LANGUAGES FOR PATH-BASED NETWORK PROGRAMMING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Shrutarshi Basu

August 2018

LANGUAGES FOR PATH-BASED NETWORK PROGRAMMING

Shrutarshi Basu, Ph.D.

Cornell University 2018

The notion of a *path* is an important abstraction for reasoning about and managing computer networks. By thinking of network configuration in terms of paths (instead of individual devices), network administrators can reap significant benefits in terms of usability, performance and correctness. Paths themselves can be specified and reasoned about using foundational computational tools such as regular expressions and Kleene Algebras. Furthermore, path-based abstractions can express key network behavior such as isolation requirements and bandwidth constraints, while supporting heterogeneity in terms of devices, physical substrates and administrative domains.

This dissertation shows that it is possible to use specifications of network paths, extended with constraints on traffic classes, bandwidth, and the capabilities of network devices, to enable flexible management of modern networks. This is realized by developing path-oriented domain-specific languages to express network policies. These languages show that by starting with abstractions for network paths, it is possible to specify route and bandwidth requirements for classes of traffic, delegate policy management to trusted parties, and enable high-level management of heterogenous networks.

Furthermore, by leveraging techniques and tools such as mixed-integer programming, SAT solvers and SDN controller frameworks, we have built practical compilers and runtimes for these languages. These compilers take high-level specifications of network policies and generate efficient configurations for a range of network devices including packet and optical switches, middlebox frameworks and end-hosts. We have tested these implementations by building and benchmarking a range of practical applications on real-world networks.

BIOGRAPHICAL SKETCH

Shrutarshi Basu was born and raised in Calcutta, India, with a few years in the United Kingdom. He attended St. Xavier's Collegiate School in Calcutta, where a number of excellent teachers helped him discover and cultivate a love of science and technology, as well as literature and public speaking. From 2007 to 2011 he attended Lafayette College in Easton, Pennsylvania, where alongside degrees in Electrical and Computer Engineering and Computer Science he picked up an appreciation of Renaissance and modern art, craft beers, and summer barbeques. In 2011 he enrolled as a PhD student in the Computer Science department at Cornell University. Through the course of his PhD he conducted research on programming languages and software-defined networks, interned at GrammaTech Inc. and Fujitsu Labs of America, started investigating the connections between the study of computation and the study and practice of law, and learned how to swim. After graduating from Cornell he will be a Post-Doctoral Research Scholar at Harvard University where he hopes to continue acquiring useful skills and mildly eccentric interests.

This document is dedicated to my grandmother, Lily Ghosh, without whose selfless dedication to my well-being I would not be here today.

ACKNOWLEDGMENTS

First, I would like to thank my thesis advisor Nate Foster for many years of support, mentorship, guidance and advice. During my time at Cornell he has taught to be a curious researcher, a deep thinker, an effective communicator, a better programmer and engineer, and a more capable human being.

I will always be grateful to my committee members for their advice and support over the years. I would like to thank Robert Kleinberg for many fruitful conversations and for pointing me in new and useful directions. Dexter Kozen has been an inspiration and guide, both in my work and as a wonderful and kind human being. Finally, my discussions with Cynthia R. Farina have been a source of important insights, and she has helped me gain confidence in pursuing new and interesting future directions.

The work described in this document has been made possible by the help and advice of many collaborators. At Cornell University, Robert Soulé, Robert Kleinberg and Emin Gün Sirer not only aided the development of early work, but helped me grow and progress as a researcher and new graduate student. I owe a special debt to Paparao Palacharla and Xi Wang at Fujitsu Labs of America for introducing me to a new line of work and helping me understand the problems tackled in the latter half of my PhD. Further thanks are due to Hossein Hojjat, Christian Skalka, Parisa Jalili Marandi and Fernando Pedone for their advice and support. I would also like to thank Praveen Kumar, Steffen Smolka and Han Wang for numerous clarifying conversations.

Prior to starting my PhD, Chun Wai Liew, Ed Kerns and Barbara Ryder helped me find my footing as a computer scientist and researcher. I would not have embarked on this path without their guidance and encouragement. Thanks to Aruni Roy Chowdhury, Scott Stinner, Gregory Earle, Michael Handzo, Alex Smith, Alex Beeman, Scott Blonde, Susan Grunewald and Jenn Bell for being the best of friends during those formative years.

There are many people in Ithaca and Cornell who deserve thanks for their support over the years. The members of the Computer Science department at Cornell, especially the Programming Languages Discussion Group and Systems Lunch group, provided me an environment which was both supportive and challenging and shaped me as a researcher. Many thanks are due to my officemates over the years, including Molly Q Feldman, Zhiyuan Teo, Adith Swaminathan, Stavros Nikolaou, Joshua Gancher, Justin Hsu, and Abhishek Anand, for being supportive sounding boards, and for opening my eyes to interesting problems and far-flung areas of computer science. To my good friends in the department, Vlad Niculae, Eleanor Birrell, Jonathan Dilorenzo, Ethan Cecchetti, Andrew Hirsch, Tobias Schnabel, Sydney Zink, Natacha Crooks, Rahmtin Rohtabi, Edward Tremel, and Ross Tate, thank you for much help and support, both professional and personal. To my roommates over the years, Sean Bell, Tom Magrino, Francisco Mota and Shreesha Srinath, thanks for putting up with me, and always making me feel happy to come home. Finally, to Sandra Wayman, Heidi Vanden Brink, Quitterie Gounot, Amelia Hall, Noelle Yaeger, Anna Waymack and Nathaniel Stetson, thank you for opening your homes and hearts to me. I wouldn't be here without each and every one of you.

Last but certainly not least, to my parents, Sukla Ghosh and Alok Basu, thank you for instilling in me a love of science and learning, a respect for both kindness and knowledge, for your unquestioning support over the years, and for always believing in me, even when I didn't believe in myself.

This work has been made possible by NSF Grant CCF-1422046: "Practical Synthesis of Network Updates" and NSF Grant CNS-1413972: "Programmable Inter-Domain Observation and Control".

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Network operators today must deal with a wide range of challenges from complex policies to a proliferation of heterogeneous devices to ever-growing traffic demands. In recent years there has been a surge of activity centered around *Software-Defined Networks* (SDNs). The core idea behind SDN is to separate the *data plane* (that is responsible for forwarding packets) from the *control plane* (that enforces network-wide policies and enables network management). Ideally, this allows commoditization and optimization of the hardware devices that implement they data plane, while enabling experimentation and innovation in frameworks, languages and APIs for the control plane. Together, SDN would enable networks that combine fast, efficient switching hardware with flexible management software capable of implementing a broad range of administrative policies.

However, there remain many barriers to the goal of full network programmability. One of the key benefits of SDN allowing networks to quickly react to changes in network conditions. However, on current SDN switches, updating a forwarding table can take several seconds, limiting the ability to quickly adapt to changing conditions. Existing SDN APIs and tools are often not expressive enough to implement the full range of desired network-wide policies. For example, programming languages for SDNs often focus on packet forwarding and cannot express policies requiring, say, bandwidth allocation or richer functionality, such as deep-packet inspection and intrusion detection. These additional functions must be implemented using middleboxes or custom hardware, and controlled via tools outside the SDN framework [25, 54, 78, 5, 58]. Although there are frameworks for handling broader concerns, such as middlebox placement and traffic engineering [27, 40, 64, 67], they either fail to provide a programmable API, or expose APIs that are extremely simple.

Additionally, many existing SDN frameworks assume that the network comprises a collection homogenous devices that can be reconfigured as policies or network conditions change. Unfortunately, this assumption is unrealistic in many situations. Many large organizations deploy thousands of network devices that are supplied by multiple different vendors, not all of which support SDN. There is also a fundamental mismatch between the capabilities of SDN switches, which allow packets to be transformed in essentially arbitrary ways at each hop, and legacy devices such as IP routers, MPLS LSRs (Multiprotocol Label Switching Label Switching Routers), and optical ROADMs (Reconfigurable Optical Add-Drop Multiplexers), which simply forward packets to their destinations. A realistic network management solution must contend with the limited capabilities of heterogenous collections of devices.

The above issues suggest that a fundamentally different approach for network programming is needed. Fortunately, there has been increasing interest in treating the network as a *programmable system* using well-defined abstractions, rather than as groups of individual devices. One fruitful avenue of research has been the development of *network programming languages* that enable network behavior to be described as programs in domain specific languages, with well-defined semantics, and compilers for real network devices. In particular, languages like FatTire [65] and NetKAT [5] have developed the notion of using paths through the network as a building block for specifying network policies. In particular, FatTire allows for writing fault-tolerant SDN programs using networks paths as a core language construct. Meanwhile, one of the applications of the NetKAT language is specifying high-level policies in terms of links in a virtual topology that can be mapped to paths in a physical network.

This dissertation presents novel language designs and associated compilation techniques that expand path-oriented network programming to include support for bandwidth constraints, delegation of policies to network users (and verification thereof),

and support for networks composed of heterogenous devices. For example, to support bandwidth constraints, we have leveraged mixed integer programming (MIP) solvers to efficiently allocate the available resources of a network. Support for heterogenous networks requires mapping abstract network functionality to specific physical devices, and analyzing legacy network configurations for forwarding routes that can be leveraged by high-level policies. Finally showing the viability of these languages and their implementations has required implementing and benchmarking a variety of applications on realistic networks. The remainder of this chapter outlines our approach to dealing with the challenges of modern networks.

1.1 The Challenges of Programming Modern Networks

As described above the challenges of managing networks using high-level, SDN-like tools remain unmet. As a result, there is widespread interest, both in academia and in industry, in higher-level languages and application-facing interfaces that provide convenient control over network resources. Any framework or language that wishes to improve the state of network management must address the following concerns.

High-level Management. Modern networks can be composed of hundreds or thousands of devices. It is infeasible to manage such networks without abstractions that allow for implementing network-wide policies without configuring devices individually. We address this in the Merlin language which allows a group of permissible paths through the network to be expressed as a regular expression. Paths may be composed of physical network devices such as switches, but also abstract functions like deep packet inspection which may be performed on multiple devices in the network.

Bandwidth Control. Controlling packet flows according to bandwidth usage, and not just header characteristics, is a key component of modern network management.

A modern framework must support policies in terms of bandwidth, and ideally be able to manage allocation of available bandwidth to traffic classes. This too is tackled in the Merlin language: paths are associated with a class of network traffic, and are subject to user-specified bandwidth constraints (either a minimum guarantee of bandwidth, or a maximum cap on bandwidth usage).

Multiple Administrative Domains. In the age of cloud computing and virtual networks, a single physical network may be shared between multiple tenants, each with their own policies and requirements. Managing such a network requires balancing the needs and priorities of the tenants. Policies in the Merlin language may also be delegated to tenants, who may refine them according to their needs. However, the compiler ensures that any modifications made by tenants do not violate the administrator's original policy.

Legacy Devices. SDN deployments, where they exist, tend to be partial in nature. Any framework for real-world network programming must be able to implement high-level policies in the presence of legacy devices. The compiler for the Merlin language generates configurations for SDN switches supporting the OpenFlow protocol, for programmable middleboxes using the Click framework, and for Linux end-hosts. Separate from Merlin, we have also developed the EdgeNetKAT compiler for NetKAT programs, which supports configuring optical ROADM devices as well as OpenFlow switches.

