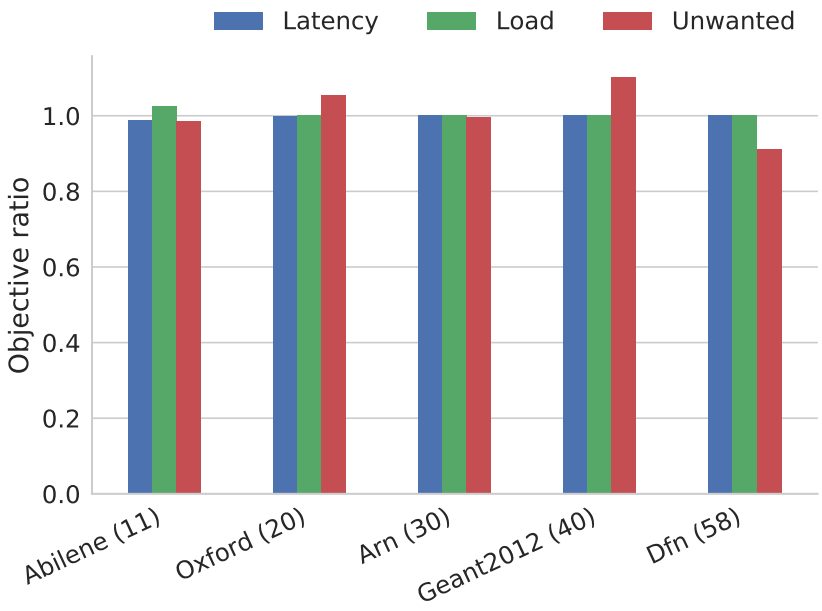


Figure 5.10: Ratio of objective functions of SNIPS optimization and Chopin-based optimization for different metrics. Values ≤ 1 indicate that Chopin-based optimization achieves better (lower) value.

5.5.1 SNIPS and Chopin

We start by comparing the results produced by the original SNIPS optimization and the SNIPS-Chopin optimization for identical topology and traffic matrix. Figure 5.10 shows the ratio of objective function value produced by the SNIPS optimization to the SNIPS-Chopin optimization. In this setup, both optimizations used equal weights across different objective functions and the Chopin version used 5 preselected paths using the relaxed path search (as described in Section 4.3.3). Values equal to 1 indicate identical solutions. Values ≥ 1 show that tailored first-principles optimization outperforms SNIPS-Chopin and vise versa. In all evaluated topologies, SNIPS-Chopin performs near-identically to the original formulation, and in one case (Abilene topology) even outperforms the first-principles optimization. This is due to the fact that Chopin is not limited to using a single routing path, but instead can choose multi-path routing, exploiting spare capacity in the network. This can negatively impact the unwanted traffic footprint (e.g., Oxford and Geant2012 topologies). However, the penalty is small, $\leq 5\%$.