Heterogeneous Functionality. While SDN packet switches allow packets to be transformed in essentially arbitrary ways at each hop, many devices operate at the granularity of entire flows of traffic between endpoints. SDN-like programming frameworks often assume a homogeneous collection of switches, and it is not immediately clear how to incorporate these devices and their limitations. One common example of heterogeneous networks are packet-optical hybrid networks, composed of a core of op-

tical devices connected to traditional packet networks on the edge. To program such networks, we have developed the Circuit NetKAT subset of the NetKAT language for programming optical circuit networks, and the EdgeNetKAT compiler which translates global programs for such a hybrid network into NetKAT programs which only affect the packet switches on the edge.

Performance Limitations: Existing frameworks tacitly assume that it is possible to rapidly reconfigure devices in response to policy updates, traffic shifts, topology changes, and other changes. However, current SDN switches can take several seconds to update a forwarding table. Non-packet switching devices, such as optical ROADMS, can take even longer. The EdgeNetKAT compiler described earlier tackles this issue by enabling changes to global network policy to be implemented via configuration changes to only the edge switches. Thus, as long as the edge devices are capable of fast configuration switching, it is possible to quickly implement changes to policy, even if the core of the network only supports slower changes.

1.2 Contributions and Outline

Overall, the contributions of this work are as follows:

- The design of path-based network management abstractions realized in an expressive policy language that model packet classification, forwarding, and bandwidth allocation.
- Novel compilation algorithms that compute forwarding paths and allocate bandwidth using a mixed-integer program formulation.
- Techniques for dynamically adapting policies using negotiators and related verification techniques, made possible by the language design.

- A practical framework for implementing high-level network policies, supporting heterogeneous devices, focusing on networks with optical circuit cores and SDN-enabled packet switches at the edge.
- Techniques to analyze and transform policies into equivalent edge configurations using the NetKAT framework and off-the-shelf linear programming solvers.
- Extensions to the above framework to support (i) multi-segment paths for applications like service chaining and network function virtualizaion, and (ii) path constraints for finer-grained control over fabric utilization.

The following chapters describe each of the contributions in detail. Chapter 2 covers how Merlin uses regular expressions to describe network paths and maps network functionality to particular devices. Chapter 3 covers the use of linear programming to allocate the bandwidth requirements as expressed in Merlin policies. Chapter 4 describes how the Merlin policy language enables the delegation of policy refinement to tenants of a shared network. Chapter 5 describes how SDN technologies can be applied to heterogeneous networks, focusing on optical networks. Chapter 6 continues the discussion of heterogenous networks by describing how the NetKAT programming language can be used to enable flexible programming of networks with an edge and fabric distinction. Chapter 7 describes the implementation and evaluation of the Merlin and EdgeNetKAT systems. Chapter 8 covers related work and finally Chapter 9 summarizes and concludes.

This document is based on material published previously in conference proceedings. Chapters 3 and 4 are based on work published in [71, 72] with Robert Soulé, Robert Kleinberg, Emin Gün Sirer, Nate Foster, Parisa Jalili Marandi and Fernando Pedone. Chapters 5, 6 and 7 are based on work published in [10] with Nate Foster, Hossein Hojjat, Paparao Palacharla, Xi Wang and Christian Skalka.

CHAPTER 2

SPECIFYING NETWORK PATHS USING REGULAR EXPRESSIONS

2.1 Regular Expressions and Deterministic Finite Automata

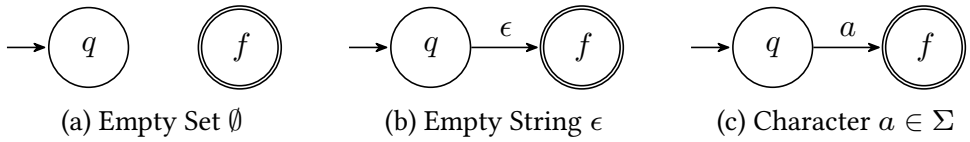
Traditionally, regular expressions ("regexes" for short) specify a pattern which is, typically used to match some text for a searching or parsing operation. When applied to strings of characters, the regular expression defines a set of strings, or a *language*. Regular expressions are defined using constants (which denote sets of strings), and operator symbols defining operations over those sets. For each regular expression E, we can also define the language it accepts as $L(E)$.

Starting with a finite alphabet of characters Σ, we can start defining regular expressions with the following constants:

- $\emptyset$ denotes the empty set. That is, $L(\emptyset)$ is $\emptyset$.
- ϵ denotes the set containing only the "empty string", i.e., a string with no characters. That is, $L(\epsilon) = \{\epsilon\}$.
- A literal character $a \in \Sigma$ denotes the set containing only the character a. That is, $L(a) = \{a\}$.
- For convenience, capital letters (R or S) are variables denoting a language. We will use the same letters to mean the regular expressions specifying the language.

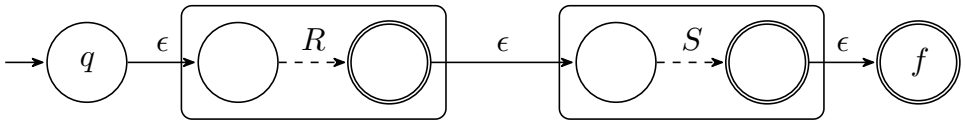
The following operations are used to combine regular expressions and the languages they denote:

- $R|S$ denotes the union of the sets denoted by R and S.
- RS denotes the set of strings obtained by concatenating a string from R with a string from S.

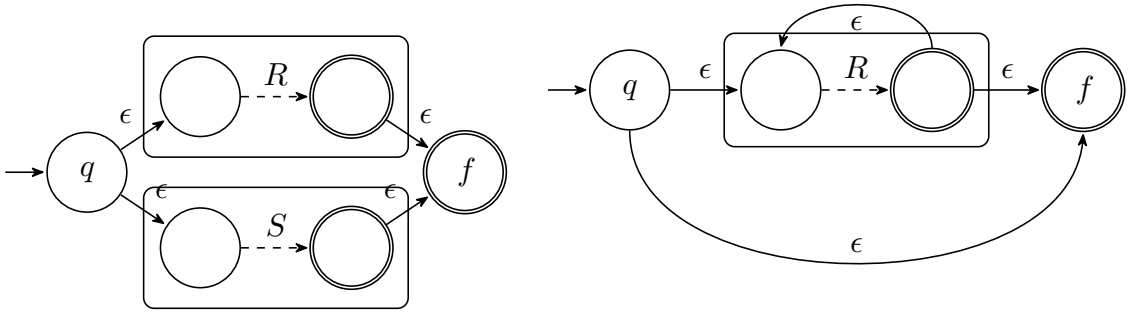


(a) Empty Set $\emptyset$ (b) Empty String ϵ (c) Character $a \in \Sigma$

Figure 2.1: Constructing Finite Automata from Regular Expression Constants



(a) Concatentation



(b) Union (c) Closure

Figure 2.2: Constructing Finite Automata using Regular Expression Operations

- $R*$ denotes the Kleene closure of R, i.e., the set of strings created by concatenating, possibly with repetition, a finite number of strings in R.

The class of languages expressed by regular expressions are the *regular languages*: exactly the same class of languages as accepted by finite automata. In fact, it is possible to convert a regular expression defined using the above constructs to an $\epsilon-\mathrm{NFA}$ using *Thompson's Construction* (as shown in Figure2.2). First, the constants are converted to the equivalent automata according to the rules shown in Figure 2.1. The operators for regular expressions can then be used to combine automata as shown in Figure 2.2.

The language defined by a regular expression is composed of all strings matching the pattern. Automata built from regular expressions can be used to check if a particular string matches the corresponding expression: if after an automaton consumes a string it is in an accepting state, then the string is in the language denoted by the

regular expression.

Practical implementations of regular expressions often extend the formal definitions given above. In particular, the POSIX standard extends the above with the character `.` (dot) to match any single character, parentheses () to delimit sub-expressions and brackets [] to match a single occurence of the characters in the brackets. Adding these constructs does not change the theoretical underpinning of regexes or their automata equivalents. For the remainder of this document, we will consider regular expressions to include these constructs.

A key insight is that paths in a network are similar to strings of characters, if we think of the alphabet Σ to be composed not of abstract characters, but of network locations. In that case, a regular expression denotes not a set of strings, but a set of *paths* through the network. For example, the expression `(h1 .* h2)` denotes all paths that start at `h1` and end at `h2`, with any other network locations in between. Similarly, the expression `(h1 .* (m1|m2|m3) .* h2)` describes all paths starting at `h1` and ending at `h1`, and passing through any one of `m1, m2` or `m3` at some point.

The remainder of this chapter describes the Merlin network programming language and how it uses the notion of regular expression as sets of network paths to define a high-level abstraction for managing modern networks.

2.2 A Network Policy Example

The Merlin policy language offers constructs for specifying the intended behavior of the network at a high level of abstraction. At the core of the language is the *path expression*: a regular expression composed of physical network locations (`l`) and abstract functions (`t`). Each path expression denotes a set of paths through the network, and is associated with a *predicate* that specifies which packets should be sent along those paths.

For example, consider a network policy in which all HTTP traffic from one network

location (h1) to another (h2) must be logged by sending it through a network logging function (log). This policy is expressed by the following Merlin language program:

```
( ip.src = h1 and ip.dst = h2 and tcp.dport = 80 ) ->
  h1 .* log .* h2 ;
```

If the policy also requires logging in the opposite direction, we can extend the Merlin program accordingly:

```
( ip.src = h1 and ip.dst = h2 and tcp.dport = 80 ) ->
  h1 .* log .* h2 ;
( ip.src = h2 and ip.dst = h1 and tcp.dport = 80 ) ->
  h2 .* log .* h1 ;
```

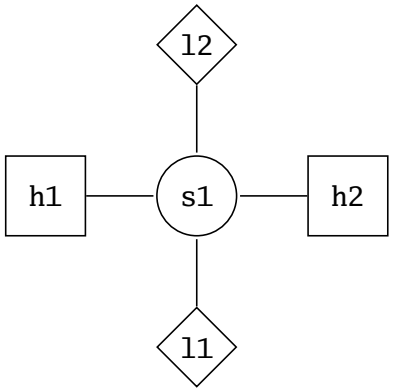
Furthermore, if the network in question carries many different classes of traffic, the policy may cap the amount of bandwidth available to HTTP traffic. This can be expressed in the program by assigning a *variable* to each class of traffic, and then writing a Presburger formula over those variables specifying the bandwidth limitation. For example, if the combined HTTP traffic should be at most 1Gb/s, we can write the following:

```
[ x: ( ip.src = h1 and ip.dst = h2 and tcp.dport = 80 ) ->
       h1 .* log .* h2 ;
  y: ( ip.src = h2 and ip.dst = h1 and tcp.dport = 80 ) ->
       h2 .* log .* h1 ; ],
max( x + y, 1Gb/s)
```

Given such a program, the compiler generates configurations for specific network devices that together implement the specified policy. To do so, the compiler needs to know the network topology, and how to implement network functions such as log.

The Merlin compiler takes as input a Merlin program, a network topology specified in the DOT graph description language, and a mapping from abstract functions to the specific network devices that may implement them.

For example, a network that might implement the above-mentioned policy is shown in Figure 2.3a. This network consists of the two hosts, connected by an SDN-capable packet switch `s1`. The switch `s1` is also connected to two network devices `l1` and `l2` that are capable of implementing the logging function `log`. These could be dedicated hardware for logging, or a programmable middlebox configured to implement logging. This mapping is communicated to the Merlin compiler as `log:={l1, l2}`.



(a) Example network for HTTP logging

```
h1[host]; h2[host]; s1[switch];
l1[middlebox]; l2[middlebox];

graph g {
  (h1) -- (s1) ;
  (h2) -- (s1) ;
  (l1) -- (s1) ;
  (l2) -- (s1) ; }

log:={l1, l2}
```

(b) DOT specification of example network

With this network and mapping the Merlin compiler is free to choose either of `l1` or `l2` to implement `log`. If they are configurable middleboxes, the compiler is capable of generating the configurations required for logging behavior. To satisfy the path expression `(h1 .* log .* h2)`, the compiler can choose any of the paths that start at `h1`, go through `s1` to `l1` or `l2`, and then back through `s1` to `h2`. This requires generating the appropriate forwarding rules for `s1` (assuming that `l1` and `l2` simply reflect back logged packets). Finally, to satisfy the bandwidth requirements, the compiler generates configurations for the network stacks on the end-hosts to cap HTTP traffic.

The following section describes in detail the Merlin policy language and following chapters describe the compilation process.

Metavariables

Network locations l	*Values* v
Abstract functions t	*Variables* x
Header fields f	*Traffic Rates* n

Syntax

Predicates	p ::=	$f = v \mid \mathsf{true} \mid \mathsf{false}$
	$\mid$	p_1 and $p_2 \mid p_1$ or $p_2 \mid !\, p_1$
Path expressions	r ::=	$. \mid l \mid t \mid r\, r \mid r \mid r \mid r^* \mid !\, r$
Bandwidth expressions	e ::=	$n \mid x \mid e + e$
Presburger formulas	ϕ ::=	$\mathsf{max}(e, n) \mid \mathsf{min}(e, n) \mid \phi_1$ and ϕ_2
Statements	s ::=	$x : p \rightarrow r$
Programs	m ::=	$[s_1; \ldots; s_n], \phi$

Figure 2.4: Merlin Syntax.

2.3 Design of the Merlin Network Programming Language

The design of the language is based on two observations. First, network policies are intuitively expressed by separating network traffic based on the contents of header fields. Second, paths in a network, and their associated bandwidth capacities, are key network resources and network policies often require dividing these resources between traffic classes. Thus, the Merlin language allows network traffic to be divided into named classes using predicates on packet headers. Each such class can then be assigned a set of paths that all traffic in that class is allowed to take. These sets of paths are defined using path expressions, as shown in the previous section. Finally, bandwidth resources may be divided among these classes using formulas expressed using Presburger arithmetic.

The syntax of the Merlin policy language is defined by the grammar in Figure 2.4. High-level network policies can be expressed as Merlin programs. A Merlin program m is a set of *statements*, each of which specifies the handling of a subset of traffic, together with a *logical formula* that expresses a global bandwidth constraint. For simplicity, we require that the statements in a particular program have disjoint predicates and

together match all packets. These requirements are enforced by the Merlin compiler.

Statements Each program statement is composed of several components: a *variable*, a *logical predicate*, and a *path expression*. The variable identifies the set of network packets matching the predicate, and the path expression specifies the forwarding paths and packet-processing functions that should be applied to matching packets. These abstractions allow administrators a great deal of flexibility in specifying their policies.

Administrators can start by treating the entire network as a single switch that forwards traffic between its external ports (i.e., a "big switch" [41]). This can be written as `true -> (src .* dst)`, where `src` and `dst` denote source and destination locations. This program can be refined by either dividing the predicate into finer traffic classes, or by reducing the set of permissible paths by further constraining the path expression.

Logical predicates Merlin supports a predicate language for classifying packets. Atomic predicates of the form $f = v$ denote the set of packets whose header field f is equal to v. Predicates can be combined using conjunction (`and`), disjunction (`or`), and negation (`!`).

For instance, a predicate such as `(ip.src = h1 and tcp.dport = 80)` matches packets with the same `ip` source address as `h1` and `tcp` destination port `80`. Merlin provides a number of conveniences for writing predicates. The compiler recognizes atomic predicates for a number of standard protocols including Ethernet, IP, TCP, and UDP, and a special predicate for matching packet payloads. Also, if a topology specifies IP and MAC addresses for network locations, the compiler will use the corresponding values when the locations are used in predicates. For example, if `h1` has IP address `10.0.1.1`, the above predicate will be interpreted as `(ip.src = 10.0.1.1 and tcp.dport = 80)`.

Path expressions Merlin programmers specify forwarding paths using a syntax similar to regular expressions. Rather than matching strings of characters, as with stan-

dard regular expressions, path expressions match sequences of locations (`l`) or network functions (`t`), as described below. As with POSIX regular expressions, the dot symbol (`.`) matches an arbitrary path element. The compiler is free to select any matching path, provided the other constraints in the program are satisfied. We assume that the set of locations is finite and is a subset of the network topology supplied during compilation.

Network functions Path expressions may contain names of network functions that transform the headers and contents of packets—e.g., deep packet inspection, logging, network address translation, content caching, proxying, traffic shaping, etc. The compiler determines the locations where each function can be enforced, using a mapping from function names to possible locations supplied as an input.

Network functions must take a single packet as input and generate zero or more packets as output, and they must only access local state. The restriction to local state allows the compiler to freely place functions without accounting for global state.

Network functions may modify packet headers—e.g., a network address translation NAT function may rewrite IP addresses and port numbers. To allow such functions to coexist with predicates on packet headers that identify sets of traffic, Merlin uses a tag-based routing scheme: all traffic matching a particular predicate gets attached with a VLAN (Virtual Local Area Network) tag that is then used as the matching predicate in the interior of the network.

Bandwidth constraints Merlin programs use logical formulas to specify constraints that either limit (`max`) or guarantee (`min`) bandwidth. In addition to conjunction (`and`), disjunction (`or`), and negation (`!`), Merlin supports an addition operator. The addition operator can be used to specify an aggregate cap on traffic, such as `max(x + y, 1Gb/s)`. By convention, programs without a rate clause are unconstrained—programs that lack a minimum rate are not guaranteed any bandwidth, and those that lack a maximum rate

may send traffic at rates up to line speed. Bandwidth constraints are expressed formally using first-order logic with addition—a fragment known as Presburger arithmetic. Note that excluding multiplication ensures decidability.

Syntactic sugar Merlin supports several forms of syntactic sugar that simplify the expression of complex policies such as set comprehensions. For example, the following program is equivalent to the example program used previously in Section 2.2.

```
hosts := {h1, h2}
forall (s,d) in cross_distinct(hosts,hosts):
  ip.src = s and ip.dst = d and tcp.dport = 80 ->
  s .* log .* d
  at max(1Gbps)
```

The notation `hosts := {h1, h2}` defines a set of network locations composed of `h1` and `h2`. The `cross_distinct` operator takes the cross product of these sets, removing elements where both components are the same (`(h1,h1)` and `(h2,h2)` in this case).

The `forall` statement iterates over the resulting set, and generates a fresh statement for each pair where the elements of the pair replace the source `s` and destination `d`. Two such statements would be generated in this case, one each for `(h1,h2)` and for `(h2,h1)` and each would be associated with a unique variable (say `x` and `y`). A bandwidth expression is generated by summing over the variables and limiting that sum with the bandwidth term specified above: `max(x + y, 1Gbps)`. Thus the above program generates the same program as in Section 2.2:

```
[ x: ( ip.src = h1 and ip.dst = h2 and tcp.dport = 80 ) ->
       h1 .* log .* h2 ;
  y: ( ip.src = h2 and ip.dst = h1 and tcp.dport = 80 ) ->
       h2 .* log .* h1 ; ],
max( x + y, 1Gb/s)
```

There is also a `foreach` construct that has the same effect, except that each variable is constrained separately. In this case, using `foreach` would generate a bandwidth term `max(x, 1Gbps) and max(y, 1Gbps)`.

Limitations While the Merlin language can express a wide range of policies, there are some key limitations. First, network functions are treated as abstract and required to use only local state. Thus, complicated network functions which may have multiple components, such as a distributed intrusion detection system, must be placed manually in a network by specifying the locations of individual physical devices. Since Merlin does not model changes to headers a network function may make (such as a NAT box), users must ensure that such changes do not interfere with whatever tagging scheme that Merlin uses. Finally, bandwidth constraints must be expressed using only `min`, `max` and addition operators. This makes it difficult to express complicated regulations between the bandwidth usage of different classes (eg., that one class use only a certain fraction of another class' usage).

Summary Merlin enables direct expression of high-level network policies. Programmers write policies as though they were centralized programs executing on a single device. In reality, a variety of distributed devices collaborate to collectively enforce the policy. The next section presents Merlin's compilation techniques for path expressions.

2.4 Compiling Path Expressions

Implementing Merlin program atop a physical network requires the Merlin compiler to perform three key tasks:

1. **Localization:** A Merlin program describes the global behavior of the network. This behavior must be translated into per-device, locally-enforceable policies.

2. **Path selection and bandwidth allocation:** The Merlin language provides administrators two facilities for controlling what paths through the network certain classes of traffic may take: path expressions and bandwidth allocation. Furthermore, each statement has its own requirements that might affect what paths are available to traffic mentioned in other statements. It is up to the compiler to balance these requirements and allocate paths to traffic classes (if possible).
3. **Code generation:** Finally, once a suitable set of paths has been determined, the compiler must generate low-level configuration instructions for network devices and end hosts. Since networks might be composed of a variety of devices, the compiler must support multiple backends.

The input to the compiler is the Merlin program, physical topology, and a mapping from network functions to possible placements. These are used to build a *logical topology* incorporating the structure of the physical topology and the constraints encoded by the program statements. The compiler analyzes this logical topology to assign paths to traffic classes and allocate bandwidth on those paths. Finally, the path information is used to generate low-level configurations for SDN-capable switches, middleboxes, and end hosts.

Note that localization is only an issue when multiple traffic classes share a bandwidth allocation. Without a bandwidth constraint, the compiler is only required to satisfy the path expression for each statement. The compiler checks that the traffic classes described by the predicates in a program are disjoint, and so there is no possibility of conflict between different statements. The remainder of this section focuses on handling path expressions. The techniques described here are extended to include bandwidth constraints in the following chapter.

2.4.1 Building a Logical Topology

Each program statement contains a predicate and a path expression. The path expression constrains the set of forwarding paths that packets satisfying the predicate might take. To compute paths satisfying these constraints, the compiler constructs a *logical topology* as a directed graph $\mathcal{G}$ in which each path corresponds to a physical path that satisfies the path expression for some statement in the program. The overall graph $\mathcal{G}$ for the program is a union of disjoint components $\mathcal{G}_i$, one for each statement.

The first step in constructing $\mathcal{G}_i$ is to ensure that the corresponding path expression is over network locations *only*. Each statement i has a regular expression r_i over the set of both locations and packet-processing functions. Each such r_i is transformed into a path expression $\bar{r}_i$ over the set of locations only using a simple substitution. Since the compiler takes an auxiliary input that maps functions to network locations, each such function is replaced with the union of all locations that function could be mapped to. For example, if `h1`, `h2`, and `m1` are the three locations capable of running a Deep Packet Inspection function (`dpi`), then the regular expression `(.* dpi .*)` would be transformed into `(.* (h1|h2|m1) .*)`.