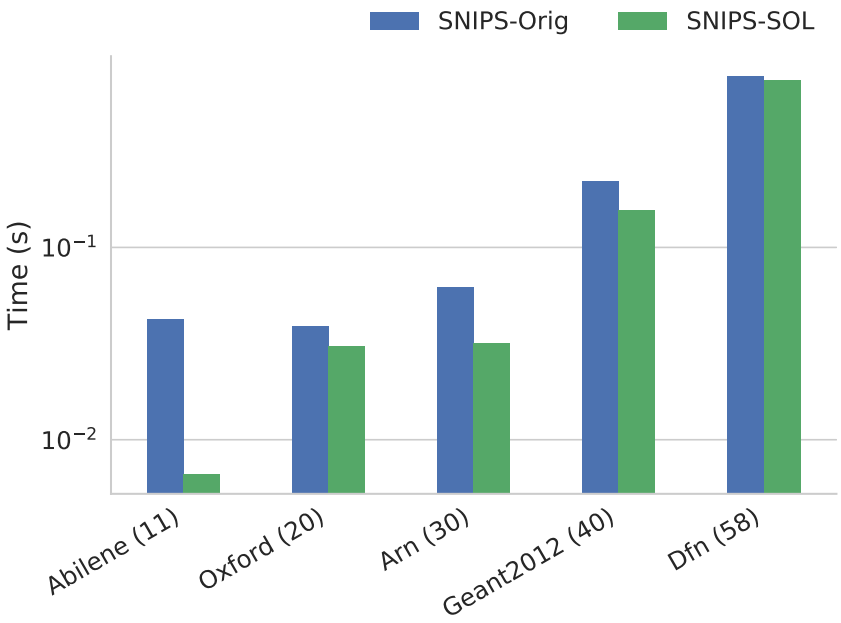


Figure 5.11: Time to compute optimal solution using original SNIPS and SNIPS-Chopin optimizations.

We also compare the runtime of both optimizations, as solved by the same Gurobi solver. Figure 5.11 shows that the Chopin optimization is significantly faster in some cases (e.g., Abilene, Oxford topologies). This is due to the offline selection of optimal paths.

5.5.2 SNIPS benefits

In evaluating SNIPS we highlight the performance benefits over other NIPS architectures and provide some system benchmarks for the SDN implementation.

We start with a baseline result with a simple configuration before evaluating the sensitivity to different parameters. For the baseline, we set the SNIPS parameters $\beta = \alpha = 0.333$; i.e., all three factors (latency, unwanted hops, load) are weighted equally in the optimization. We fix the fraction of unwanted traffic to be 10%. For all results, the maximum allowable link load is 40%.

Improvement over current NIPS architectures: We compare the performance of

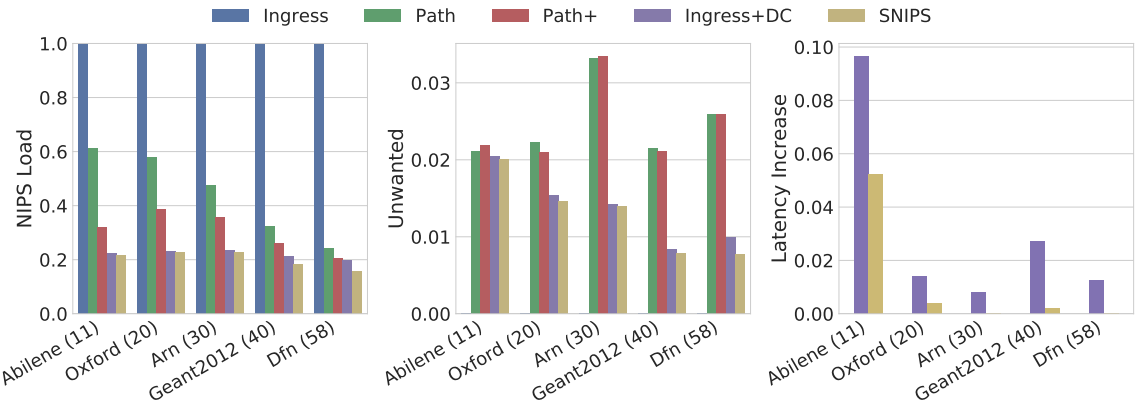


Figure 5.12: Trade-offs between current deployments and SNIPS

SNIPS against today's *Ingress* NIPS deployments. As an intermediary point, we also consider three other deployments: 1) *Ingress+DC* deployment, where all processing/offloading happens at the ingress of each path and the datacenter. 2) *Path* deployment, modeling the on-path deployment described in [89]; and 3) *Path+*: identical to *Path* except each node has an increased capacity of $DCap^r/N$.

Figure 5.12 shows three normalized metrics for the topologies: load, added latency, and unwanted footprint. For ease of presentation, we normalize each metric by the maximum possible value for a specific topology so that it is between 0 and 1.[10] Higher values indicate less desirable configurations (e.g., higher load or latency).

By definition, the Ingress deployment introduces no additional latency and has no unwanted footprint, since all of the processing is at the edge of the network. Such a deployment, however, can suffer overload problems as shown in the result. SNIPS offers a more flexible trade-off: a small increase in latency and unwanted footprint for a significant reduction in the maximum compute load. We reiterate that SNIPS does not affect the security guarantees; it will drop *all* unwanted traffic, but it may choose to do so after a few extra hops. In some topologies (e.g., Geant2012) SNIPS can reduce the maximum load by 5× compared to a naive ingress deployment while only increasing the latency by 2%. Note that these benefits arise with a very simple equi-weighted trade-off across

[10]Hence the values could be different across topologies even for the ingress deployment.

the three objective components; the benefits could be even better with other configurations.