The next step is to transform the regular expression $\bar{r}_i$ into a deterministic finite automaton (DFA), denoted $\mathcal{A}_i$, that accepts the set of strings in the regular language given by $\bar{a}_i$. The transformation is performed using the standard algorithms [1] described in Section 2.1 and depicted in Figures 2.1 and 2.2. With this $\mathcal{A}_i$ we can now construct $\mathcal{G}_i$.

Let L denote the set of locations in the physical network and $\mathcal{Q}_i$ denote the state set of $\mathcal{A}_i$. The vertex set of $\mathcal{G}_i$ is the Cartesian product $L \times \mathcal{Q}_i$ together with two special vertices, s_i and t_i, respectively representing a universal source and sink for paths generated for statement i.

Between two states (u, q) to (v, q') of $\mathcal{G}_i$ the graph $\mathcal{G}_i$ has an edge iff:

1. $u = v$ or (u, v) is an edge of the physical network, and
2. there is a transition in $\mathcal{A}_i$ from q to q' labeled with v

Likewise, there is an edge from source state s_i to (v, q') iff (q^0, q') is a valid state transition of $\mathcal{A}_i$ labeled with v (where q^0 denotes the start state of $\mathcal{A}_i$). Finally, there is an edge from (u, q) to the sink state t_i iff q is an accepting state of $\mathcal{A}_i$.

$\mathcal{G}_i$ has been constructed such that paths in $\mathcal{G}_i$ correspond to paths in the physical network that satisfy the path constraints of statement i. This property is captured in the following lemma:

Lemma 1. *A sequence of locations $u_1, u_2, \ldots, u_k$ satisfies the constraint described by the path expression $\bar{r}_i$ iff $\mathcal{G}_i$ contains a path of the form $s_i, (u_1, q_1), (u_2, q_2), \ldots, (u_k, q_k), t_i$ for some sequence of states $q_1, \ldots, q_k$. A path of this form is a "lifting" of $u_1, u_2, \ldots, u_k$.*

Proof. The construction of $\mathcal{G}_i$ ensures that

$$s_i, (u_1, q_1), (u_2, q_2), \ldots, (u_k, q_k), t_i$$

is a path in the graph iff (i) the sequence $u_1, \ldots, u_k$ represents a path in the physical network (possibly with vertices of the path repeated more than once consecutively in the sequence), and (ii) the automaton $\mathcal{A}_i$ has an accepting computation path for $u_1, \ldots, u_k$ through the sequence of states $q^0, q^1, \ldots, q^k$. The lemma follows from the fact that a string belongs to the regular language defined by $\bar{r}_i$ if and only if there is a path through $\mathcal{A}_i$ that accepts that string. □

Figure 2.5 shows the construction of the graph $\mathcal{G}_i$ for a statement with path expression `(h1 .* dpi .* nat .* h2)`, on an example network. For this example, deep packet inspection (`dpi`) can be performed at `h1`, `h2`, or `m1`, but network address translation (`nat`) can only be performed at `m1`. The thick, red path illustrates one lifting of an accepting path in $\mathcal{A}_i$ to $\mathcal{G}_i$. Notice that the physical network also contains other

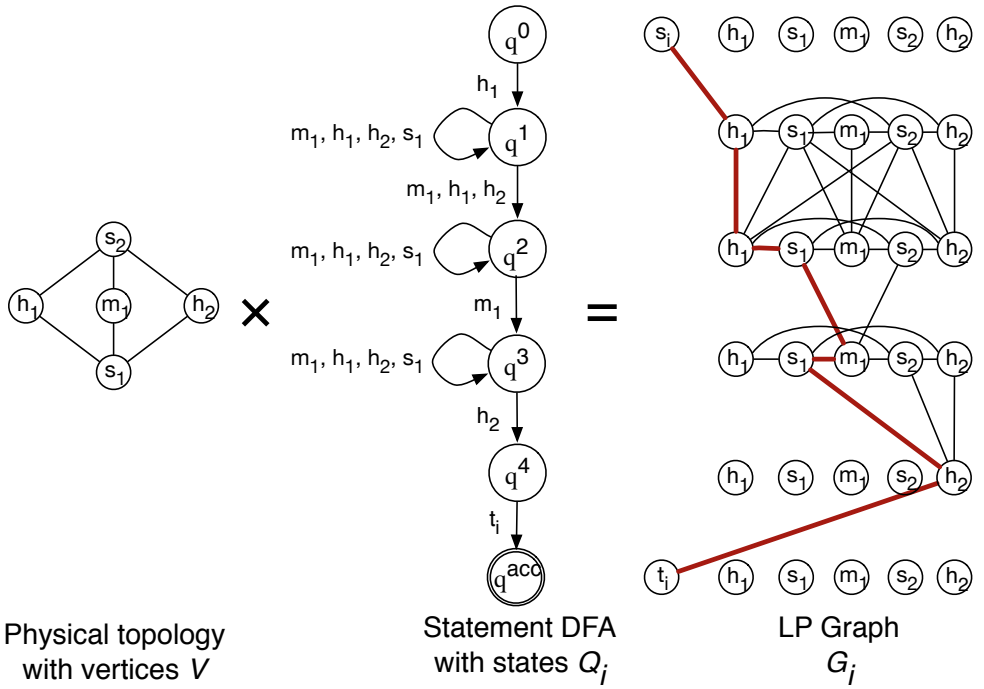


Figure 2.5: Example logical topology and a possible solution.

paths (such as `h1–s1–h2`) that do not satisfy the path expression. These paths do not lift to any path in $\mathcal{G}_i$. For instance, consider the rows of nodes corresponding to states q^2 and q^3 of the NFA. All edges between these rows lead into node (m_1,q^3). Thus, any path avoiding `m1` in the physical network cannot be lifted to an s_i–t_i path in $\mathcal{G}_i$.

2.4.2 Path Selection

The logical topology $\mathcal{G}_i$ for each statement has the property that each path from s_i to t_i through it satisfies the corresponding path expression r_i. To find a path through the network for the corresponding traffic class, the Merlin compiler computes $\mathcal{G}_i$ as described above, and then performs a breadth-first search over the resulting graph.

For statements whose path expressions differ only in the end-hosts (such as the example in Section 2.2) we can perform an optimization. Instead of generating an $\mathcal{G}_i$ for each such statement, the compiler generates a single $\mathcal{G}_i$ from a network topology

that includes only switches, and a path expression without the end-hosts. Then the compiler computes an *optimal sink tree* for each egress switch—a tree rooted at the egress switch and containing only the edges required for shortest-length paths from all the ingress switches to that egress switch. Based on the predicates, the compiler adds instructions to forward traffic from hosts to ingress switches and then from the egress switches to the hosts during code generation. Using sink trees, the solution can be computed in $O(|V||E|)$, where $|V|$ is the number of switches rather than the number of hosts.

2.4.3 Code Generation

Merlin enables network administrators to write high-level programs without worrying about how those policies are implemented. The Merlin compiler uses *program partitioning* to transform the program into separate programs, instructions, and configurations that are deployed across the following classes of devices.

- *Switches.* Merlin generates configurations for SDN-capable network switches using the OpenFlow [53] libraries provided by the Frenetic SDN Controller [24]. For bandwidth enforcement, Merlin uses the min-rate queues defined in OpenFlow specification version 1.0, and device-specific port queue configurations.
- *Middleboxes.* For functionality such as deep packet inspection, load balancing, and intrusion detection, Merlin generates configuration scripts for Click [45] that define the sequence of packet-processing functions to apply. Other approaches are possible—e.g., Merlin could generate Puppet [63] scripts to provision and manage virtual machines instead.
- *End hosts.* Traffic filtering and rate limiting are implemented using standard Linux utilities (`iptables` and `tc`).

Tag-based routing Because Merlin controls forwarding paths but also supports packet-processing functions that may modify headers (such as NAT boxes), the compiler must use a forwarding mechanism that is robust to changes in packet headers. The current implementation uses one VLAN tag per sink tree to encode paths to destination switches. All packets destined for a given destination are tagged with a tag when they enter the network. Subsequent switches simply examine the tag to determine the next hop. At the egress switch, the tag is stripped off and replaced with a unique identifier for the host (e.g., the MAC address). Similar approaches are used in other systems for combining programmable switches and middleboxes such as FlowTags [22].

The Merlin compiler is designed with flexibility in mind and can be easily extended with additional backends that capitalize on the capabilities of the carious devices available in the network. Although the expressiveness of policies is bounded by the capabilities of the devices, Merlin provides a unified interface for programming them.

2.5 Summary

Regular expressions are a well-studied formalism for specifying sets of strings of characters. In the context of network programming, they denote paths through a network. These path expressions form the basis of the Merlin network programming language. With Merlin, administrators express network policies as programs in a high-level language that uses logical predicates to identify sets of packets, regular expressions to encode forwarding paths, and arithmetic formulas to specify bandwidth constraints.

Path expressions are interpreted as finite automata, that the compiler uses to build a logical topology, whose paths correspond to paths through the network that satisfy the path expression. This logical topology is used to derive paths that different classes of traffic may take through the network. Finally, the compiler generates code that can be executed on the network elements to enforce the policies.

CHAPTER 3

BANDWIDTH ALLOCATION

The previous chapter describes how Merlin uses predicates on packet headers and path expressions over network locations and abstract functions to give network administrators a great deal of flexibility in how traffic traverses their network. However, modern network administration involves more than just deciding what traffic takes what routes. In particular, network bandwidth is a limited resource that must be managed carefully and allocated between different traffic classes, paths and users. The discussion of Merlin syntax in Section 2.3 showed briefly how network administrators can assign a bandwidth constraint to groups of statements using Presburger arithmetic formulas. This chapter discusses how Merlin's bandwidth constraints may be used in practice and expands the discussion of compilation techniques in Section 2.4.

3.1 A Bandwidth Allocation Example

For the purposes of this example, assume a network where hosts can communicate with each other via both HTTP and FTP protocols. The HTTP protocol uses TCP port 80, while FTP uses TCP port 20 for transferring data and TCP port 21 for control commands. In such a network, HTTP may be used for real-time website traffic and must always have some minimum share of the network bandwidth to process requirements in a reasonable amount of time. On the other hand, FTP may be used for long-running backup processes. These jobs do not have a strict timing requirement, and so should never take up more than a certain amount of bandwidth, in order to not degrade the website performance. These requirements constitute a policy that can be implemented as a Merlin program.

To implement the above policy, we can place a bandwidth cap on FTP traffic, and provide a bandwidth guarantee to HTTP traffic. Each statement in the program be-

low consists of a variable that represents the bandwidth used by matching packets, a predicate on packet headers that identifies the set of matching packets, and a regular expression that describes the set of legal forwarding paths:

```
[ x : (ip.src = 10.0.1.1 and ip.dst = 10.0.1.2 and
       tcp.dport = 20) -> .* ;
  y : (ip.src = 10.0.1.1 and ip.dst = 10.0.1.2 and
       tcp.dport = 21) -> .* ;
  z : (ip.src = 10.0.1.1 and ip.dst = 10.0.1.2 and
       tcp.dport = 80) -> .* dpi *. nat .* ],
  max(x + y,100Mb/s) and min(z,900Mb/s)
```

The first two statements (for variables x and y) assert that FTP traffic (both control and data) can travel from the host at IP address 10.0.1.1 to the host at address 10.0.1.2 across any available network path. The statement for variable z identifies and constrains HTTP traffic between the same hosts. However, for HTTP traffic, the set of paths are restricted to those that include both a deep-packet inspection function (dpi, possibly to check for malicious traffic) and a network address translation (nat) function. Finally, the formula at the end declares a joint bandwidth cap (max) for the FTP traffic, and a bandwidth guarantee (min) for the HTTP traffic. The amounts are chosen such that they add up to the capacity of a 1Gb/s physical link.

3.2 Compiling Bandwidth Allocations

As described in Section 2.3, Merlin programs use logical formulas to specify bandwidth constraints that limit (max) or guarantee (min) bandwidth. Individual classes of traffic may be given a joint allocation by using the addition operator (+) to sum over their respective variables, and then constrain the result. Allocations may be combined using conjunction (and), disjunction (or), and negation (!).

A formula specifies the rate at which sources of various types of traffic may emit packets. Formally, the universe of rates is `[0, MAX]` where `MAX` is the line speed dependent on physical constraints. `max(x, 100Mbps)` constrains the rate of `x` traffic to be in the interval `[0, 100Mbps]`, whereas `min(x, 100Mbps)` constrains rate of `x` traffic to be within `[100Mbps, MAX]`, (assuming the source is transmitting at `100Mbps` or higher).

Bandwidth constraints differ from locations and functions in one important aspect: they represent an explicit allocation of global network resources. The first step in enforcing the bandwidth allocations is to translate the global requirements to local requirements that can be enforced on individual devices The first step in enforcing the bandwidth allocations is to translate the global requirements to local requirements that can be enforced on individual devices.

3.2.1 Localization

Global bandwidth constraints are expressed as Presburger arithmetic formulas. However, implementing them leads to several challenges: aggregate guarantees can be enforced using shared quality-of-service queues on network switches, but aggregate limits are more difficult, since they require distributed state in general. To solve this problem, Merlin adopts a pragmatic approach. The compiler first rewrites the formula so that bandwidth constraints apply to packets at a single location. Given a formula with one term over n identifiers, the compiler produces a new formula of n local terms that collectively imply the original. By default, the compiler divides bandwidth equally among the local terms, although other schemes are permissible.

For example, the formula in the previous section uses `x` and `y` to refer to FTP traffic and to `z` refers to HTTP traffic, and requires the following guarantees:

```
max(x + y,100Mb/s) and min(z,900Mb/s)
```

The Merlin compiler first rewrites this to a form where `x` and `y` have separate guarantees

that can be enforced independently:

```
max(x,50Mb/s) and max(y,50Mb/s) and min(z,900Mb/s)
```

Rewriting programs in this way involves a tradeoff: localized enforcement increases scalability, but risks underutilizing resources. Merlin navigates this tradeoff via run-time mechanisms called *negotatiors* that allow users of a network to dynamically adjust allocations to their needs. For example, a negotiator for an FTP-based network backup service might know that FTP control traffic (associated with y above) requires much less bandwidth than FTP data traffic (associated with x). Such a negotiator could decide that a better split for the FTP allocation would be:

```
max(x,99Mb/s) and max(y,1Mb/s) and min(z,900Mb/s)
```

Chapter 4 describes how the design of the Merlin programming language allows negotatiors to refine Merlin programs and allows the compiler to verify that a refined program does not violate the guarantees of the original.

3.2.2 Provisioning Bandwidth Allocations

Once global guarantees in a Merlin program have been localized, the next step is to provision for the required bandwidth allocation. To do this, the Merlin compiler encodes the input program and the topology into a constraint problem whose solution can be used to determine device configurations.

Logical Topology The first step in this process is the same as described in Section 2.4.1: creating a logical topology from the input Merlin program, physical network topology and function mapping. Recall that this logical topology is a graph $\mathcal{G}$ that is the union of several smaller graphs $\mathcal{G}_i$, one for each statement i in the original Merlin program. Each $\mathcal{G}_i$ is a cross product of the physical network topology and the

NFA derived from the expanded path expression $\bar{r}_i$ from the statement i, with a distinguished source and sink nodes s_i and t_i. This cross product construction ensures that each path from a source s_i to a sink t_i corresponds to a path in the physical topology that satisfies the corresponding path expression $\bar{r}_i$. Such a path in $\mathcal{G}_i$ is called a *lifting* of the corresponding path in physical topology.

Path selection Next, the compiler determines an assignment of paths that satisfy the bandwidth constraints encoded in the policy. The problem is similar to the well-known *multi-commodity flow problem* [2], with two additional types of constraints:

1. *integrality constraints* specify that only one path is selected per statement; and
2. *path constraints* as determined by the corresponding path expressions

Since the logical topology gives us a way to encode path constraints, we formulate the problem in the graph $\mathcal{G} = \bigcup_i \mathcal{G}_i$ described above, rather than in the physical network. Unfortunately, incorporating integrality constraints renders the problem NP-complete in the worst case. However, several practical approaches have been developed, ranging from approximation algorithms [13, 16, 19, 44, 46], to specialized algorithms for expanders [11, 26, 43] and planar graphs [60], to the use of mixed-integer programming [9]. The following technique is based on the mixed-integer programming approach.

The compiler generates a Mixed-Integer Program (MIP) having a $\{0, 1\}$-valued decision variable x_e for each edge e of $\mathcal{G}$. Selecting a route for each statement corresponds to selecting a path from s_i to t_i for each i and setting $x_e = 1$ on the edges of those paths. For all other edges in $\mathcal{G}$, $x_e = 0$. These variables are required to satisfy the

following flow conservation equations:

$$\forall v \in \mathcal{G} \sum_{e \in \delta^+(v)} x_e - \sum_{e \in \delta^-(v)} x_e = \begin{cases} 1 & \text{if } v = s_i \\ -1 & \text{if } v = t_i \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

$\delta^+(v)$, $\delta^-(v)$ denote the sets of edges exiting and entering v, respectively. The MIP also has real-valued variables r_{uv} for each physical network link (u, v), representing what fraction of the link's capacity is reserved for statements whose assigned path traverses (u, v). Finally, there are variables $r_{\max}$ and $R_{\max}$ representing the maximum fraction of any link's capacity devoted to reserved bandwidth, and the maximum net amount of reserved bandwidth on any link, respectively.

We can now write down the equations and inequalities that govern the behavior of these additional variables. For each statement i, $r^i_{\min}$ denotes the minimum amount of bandwidth guaranteed in the corresponding bandwidth clause. If the statement has no bandwidth guarantee:

$$r^i_{\min} = \begin{cases} \rho & \text{if statement } i \text{ has bandwidth guarantee } \rho \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

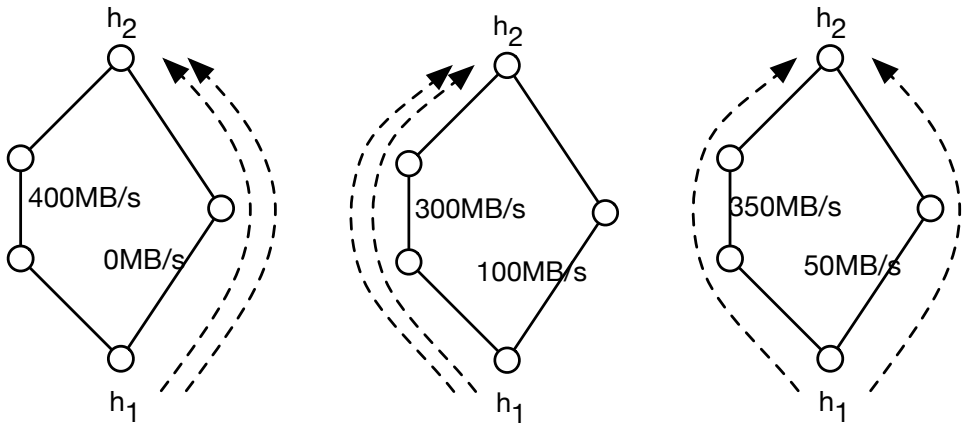
For any physical link (u, v), c_{uv} denotes its capacity and $E_i(u, v)$ denotes the set of all edges in $\mathcal{G}_i$ that connect nodes corresponding to the physical nodes u and v. (Formally, edges of the form $((u, q), (v, q'))$ or $((v, q), (u, q'))$ in $\mathcal{G}_i$.) The relevant constraints can be formalized as:

$$\forall (u, v) \;\; r_{uv} c_{uv} = \sum_i \sum_{e \in E_i(u,v)} r^i_{\min} x_e \tag{3.3}$$

$$\forall (u, v) \;\; r_{\max} \geq r_{uv} \tag{3.4}$$

$$\forall (u, v) \;\; R_{\max} \geq r_{uv} c_{uv} \tag{3.5}$$

$$r_{\max} \leq 1 \tag{3.6}$$



(a) Shortest-Path (b) Min-Max Ratio (c) Min-Max Reserved

Figure 3.1: Path selection heuristics. The edge labels in the graphs indicate the remaining capacities after path selection.

Constraint 3.3 defines r_{uv} to be the fraction of capacity on link (u, v) reserved for bandwidth guarantees. Constraints 3.4 and 3.5 ensure that $r_{\max}$ (respectively, $R_{\max}$) is at least the maximum fraction of capacity reserved on any link (respectively, the maximum net amount of bandwidth reserved on any link). Constraint 3.6 ensures that the path assignment will not exceed the capacity of any link, by asserting that the fraction of reserved capacity does not exceed 1.

Path selection heuristics There may be multiple assignments that satisfy the path and bandwidth constraints. The Merlin compiler provides network administrators with three heuristics to guide the path selection process:

- *Weighted shortest path* minimizes the total number of hops in selected paths weighted by bandwidth guarantees: $\min \sum_i \sum_{u \neq v} \sum_{e \in E_i(u,v)} r^i_{\min} x_e$. This heuristic is for minimizing latency.
- *Min-max ratio* minimizes the maximum fraction $r_{\max}$ of capacity reserved on any link. This heuristic is appropriate for balancing load across links.

- *Min-max reserved* minimizes the maximum amount of bandwidth reserved on any single link (i.e., $R_{\max}$). This heuristic guards against failures, since it limits the maximum amount of traffic that may be disrupted by a single link failure.

Figure 3.1 shows the differences between these heuristics. The figure depicts a network with hosts `h1` and `h2` connected by two disjoint paths. The left path has three edges of capacity 400MB/s while the right path has two edges of capacity 100MB/s. Suppose that two statements each request 50MB/s of guaranteed bandwidth (i.e., `min(x1, 50MB/s)` and `min(x2, 50MB/s)`). The MIP solver will either select two-hop paths (weighted shortest path), reserve no more than 25% of capacity on any link (min-max ratio), or reserve no more than 50MB/s on any link (min-max reserved).

The integrality constraint specified above forces the Merlin compiler to produce solutions that use a single path for each traffic class. While there exist approaches to multi-commodity flow that take advantage of multiple paths, we leave this extension as a topic for future work.

Code generation The output from the MIP solver can be interpreted as a single path through for each class of traffic (if a solution to the problem can be found). Also, recall that the localized version of the Merlin program associates each statement (or rather, its corresponding variable) with a single bandwidth allocation. Thus the final step in the compilation process is to match up the solution paths with the requested bandwidth allocation in the Merlin program.

Concretely, a careful selection of variable names in the MIP allows the compiler to match up selected edges in the solution with the corresponding variable and statement in the localized Merlin program. The compiler then refers to the program to determine the bandwidth allocation for those edges.