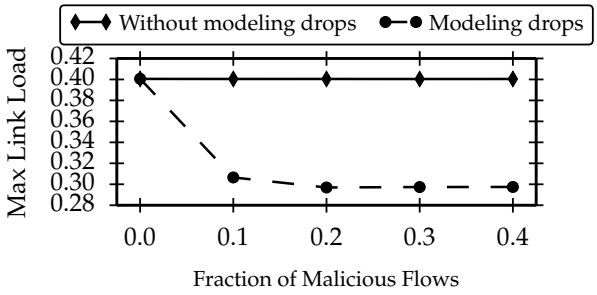


Figure 5.13: Link load as a function of fraction of "unwanted" traffic.

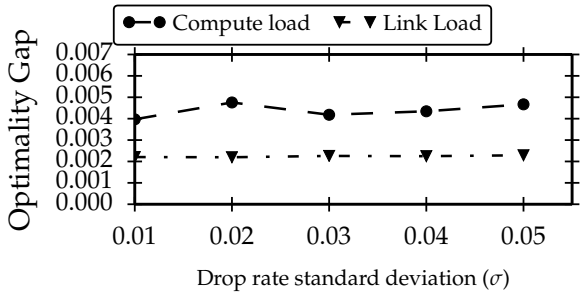


Figure 5.14: Compute and link load optimality gap as functions of drop rate deviation; estimated drop rate = distribution mean $\mu = 0.1$

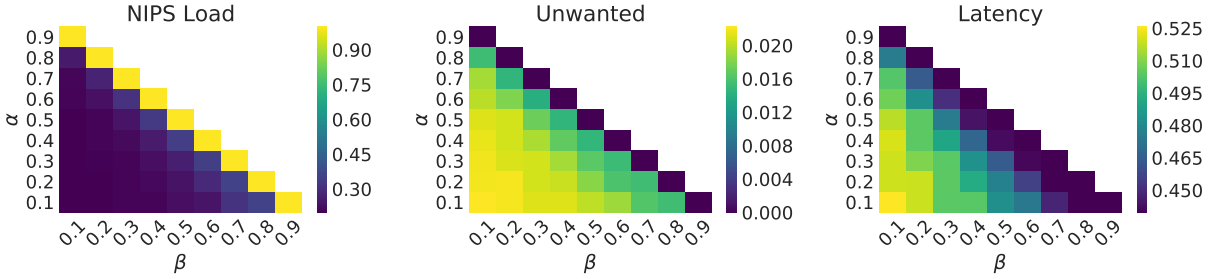


Figure 5.15: Visualizing trade-offs in choosing different weight factors on Abilene topology.

Impact of modeling traffic drops: SNIPS provides a higher fidelity model compared to past works in NIDS offloading because it explicitly incorporates the impact of traffic drops. We explore the impact of modeling these effects. For this result, we choose the Internet2 topology and use our simulator to vary the fraction of malicious flows in the network. Figure 5.13 shows the maximum observed link loads, averaged over 50 simulation runs. In addition to directly using the SNIPS-recommended strategy, we also consider a naive setup that does not account for such drops.

There are two key observations. First, the max link load is significantly lower with SNIPS which means that SNIPS can exploit more opportunities to offload under overload compared to the naive model. Second, by assuming no drops, "no drop" setup ig-

nores the $HopsUnwanted$ factor, thus potentially obstructing the link to the datacenter with unwanted traffic that could have been dropped at an earlier point in the network (this effect is represented in Figure 5.13).

5.5.3 Sensitivity Analysis

Sensitivity to weights: As an illustrative result, we show the result of varying the weighting factors for the Abilene topology in Figure 5.15. (We show only one topology due to space limitations). In the figure, darker regions depict higher values, which are less desirable. Administrators can use such visualizations to customize the weights to suit their network topology and traffic patterns and avoid undesirable regions. In particular, our equi-weighted configuration is a simple but reasonable choice (e.g., mostly low shades of gray in this graph).