Finally the compiler generates instructions for all network devices on a path that enforces the required bandwidth allocation. Minimum allocations must be enforced on

each SDN-capable switch on a path. Merlin uses the min-rate queues defined in version 1.0 of the OpenFlow specification version, and generates device-specific instructions to configure the relevant queues and ports. For maximum caps, rate limits are enforced directly on the end-hosts using standard Linux utilities such as `iptables` and `tc`.

3.3 Summary

The Merlin language allows administrators to write global network policies that distribute the bandwidth resources of a network between various traffic classes. In order to implement such allocations, the Merlin compiler must localize such policies so that they can be implemented efficiently on individual devices, and then select paths for the relevant traffic classes such that all requested allocations are satisfied.

The previous chapter detailed how a logical topology can be used to satisfy the path expressions in a Merlin program. This chapter shows how the compiler uses that logical topology, and a set of intuitive constraints and heuristics, to craft a MIP whose solution is a selection of paths that satisfy both the path expressions and bandwidth constraints. Finally, the compiler matches these selected paths with the required allocation in the original Merlin program to generate relevant configurations for the network devices that must enforce the bandwidth allocations.

A key component of this process is the transformation of global constraints such as "`max(x + y, 10Mb/s)`" to equivalent local constraints such as "`max(x, 10Mb/s)` and `max(y,5Mb/s)`". The compiler uses a simple equal division strategy to perform such localization in the general case. However, it is possible that such a division will be inefficient and that some domain knowledge can be applied to improve the efficiency tradeoff. The next chapter describes how Merlin policies can be safely delegated to network users who can then refine them according to their individual needs, without breaking any guarantees provided by the original.

CHAPTER 4

DELEGATION AND VERIFICATION

The first step in the compilation process described in the preceding chapter is to translate global Merlin programs into local programs. However, a naïve localization process (say by just distributing bandwidth allocations evenly) is clearly inefficient. Such an allocation may underutilize resources by giving too much of a share to some traffic classes, and too little to others. Additionally, in the dynamic environment of a network, a localization that was originally adequate may become stale and inefficient as traffic demands evolve over time.

Moreover, in a shared environment, such as a campus network or a shared datacenter, users may wish to customize global policies to suit their needs. These users—tenants henceforth—might have particular desires for how to divide their share of network bandwidth, imposing constraints on how the localization process occurs. They may also wish to constrain what paths their own traffic takes, perhaps to enforce security policies, or isolation between different services and traffic classes.

The realities of managing a modern network suggest that in addition to a well-defined language for expressing network policy, we require some mechanism for dynamism and adaptation, both to changing network conditions, as well as to differing administrative and application domains. A suitable mechanism should support both *delegation* and *verification*. As noted earlier, in a shared network, tenants might have a better understanding of their network needs, and a desire to enforce their own policies. This suggests a need for *delegation*: tenants should be able to manage their own share of the network, as long as they do not affect other tenants or the network owner. Enforcing that condition requires *verification*: the ability to check that a Merlin program that has been refined by a tenant does not violate any guarantees of the original.

This chapter first describes the notion of *negotiators*—run-time components that fa-

cilitate program delegation to tenants—and then details how the properties of the Merlin language support verification of the resulting refined programs against an original.

4.1 Negotiators

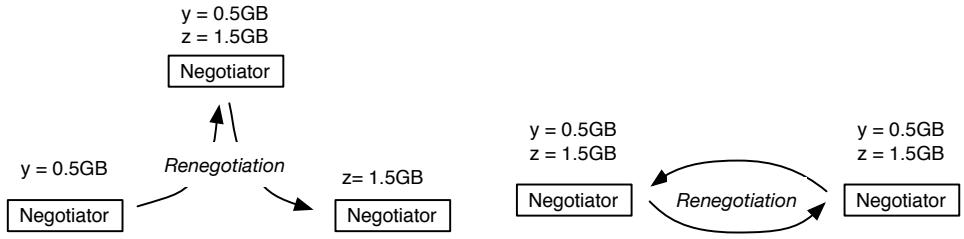
To support dynamic modification of programs, Merlin uses small run-time components called *negotiators*, which transform and verify programs. Negotiators allow program refinement to be delegated to tenants and they provide mechanisms for verifying that choices made by tenants do not lead to violations of the original program. Negotiators depend critically on Merlin's language-based approach—the abstractions used to express network policies (i.e., predicates, regular expressions, and bandwidth constraints), also make it easy to support verifiable program transformations.

4.1.1 Negotiator Overlays

Negotiators are distributed throughout the network in a tree, forming a hierarchical overlay over network elements. Each negotiator is responsible for the network elements in the subtree for which it is the root. Parent negotiators impose programs on their children. Since a negotiator cannot affect programs upwards in the tree, it is limited to either refining its inherited program for its children, or negotiated specifics of the inherited program with its siblings.

Children may refine their own programs, as long as the refinement implies the parent program. This involves either more finely subdividing traffic classes, constraining the set of usable paths, or redistributing a bandwidth allocation amongst its own children. The following section describes the possible refinements in greater detail.

A more interesting case is that of multiple sibling negotiators jointly modifying a policy inherited from its shared parent. Particularly, siblings may renegotiate resource assignments cooperatively, as long as they do not violate parent programs. For exam-



(a) Broker-based. (b) Peer-to-peer.

Figure 4.1: Broker-based and peer-to-peer re-negotiation.

ple, siblings may dynamically adjust bandwidth allocations between themselves to fit particular deployments and traffic demands. The allocations can be adjusted in two ways: a central negotiator (possibly the parent) can act as a broker (Figure 4.1a), or between themselves in a peer-to-peer fashion (Figure 4.1b). Knowledge revealed during negotiation is limited to information about the participants and the global program. Choosing between the broker and peer-to-peer strategies is a tradeoff between performance and privacy. The broker approach is privacy-preserving (siblings cannot know about each others' allocations), but adds the overhead of working through the broker. By contrast, peer-to-peer negotiations avoid the need of a broker, but requires siblings to revealing information to each other.

Merlin does not specify protocols for reaching agreement on new allocations as the details of such a protocol depend on a variety of factors, such as the trust relationships between tenants, and the tolerance for time spent reaching a consensus. These are exogenous concerns better handled outside of the core system. The system evaluation in Chapter 7 uses a peer-to-peer negotiator, and a simple protocol that assumes cooperative peers requesting reallocations in the collective best-interest.

4.2 Valid Refinements

With negotiators, tenants can transform global network programs by *refining* the delegated programs to suit their demands. Tenants may modify programs in three ways:

1. The traffic class for one statement may be partitioned into finer classes.
2. The set of forwarding paths that a particular class of traffic may take may be further constrained.
3. The bandwidth allocations may be reduced or redistributed.

Traffic class partitioning Merlin programs classify packets into sets using predicates that combine matches on header fields using logical operators. These sets can be refined by introducing additional constraints to the original predicate. For example, consider a statement that matches all IP and sends it from one host to another, while pasing through one of a number of middleboxes (`m1` through `m3`):

```
x: eth.proto = ip -> h1 .* (m1|m2|m3) .* h2
```

An example of a valid refinement is one that separates out UDP and TCP traffic and sends the matching through separate middleboxes:

```
x1: eth.proto = ip and ip.proto = tcp -> h1 .* m1 .* h2 ;
x2: eth.proto = ip and ip.proto = udp -> h1 .* m2 .* h2 ;
x3: eth.proto = ip and ip.proto != tcp and ip.proto != udp
    -> h1 .* m3 .* h2 ;
```

A refinement of this form is valid if the refined program meets two criteria: First, the partitioning must be total—all packets matched by the original predicate must be matched by some combination of new predicates. Second, the set of paths allowed by the partitioned classes must be a subset of the set of paths allowed by the original traffic class.

Constraining paths Merlin programmers constrain what paths through a network a certain class of traffic may take by using path expressions over network locations or packet processing functions. There are two ways to constrain the path expression that reduces the set of paths that the corresponding class of traffic may take. First, if there is an alteration between a number of locations in the original expression, a refined expression may pick a subset of those locations for the same position in the expression. In the previous example, the refined statements constrain the paths allowed by specifying one middlebox out of an original set of three.

A second way to constrain the path set is by adding more functions or locations in sequence. For example, an expression that sends all packets through a traffic logger (`log`) function,

```
.* log .*
```

can be further constrained by adding a DPI function in sequence:

```
.* log .* dpi .*
```

This disqualifies all paths that include the `log` function, but not the `dpi` function later.

For a path constraint to be valid, the set of paths denoted by the new expression must be a subset of the paths denoted by the original. This ensures that all traffic stays within the bounds set by the original program.

Bandwidth Redistribution Merlin's limits (`max`) and guarantees (`min`) constrain allocations of network bandwidth. After a program has been refined, these constraints can be redistributed to improve utilization. To be valid, the sum of allocations in the refined program must not exceed the original allocation. The subtlety for redistribution is that only `min` expressions reserve bandwidth, while `max` expressions cap bandwidth usage. Therefore, it is always valid to reduce existing caps or place additional caps on

bandwidth allocation with more `max` expressions, but the sum of `min` expressions must not increase.

4.2.1 Refinement Example

As an example that illustrates the use of all three transformations, consider the following program. The original program caps all traffic between two hosts at 700MB/s:

```
[x : (ip.src = 10.0.1.1 and ip.dst = 10.0.1.2) -> .*],
 max(x, 700MB/s)
```

This program can be refined to separate out HTTP and SSH traffic and to distribute bandwidth accordingly.

```
[ x : (ip.src = 10.0.1.1 and ip.dst = 10.0.1.2 and tcp.dport = 80)
      -> .* log .* ;
  y : (ip.src = 10.0.1.1 and ip.dst = 10.0.1.2 and tcp.dport = 22)
      -> .* ;
  z : (ip.src = 10.0.1.1 and ip.dst = 10.0.1.2 and
      !(tcp.dport=22|tcp.dport=80))
      -> .* dpi .* ],
max(x, 500MB/s) and max(y, 100MB/s) and max(z, 100MB/s)
```

The refined program gives 500MB/s to HTTP traffic, but requires it to flow through a `log` box that monitors requests. 100MB/s are given to SSH traffic, and the remaining 100MB/s to all remaining traffic, which must flow through a `dpi` box. Note that since the original program does not contain a `min` expression, it would be invalid to add a `min` allocation in the refined program.

4.3 Verification

In order to allow for maximum flexibility, Merlin does not place any direct limitations on the negotiators, which perform refinements to Merlin programs. However, allowing tenants to make arbitrary modifications to programs would not be safe. For example, a tenant could lift restrictions on forwarding paths, eliminate transformations, or allocate more bandwidth to their own traffic—all violations of the original global program. Fortunately, the design of the Merlin language facilitates checking *program inclusion*, which the compiler uses to establish that refinements implemented by untrusted tenants do not violate the original policy.

Intuitively, a valid refinement of a program is one that makes it only more restrictive. To verify that a program modified by a tenant is a valid refinement of the original, the compiler has to check that for every statement in the original program, the set of paths allowed for matching packets in the refined program is included in the set of paths in the original, and the bandwidth constraints in the refined program imply the bandwidth constraints in the original.

These conditions can be decided using an algorithm that performs a pair-wise comparison of all statements in the original and modified programs and checks two conditions. First, the compiler checks for language inclusion between the regular expressions in statements with overlapping predicates. This can be performed using standard algorithms as described in [34]. Second, the compiler checks that the sum of the bandwidth constraints in all overlapping predicates implies the original constraint.