Sensitivity to estimation errors: We also show that the parameter estimation (such as drop rate) for our framework need not be precise. For this, we choose to run a number of simulations with imperfect knowledge of the drop rate. In that case, the drop rate is sampled from a Gaussian distribution with mean of 0.1 (the estimated drop rate) and changing standard deviation σ. Figure 5.14 shows the relative gap for compute and link loads, between values predicted by the optimization with exact drop rate knowledge and the simulated values. This result shows that even with large noise levels the difference in load on links and nodes is insignificant.

CHAPTER 6: Conclusions

With many appealing capabilities, Software-Defined Networking continues to gain popularity. However, its successful adoption depends on the ease with which new applications can be created and deployed. This dissertation explores one particular class of applications well-suited to the SDN paradigm — optimization applications. We investigated a variety of such applications and their requirements, and shown that a general and efficient framework can be built to express them. The framework (SOL) and its extensions (Chopin) provide numerous features: rapid application prototyping, generalized heuristics for fast solution computation, simplified deployment, and automated robust application composition.

We evaluated SOL and Chopin by expressing multiple existing applications and developing a new one (SNIPS), showing benefits in both usability and efficiency. Based on the empirical evidence, we conclude that: *a path-based optimization framework supports expression and composition of different applications, and generates resource-efficient solutions*.

If enterprises and Internet Service Providers choose to embrace SDN as their core network technology, we hope that this work will help make that a vision a reality. The tools made available to the public as a result of this research lower the barrier of entry to the adoption of SDN. We deem efficient and flexible networks to be at the foundation of modern computer infrastructure. Hence we are optimistic that enabling innovation in this space has the potential to impact users beyond solely the tech community.

We also believe that this work paves the way for future research in SDN and optimization. For example, is it possible to show a theoretical approximation bound on the solution given certain path selection strategies? Are there resource-management applications that cannot be (efficiently) expressed using the path abstraction? What are the

usability concerns for the given framework from a network operator's point of view? We leave these questions to be addressed in future work.

BIBLIOGRAPHY

[1] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network flows: theory, algorithms, and applications*. Prentice hall, 1993.

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

[3] Anthony B Atkinson. On the measurement of inequality. *Journal of economic theory*, 2(3):244–263, 1970.

[4] Alvin AuYoung, Sujata Banerjee, Jeongkeun Lee, Jeffrey C Mogul, Jayaram Mudigonda, Lucian Popa, Puneet Sharma, and Yoshio Turner. Corybantic: Towards the modular composition of SDN control programs. In *ACM HotNets*, 2013.

[5] Alvin AuYoung, Yadi Ma, Sujata Banerjee, Jeongkeun Lee, Puneet Sharma, Yoshio Turner, Chen Liang, and Jeffrey C Mogul. Democratic resolution of resource conflicts between sdn control programs. In *ACM CoNEXT*, pages 391–402. ACM, 2014.

[6] Yossi Azar, Edith Cohen, Amos Fiat, Haim Kaplan, and Harald Racke. Optimal oblivious routing in polynomial time. In *ACM Symposium on Theory of Computing*, pages 383–388. ACM, 2003.

[7] Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. *Robust optimization*. Princeton University Press, 2009.

[8] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[9] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. Deep packet inspection as a service. In *ACM CoNEXT*, pages 271–282, 2014.

[10] Andrew T Campbell, Herman G De Meer, Michael E Kounavis, Kazuho Miki, John B Vicente, and Daniel Villela. A survey of programmable networks. *ACM SIGCOMM Computer Communication Review*, 29(2):7–23, 1999.

[11] Eduardo Camponogara and Luiz Fernando Nazari. Models and algorithms for optimal piecewise-linear function approximation. *Mathematical Problems in Engineering*, 2015.

[12] Zizhong Cao, Murali Kodialam, and TV Lakshman. Traffic steering in software defined networks: planning and online routing. In *ACM SIGCOMM Workshop on Distributed Cloud Computing*, pages 65–70, 2014.

[13] Martín Casado et al. Ethane: Taking control of the enterprise. In *ACM SIGCOMM*, 2007.

[14] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. Robust validation of network designs under uncertain demands and failures. In *USENIX Symposium on Networked Systems Design and Implementation*, 2017.

[15] Moses Charikar, Yonatan Naamad, Jennifer Rexford, and Kelvin Zou. Multi-Commodity Flow with In-Network Processing. Manuscript, www.cs.princeton.edu/~jrex/papers/mopt14.pdf.

[16] Emilie Danna, Subhasree Mandal, and Arjun Singh. A practical algorithm for balancing the max-min fairness and throughput objectives in traffic engineering. In *IEEE Conference on Computer Communications*, pages 846–854, 2012.

[17] Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. Multi-objective optimization. In *Decision Sciences: Theory and Practice*, pages 145–184. CRC Press, 2016.

[18] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, (2):222–232, 1987.

[19] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Predicting the resource consumption of network intrusion detection systems. In *Symposium on Recent Advances in Intrusion Detection*, 2008.

[20] Seyed Kaveh Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic DDoS defense. In *USENIX Security Symposium*, pages 817–832, 2015.

[21] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving traffic demands for operational IP networks: methodology and experience. *ACM/IEEE Transactions on Networking*, 9, 2001.

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

[23] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional IP routing protocols. *IEEE Communications Magazine*, 40, 2002.

[24] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *IEEE Conference on Computer Communications*, volume 2, 2000.

[25] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291, 2011.

[26] Robert Fourer, David M Gay, and Brian W Kernighan. *AMPL: A mathematical programming language*. AT&T Bell Laboratories Murray Hill, 1987.

[27] Open Networking Fundation. Software-defined networking: The new norm for networks. 2:2–6, 2012.

[28] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Vyas Sekar, and Aditya Akella. Stratos: A network-aware orchestration layer for virtual middleboxes in clouds. *arXiv preprint arXiv:1305.0209*, 2013.

[29] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. In *ACM SIGCOMM*, 2014.

[30] Glen Gibb, Hongyi Zeng, and Nick McKeown. Outsourcing network functionality. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2012.

[31] Google Research: No Mobile Site = Lost Customers. http://www.forbes.com/sites/roberthof/2012/09/25/google-research-no-mobile-site-lost-customers/.

[32] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[33] Gurobi. http://www.gurobi.com/.

[34] Peter J Hammond. Equity, arrow's conditions, and rawls' difference principle. *Econometrica: Journal of the Econometric Society*, pages 793–804, 1976.

[35] Nikhil Handigol, Mario Flajslik, Srini Seetharaman, Nick McKeown, and Ramesh Johari. Aster*x: Load-balancing as a network primitive. In *9th GENI Engineering Conference*, pages 1–2, 2010.

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

[37] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[38] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving energy in data center networks. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 19–21, 2010.

[39] V. Heorhiadi, S. K. Fayaz, M. K. Reiter, and V. Sekar. SNIPS: A software-defined approach for scaling intrusion prevention systems via offloading. In *10th International Conference on Information Systems Security*, December 2014.

[40] Victor Heorhiadi, Michael K. Reiter, and Vyas Sekar. New opportunities for load balancing in network-wide intrusion detection systems. In *ACM CoNEXT*, 2012.

[41] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. Simplifying software-defined network optimization using SOL. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 223–237, 2016.

[42] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, pages 15–26, 2013.

[43] ILOG, IBM. Cplex. https://www.ibm.com/us-en/marketplace/ibm-ilog-cplex, 2016.

[44] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, pages 3–14, 2013.

[45] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. Kargus: a highly-scalable software-based intrusion detection system. In *ACM Conference on Computer and Communications Security*, 2012.

[46] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 87–101, 2015.

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

[48] Norman L Johnson, Samuel Kotz, and N Balakrishnan. *Continuous multivariate distributions, volume 1, models and applications*, volume 59. New York: John Wiley & Sons, 2002.