Placing a verification step in between the negotiators and the compilation process, decouples program refinement from their compilation to network devices. This allows tenants to implement negotiators however is convenient, while ensuring that there is no chance of implementing invalid programs that violate the terms of the original.

4.4 Overhead

If a refined program only re-allocates bandwidth as compared to the original, does not require the original program to be recompiled. Only the switch queue configurations and end-host `tc` commands need to be changed, and thus bandwidth re-allocation can happen quite rapidly.

Changes in path constraints or traffic classes require global recompilation. At a minimum, they require solving a fresh MIP problem and updating forwarding rules on the switches. Depending on how network functions are implemented, changing path constraints may also require regenerating middlebox configurations. However, in a realistic network deployment, changes to paths are likely to occur less frequently than changes to bandwidth allocations. For example, racks of machines in a datacenter might be tasked to a single function, but bandwidth requirements may change based on time of day. Web servers servicing requests may require more bandwidth during the day, while backup servers may get a higher allocation at night.

4.5 Summary

The design of the Merlin language allows programs to be refined by network tenants to better suit there purposes. Tenants may refine programs by partitioning traffic classes, constraining the set of allowed forwarding paths, or by reallocating available bandwidth. These refinements are performed by runtime components called negotiators that can refine inherited programs, and by coordinate with sibling negotiators. Before compiling a refined program, the compiler ensures that the guarantees of the original program are respected by checking straightforward properties of the refined program.

CHAPTER 5

HETEROGENEOUS NETWORKS

The previous chapters have described the use of a high-level programming language to manage a modern network. The Merlin programming language uses a notation similar to regular expressions to denote sets of paths through the network. These path expresssions are combined with predicates on packet headers to select classes of network traffic and with Presburger arithmetic formulas that allows administrators to specify bandwidth requirements.

Compiling Merlin programs to working network configurations requires the compiler to target a number of different devices: SDN-capable packet switches, programmable middleboxes, and end-hosts using the Linux network stack. This approach acknowledges the realities of a modern network environment: such a network is a collection of heterogeneous devices with differing capabilities and interfaces.

However, the Merlin approach is still very much SDN-centric. Recall from Chapter 1 that the core concept of SDN is to separate the data plane (tasked with forwarding packets at individual network locations) from the control plane (tasked with implementing overall network policy). The developments of the previous chapters depend on having a network of devices that have the follow properties:

1. Devices operate on invidual network packets and have access to the contents of packet headers.
2. Packets may be redirected by devices at every hop of the network, based on the contents of the packet headers.
3. Devices may be reconfigured as necessary with minimal latency.

Unfortunately, these assumptions do not hold uniformly across all classes of networks. In fact, realistic network deployments often break these assumptions in multiple ways:

1. **Legacy Devices:** Large organizations often deploy network devices supplied by multiple different vendors. SDN deployments, where they exist, tend to be partial at best. Hence, any framework for real-world network programming must be able to implement high-level policies in the presence of legacy devices.
2. **Heterogeneous Functionality:** There is a fundamental mismatch between the capabilities of SDN switches, which allow packets to be transformed in essentially arbitrary ways at each hop, and devices such as IP routers, MPLS LSRs, and optical ROADMs, which simply forward packets to their destinations
3. **Performance Limitations:** Existing frameworks tacitly assume that it is possible to rapidly reconfigure devices in response to change in network state such as policy updates, traffic shifts or topology changes. However, on current SDN switches, updating a forwarding table can take several seconds, limiting the network's adaptability.

These differences are particularly relevant in optical circuit networks that are used to connect traditional packet networks. In these networks, traffic is transported using optical channels which provide very high bandwidth (on the order of upto terabits per second) but which take on the order of several seconds to set up or reconfigure. Furthermore, unlike packet networks, where the header fields of each packet can be used to control forwarding, optical forwarding devices typically can only forward according to the frequency range occupied by a channel. Looking at the headers of the transported packets requires converting the optical signal to electronic packets and then back to an optical signal for retransmission. This optical-electronic-optical (OEO) conversion process is extremely time consuming. Naïvely recovering the flexibility of packet networks would require performing this slow OEO conversion at every hop, severely negating the performance advantages of using an optical transport.

Given the importance of optical networks, both in wide-area-networks, and in-

creasingly in datacenter networks, having flexible, high-level tools for programming optical networks is of utmost importance. The remainder of this chapter explains the properties and challenges of optical networks, and lays the foundation for enabling their high-level programmability.

5.1 Properties of Optical Networks

Modern optical networks are often used to provide high-bandwidth connectivity between traditional packet switching networks connected at their edges. For example, a cross-country optical network can be used to provide connectivity between urban networks, or between datacenters in different geographical regions. On smaller scale, recent advances in datacenter deployments have included using optical transports to connect top-of-rack switches, with each rack acting as a packet network.

In the optical core of such networks, nodes are connected to each other via optical fibers. Information is transmitted using a technique known as *wave-division multiplexing* WDM. Using WDM, multiple beams of light can be transmitted simultaneously along a single physical fiber. Each such beam has a unique peak frequency, but may stretch across a number of frequency slots. The peak frequencies are spaced far enough apart that each such beam forms a separate *channel*. This WDM technique allows for massive amounts of information to be transported on each fiber. As of 2010, experimental WDM systems are capable of carrying up to 640 channels, each at a capacity of 107 Gbit/s. Commercial systems of 16 channels with 100 Gbit/s per channel are common.

Wave division multiplexing requires a multiplexer at the source to combine multiple optical channels and a demutiplexer at the destination to separate them back out. These functions can be combined into a single physical device called an *optical add-drop multiplexer* (OADM). OADMs typically have multiple ports, each connecting an optical fiber to the OADM. On *reconfigurable optical add-drop multiplexers* (ROADMs), the opti-

cal channels that are demultiplexed from a particular port can be directed to different outgoing ports, via software reconfiguration. The rest of this document assumes the presence of ROADMs throughout the optical network.

An optical network is typically connected to multiple packet networks on the edge. These packet networks are composed of electrical switches connected to ROADMs via *transceiver ports*. When an edge packet switch sends data to the optical network, the transceiver converts the incoming electrical signals to an optical signal, occupying a single optical channel. The multiplexer on the ingress ROADM combines several such single channels and emits the resulting signal on a physical optical fiber. Conversely, on the receiver side, the egress ROADM's multiplexer separates out the optical signal from a physical fiber into the constituent channels, and emits each channel to a transceiver on a particular port. The transceiver converts the optical signal to an electrical signal for transmission to the receiving host.

Unlike flows of packets (which may copied, scheduled, dropped and routed with impunity), the optical channels and lightpaths are far more restrictive. In particular, they have the following 4 constraints:

1. **Optical continuity**: an incoming channel can be dropped or forwarded to an outgoing port, without changing the frequency slots the channel occupies.
2. **Split restriction**: an incoming channel cannot be forwarded to more than one outgoing ports.
3. **Merge restriction**: channels occupying overlapping frequency slots coming from multiple incoming ports cannot be merged to the same outgoing port.
4. **Transponder restriction**: a transponder port (that converts between electrical and optical signals) cannot input or output on more than one optical channel.

The optical channel connecting an ingress and egress ROADM is called a *lightpath*—a single-source, single-sink channel that may span multiple contiguous fiber links, but

must occupy the same set of frequency slots on each such link. Each ROADM in a multi-link lightpath demultiplexes incoming optical signals, routes the separated channels to possibly different ports and then multiplexes the signals for each port on to the connected optical fiber.

Crucially, reconfiguring an optical lightpath may take several seconds, as opposed to packet switches which can be reconfigured in milliseconds. This has important repercussions for network management, especially for latency critical applications and fault tolerance. To quickly respond to policy changes, a network management solution must minimize reconfigurations of optical lightpaths.

5.2 The Challenges of Programming Optical Networks

As an example of an optical core network connected to a packet-based edge, consider the topology in Figure 5.1. It consists of a core of ROADMs (diamonds), SDN-enabled packet switches on the edge (circles) and end-hosts connected to the switches (squares). The hosts can be considered to be standing for separate packet networks.

There are a number of challenges in properly utilizing such a network. First, the ROADMs need to be configured to set up optical channels connecting points on the edge. This requires management tools that understand the properties and limitations of optical networks (as discussed in the previous section). These tools should reject policies that cannot actually be implemented given the limitations of optical networks.

Second, the edge (the packet switches and the hosts) might be in a different administrative domain from the optical core. For example, the edge may consist of corporate datacenters with the core being the ISP connecting them. Since the core may connect to many different entities on the edge, the administrator responsible for it may wish to minimize the information that needs to be shared with these edge entities. Conversely, entities operating the edge networks should not need to be bothered about the man-

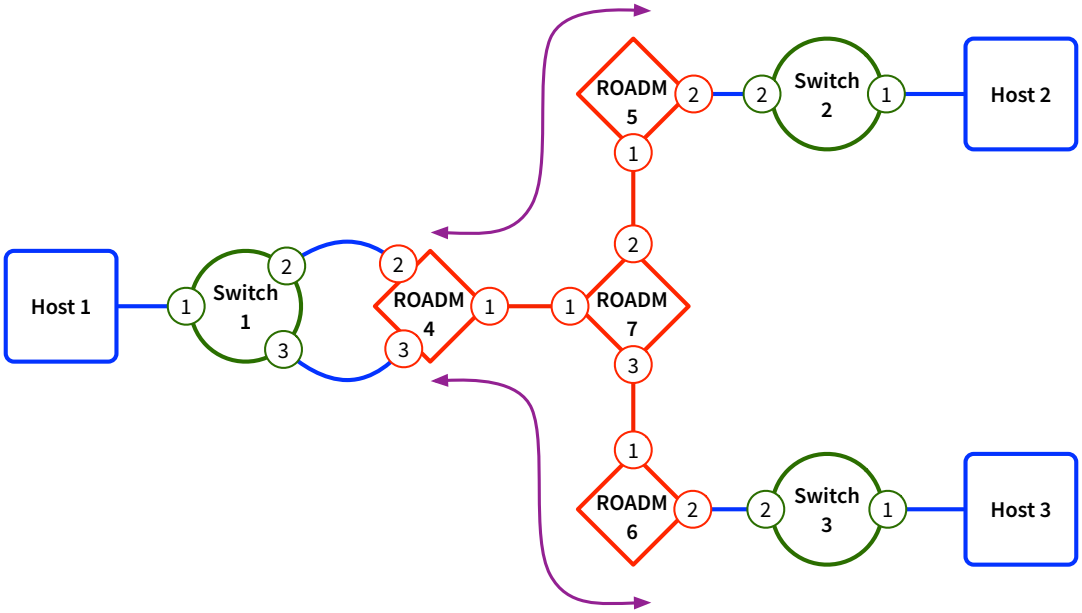


Figure 5.1: Optical Fork topology

agement of the core: it should suffice to give them some guarantee that the core will provide connectivity between relevant points on the edge.

Third, setting up and reconfiguring optical channels can be expensive, especially in comparison to reconfiguring packet switches. A programming paradigm for optical networks should acknowledge that they are likely to be connected to flexible packet switches at the edge, and leverage this fact to provide maximum programmability, while reducing the amount of churn in the optical network itself.