[49] Min Suk Kang, Virgil D Gligor, and Vyas Sekar. SPIFFY: Inducing cost-detectability tradeoffs for persistent link-flooding attacks. *Symposium on Network and Distributed System Security*, 2016.

[50] N Kang, M Ghobadi, J Reumann, and A Shraer. Efficient traffic splitting on SDN switches. In *ACM CoNEXT*, 2015.

[51] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the one big switch abstraction in software-defined networks. In *ACM CoNEXT*, pages 13–24, 2013.

[52] Kalapriya Kannan and Subhasis Banerjee. Scissors: Dealing with header redundancies in data centers through SDN. In *Workshop on Systems Virtualiztion Management*, pages 295–301. IEEE, 2012.

[53] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[54] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998.

[55] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765 –1775, October 2011.

[56] Murali Kodialam, TV Lakshman, and Sudipta Sengupta. Traffic-oblivious routing in the hose model. *IEEE/ACM Transactions on Networking*, 19(3):774–787, 2011.

[57] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. Network virtualization in multi-tenant datacenters. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 203–216, 2014.

[58] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, and Robert Soulé. Kulfi: Robust traffic engineering using semi-oblivious routing. *arXiv preprint arXiv:1603.01203*, 2016.

[59] TV Lakshman, T Nandagopal, R Ramjee, K Sabnani, and T Woo. The softrouter architecture. In *ACM HotNets*, volume 2004. Citeseer, 2004.

[60] Janghaeng Lee, Sung Ho Hwang, Neungsoo Park, Seong-Won Lee, Sunglk Jun, and Young Soo Kim. A high performance NIDS using FPGA-based regular expression matching. In *ACM Symposium on Applied Computing*, 2007.

[61] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, Anja Feldmann, et al. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *USENIX Annual Technical Conference*, 2014.

[62] Xuan Liu, Sudhir Mohanraj, Michal Pioro, and Deep Medhi. Multipath routing from a traffic engineering perspective: How beneficial is it? In *IEEE International Conference on Network Protocols*, pages 143–154, 2014.

[63] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[64] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *USENIX Security Symposium*, 2010.

[65] Mininet. http://mininet.org/.

[66] Stuart Mitchell, Michael O'Sullivan, and Iain Dunning. Pulp: a linear programming toolkit for python, 2011.

[67] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *USENIX Symposium on Networked Systems Design and Implementation*, volume 13, pages 1–13, 2013.

[68] Mosek. https://mosek.com/.

[69] Network functions virtualisation – introductory white paper. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[70] Navid Nikaein, Eryk Schiller, Romain Favraud, Kostas Katsalis, Donatos Stavropoulos, Islam Alyafawi, Zhongliang Zhao, Torsten Braun, and Thanasis Korakis. Network store: Exploring slicing in future 5G networks. In *International Workshop on Mobility in the Evolving Internet Architecture*, pages 8–13. ACM, 2015.

[71] Opendaylight SDN controller. http://www.opendaylight.org/.

[72] Openflow standard. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[73] S. Palkar, C. Lan, S. Han, K. Jang, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A runtime framework for network functions. In *ACM Symposium on Operating Systems Principles*, 2015.

[74] Christos H Papadimitriou. On the complexity of integer programming. *Journal of the ACM (JACM)*, 28(4):765–768, 1981.

[75] Antonis Papadogiannakis, Michalis Polychronakis, and Evangelos P. Markatos. Tolerating Overload Attacks Against Packet Capturing Systems. In *USENIX Annual Technical Conference*, 2012.

[76] Vern Paxson. Bro: a system for detecting network intruders in real-time. In *Proc. USENIX Security*, 1998.

[77] Ruben E. Perez, Peter W. Jansen, and Joaquim R. R. A. Martins. pyOpt: A Python-based object-oriented framework for nonlinear constrained optimization. *Structures and Multidisciplinary Optimization*, 45(1):101–118, 2012.

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

[79] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM*, 45(4):29–42, 2015.

[80] Private communication with UNC administrators, 2013.

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

[82] Saqib Raza, Guanyao Huang, Chen-Nee Chuah, Srini Seetharaman, and Jatinder Pal Singh. Measurouting: a framework for routing assisted traffic monitoring. *ACM/IEEE Transactions on Networking*, 20(1):45–56, 2012.

[83] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular SDN programming with Pyretic. *;login: Magazine*, 38(5):128–134, 2013.

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

[85] Martin Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[86] Matthew Roughan. Simplifying the synthesis of internet traffic matrices. *ACM SIGCOMM Computer Communication Review*, 35, 2005.

[87] Beverly Schwartz, Alden W Jackson, W Timothy Strayer, Wenyi Zhou, R Dennis Rockwell, and Craig Partridge. Smart packets for active networks. In *Open Architectures and Network Programming Proceedings, 1999*, pages 90–97. IEEE, 1999.

[88] SDN app store. https://marketplace.saas.hpe.com/sdn, January 2017.

[89] Vyas Sekar, Ravishankar Krishnaswamy, Anupam Gupta, and Michael K. Reiter. Network-wide deployment of intrusion detection and prevention systems. In *ACM CoNEXT*, 2010.

[90] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. CSAMP: a system for network-wide flow monitoring. In *USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[91] Farhad Shahrokhi and David W Matula. The maximum concurrent flow problem. *Journal of the ACM (JACM)*, 37(2):318–334, 1990.

[92] Kevin Shatzkamer. App store portal providing point-and-click deployment of third-party virtualized network functions, April 4 2014. US Patent App. 14/245,193.

[93] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *ACM SIGCOMM*, 2012.

[94] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru M Parulkar. Can the production network be the testbed? In *USENIX Symposium on Operating Systems Design and Implementation*, volume 10, pages 1–6, 2010.

[95] Seugwon Shin, Phillip Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular composable security services for software-defined networks. In *Symposium on Network and Distributed System Security*, 2013.

[96] Randy Smith, Cristian Estan, and Somesh Jha. XFA: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy*, 2008.

[97] Robert Soule, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *ACM CoNEXT*, 2014.

[98] Ralph E Steuer. *Multiple criteria optimization: theory, computation, and applications*. Wiley, 1986.

[99] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A network-state management service. In *ACM SIGCOMM*, SIGCOMM '14, pages 563–574, New York, NY, USA, 2014. ACM.

[100] David L Tennenhouse, Jonathan M Smith, W David Sincoskie, David J Wetherall, and Gary J Minden. A survey of active network research. *IEEE communications Magazine*, 35(1):80–86, 1997.

[101] Paul Tune and Matthew Roughan. Spatiotemporal traffic matrix synthesis. *ACM SIGCOMM Computer Communication Review*, 45(4):579–592, 2015.

[102] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The NIDS cluster: scalable, stateful network intrusion detection on commodity hardware. In *Symposium on Recent Advances in Intrusion Detection*, 2007.

[103] Jacobus E Van der Merwe, Sean Rooney, L Leslie, and Simon Crosby. The tempest-a practical framework for network programmability. *IEEE network*, 12(3):20–28, 1998.

[104] Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[105] Nedeljko Vasić, Prateek Bhurat, Dejan Novaković, Marco Canini, Satyam Shekhar, and Dejan Kostić. Identifying and using energy-critical paths. In *ACM CoNEXT*, 2011.

[106] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Symposium on Recent Advances in Intrusion Detection*, 2009.

[107] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. MIDeA: a multi-parallel intrusion detection architecture. In *ACM Conference on Computer and Communications Security*, 2011.

[108] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: simplifying SDN programming using algorithmic policies. *ACM SIGCOMM Computer Communication Review*, 43(4):87–98, 2013.

[109] R. Wang, D. Butnariu, and J. Rexford. Openflow-based server load balancing gone wild. In *Hot-ICE*, 2011.

[110] Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

[111] World intrusion detection and prevention markets. http://www-935.ibm.com/services/us/iss/pdf/esr_intrusion-detection-and-prevention-systems-markets.pdf.

[112] Fang Yu, T. V. Lakshman, Martin Austin Motoyama, and Randy H. Katz. SSA: a power and memory efficient scheme to multi-match packet classification. In *ACM Symposium on Architectures for Networking and Communications Systems*, 2005.