The first issue can be tackled with a path-based approach. The key building block of an optical network is the optical channel: essentially a path through the network, combined with some way to identify the channel. With this observation, optical channels could be configured using programs such as that shown below in Listing 5.1.

```
if roadm=4 and port=2 then
   channel:=1; (r4:1 => r7:1); (r7:2 => r5:1); port:=2
```

Listing 5.1: Optical Configuration NetKAT Program

This program implements the optical channel between port 2 on ROADM 4 to port 2 on ROADM 5. The program first checks for a starting ROADM and port, selects an optical channel (denoted by the integer 1), describes the path taken by this channel as a series of links, and finally specifies an output port. Statements of the form `s1:p1 => s2:p2` correspond to a network link between port `p1` on node `s1` and port `p2` on node `s2`. Note that neither the internal forwarding behavior of `r7` nor the final ROADM location need to be specified, as they can be inferred from the rest of the program. The language used by this program can be formalized as Circuit NetKAT, as described in the next section.

The difference in administrative domains and the performance limitations of optical channels can be jointly tackled by making use of the distinction between "edge" devices at the perimeter of the network and the optical "fabric" devices in the core that connect edge devices. The idea of implementing network programs in terms of an edge-fabric distinction is not new: the term "fabric" is borrowed from a paper by Casado et al. that is motivated by some of the same issues identified above [12]. This distinction is used in systems such as Felix, which analyzes the paths used by the fabric and pushes monitoring tasks to the edge of the network [14]. Generating edge configurations that correctly implement a high-level network policy in the presence of an existing fabric requires solving a number of technical challenges. This is the focus of the next chapter.

5.3 Circuit NetKAT

The first step in increasing the programmability of optical networks is to develop a language for specifying optical channels in a network. To do so, we can use the NetKAT programming language as a starting point. NetKAT is a programming language for SDN-capable networks based on Kleene Algebras with Tests (KAT) extended with primitives for matching against and modifying packet headers [5]. A NetKAT program expresses network forwarding behavior as functions on packets. These programs are im-

Syntax

Naturals	n	$::=$	$0 \mid 1 \mid \ldots$
Fields	f	$::=$	$f_1 \mid \cdots \mid f_k$
Packets	pk	$::=$	$\{f_1 = n_1, \cdots, f_k = n_k\}$
Predicates	a, b	$::=$	$true$ *Identity*
		$\mid$	$false$ *Drop*
		$\mid$	$f{=}n$ *Test*
		$\mid$	$a + b$ *Disjunction*
		$\mid$	$a \cdot b$ *Conjunction*
		$\mid$	$\neg a$ *Negation*
Programs	p, q	$::=$	a *Filter*
		$\mid$	$f{\leftarrow}n$ *Modification*
		$\mid$	$p + q$ *Union*
		$\mid$	$p \cdot q$ *Sequencing*
		$\mid$	p^* *Iteration*
		$\mid$	dup *Duplication*

Semantics

$$
\begin{aligned}
&[\![p]\!] \in pk \to \{pk\} \\
&[\![true]\!]\ pk \triangleq \{pk\} \\
&[\![false]\!]\ pk \triangleq \{\} \\
&[\![f{=}n]\!]pk \triangleq \begin{cases} \{pk\} & \text{if } pk.f = n \\ \{\} & \text{otherwise} \end{cases} \\
&[\![\neg a]\!]\ pk \triangleq \{pk\}\ ([\![a]\!]\ pk) \\
&[\![f{\leftarrow}n]\!]\ pk \triangleq \{pk[f := n]\} \\
&[\![p + q]\!]\ pk \triangleq [\![p]\!]\ pk \cup [\![q]\!]\ pk \\
&[\![p \cdot q]\!]\ pk \triangleq ([\![p]\!] \bullet [\![q]\!])\ pk \\
&[\![p^*]\!]\ pk \triangleq \bigcup_i F^i\ pk \\
&\text{where } F^0\ pk \triangleq \{pk\} \\
&\qquad \text{and } F^{i+1}\ pk \triangleq ([\![p]\!] \bullet F^i)\ pk
\end{aligned}
$$

Figure 5.2: NetKAT abstract syntax and semantics.

plemented in the network by compiling them to flowtables for SDN-capable switches (eg., switches supporting the OpenFlow protocol).

NetKAT treats a packet as a record of fields f ranging over standard headers such as Ethernet and IP source and destination, as well as logical fields such as `sw` and `pt`. These logical fields keep track of the switch and port where the packet is currently located in the network and are useful for representing packet forwarding and for program analysis. Atomic terms in the language are predicates on, or modifications to, packet fields. Each predicate behaves like a filter on packets—packets that do not match the boolean condition encoded in the predicate are dropped. Predicates include primitive tests on field values ($f = n$), as well as standard boolean operators ($+$, $\cdot$, and $\neg$). Modifications ($f \to n$) update the field f with the value n. The union operator ($p + p'$) copies the input packet, processes one copy using p and the other copy using p', and takes the union of the resulting sets of packets.

Note that some operators are overloaded and can be applied to predicates and policies—e.g., $+$ is meant to represent disjunction on predicates and union on poli-

cies. The behavior specified in the denotational semantics in Figure 5.2 captures both cases. The sequential composition operator ($p \cdot p'$) processes the input packet using p and then feeds each output of p into p'. Iteration p^* behaves like the union of p composed with itself zero or more times. To make authoring programs easier, links ($sw1 : pt1 \rightarrow sw2 : pt2$) and conditionals (`if-then-else`) are encoded as follows:

$$sw1 : pt1 \rightarrow sw2 : pt2 \triangleq$$
$$sw = sw1 \cdot pt = pt1 \cdot sw := sw2 \cdot pt := pt2$$
$$\text{if } a \text{ then } p_1 \text{ else } p_2 \quad \triangleq \quad (a \cdot p_1) + (\neg a \cdot p_2)$$

Although NetKAT allows the specification of behaviors such as forwarding paths, it cannot be directly applied to programming optical fabrics. First, NetKAT allows programs to match and modify a full range of packet headers. But a ROADM would only be able to match against network location and optical channel identifiers (unless resorting to expensive optical-electrical-optical conversions, which we want to avoid). Also, as noted in Section 5.1 optical networks have their own set of constraints that are not enforced by default in NetKAT. To recap, these are: optical continuity, split and merge restriction and transponder restrictions.

Enforcing these constraints requires a restricted subset of NetKAT, called Circuit NetKAT. As its name suggests, a Circuit NetKAT program is a set of circuits, where each circuit is defined by a starting switch and port, a channel identifier, a list of hops and a final egress port, where each switch is an optical ROADM. The syntax is given in Figure 5.3a.

Circuit NetKAT programs are valid if they satisfy the validity conditions outlined in Figure 5.3b. Each circuit can be viewed as an *allocation (A)* from (switch, port) pairs to a channel identifier. The first condition (CONTINUITY) states that all points on the path defined by the circuit are allocated the same channel identifier. This satisfies the optical continuity restriction. The second condition (DISJOINTNESS) states that if two circuits use the same channel identifier, then their paths must be disjoint, i.e., at each

$$
\begin{array}{rrcl}
\text{Filter} & f & ::= & \text{switch} = n \cdot \text{port} = n \\
\text{Channel} & w & ::= & \text{channel} := n \\
\text{Links} & l & ::= & sw : pt \rightarrow sw' : pt' \\
 & & \mid & l \cdot l \\
\text{Circuit} & c & ::= & f \cdot\ w \cdot\ l \cdot\ \text{port} := n \\
\text{Programs} & p & ::= & c \\
 & & \mid & p + p
\end{array}
$$

(a) Circuit NetKAT syntax.

$$
\begin{array}{rrcl}
\text{Allocation} & A \in (sw, pt) & \rightarrow & \text{channel} \\
\text{Channel} & C \in \text{circuit} & \rightarrow & \text{channel} \\
\text{Path} & P \in \text{circuit} & \rightarrow & \{(sw, pt)\}
\end{array}
$$

CONTINUITY

$$
\frac{\begin{array}{c} f = \text{switch} = sw \cdot \text{port} = pt \\ w = \text{channel} := n \\ c = f \cdot\ w \cdot\ l \cdot\ \text{port} := pt' \\ \text{last}(c) = (sw', pt') \\ \forall (sw, pt) \in P(c).\ A(sw, pt) = n \end{array}}{A \vdash c}
$$

DISJOINTNESS

$$
\frac{A \vdash c_1 \qquad A \vdash c_2 \qquad \mathrm{C}(c_1) = \mathrm{C}(c_2) \Rightarrow \mathrm{P}(c_1) \cap \mathrm{P}(c_2) = \emptyset}{A \vdash c_1 + c_2}
$$

(b) Circuit NetKAT validity rules.

Figure 5.3: Circuit NetKAT syntax and validity rules.

port, a particular channel comes from, and is forwarded to, at most one destination. This condition satisfies the split, merge and transponder restrictions above.

A compiler takes a Circuit NetKAT program and checks if the program is valid according to the above conditions. If the program is valid, it is converted into a standard NetKAT program, otherwise it is rejected. This conversion simply involves inserting port assignments in between the links to properly forward signals from ingress to egress ports on each ROADM. The resulting program can then be compiled to flowtables that implement an optical forwarding fabric, by leveraging an optical extension to the OpenFlow protocol that supports optical devices.

5.4 Summary

Modern networks are increasingly heterogeneous systems composed of a variety of devices with differing abilities and interfaces. While technologies like SDN bring greater

programmability to a certain class of networks and devices, other important types of networks are left behind by the assumptions of current SDN standards and implementations. In particular, optical networks have several properties and limitations that make it difficult to simply adopt technologies developed for packet-based networks.

This chapter described the specific properties and challenges of bringing increased programmability to optical networks and their interplay with traditional packet networks. The Circuit NetKAT language is developed as an approach for programming optical networks, based on the insight that optical channels are essentially paths through the network. This lays the groundwork for the next chapter where the edge/fabric distinction between modern deployments of packet and optical networks is leveraged to address the challenges of flexibility and performance inherent to these heterogeneous networks.

CHAPTER 6

EDGE PROGRAMMING

The previous chapter looks at the heterogeneous nature of modern networks, focusing on optical networks are connected to traditional packet networks at the edges. As described in Section 5.2, these *hybrid networks* pose a number of challenges to programmability: heterogenous devices with differing capabilities and interfaces, multiple administrative domains, and non-trivial and inherent performance costs. One way to address these challenges is to distinguish the "edge" devices at the perimeter of the network from the "fabric" devices in the core that connect between edge points. So long as the edge devices provide SDN-like functionality, it is possible to implement a broad set of policies. Meanwhile the constrained fabric devices (such as ROADMs), only need to implement the "plumbing" to carry packets across the network.

This chapter presents a practical framework—EdgeNetKAT—for implementing high-level policies at the edge, using a fixed fabric to provide connectivity through the core of the network. The main approach is to *(*unravel) a high-level program representing the behavior of an entire network, into a program that describes only the configuration of edge devices, by utilizing an existing fabric to provide connectivity between the edge devices. This involves overcoming a series of technical challenges in building our framework for "unraveling" policies into configurations:

- **Analysis:** Generating configurations for the devices at the edge, requires knowledge of how the fabric forwards traffic between end points. EdgeNetKAT builds on advances in data plane verification, and the NetKAT framework [5, 70], to compute the both requirements of the policy and the forwarding functionality provided by the fabric.
- **Adaptation:** To faithfully implement a high-level policy using a fixed fabric, we need to check that the transformations on packets performed in the fabric are not

in conflict with the transformations performed at the edge. Sometimes it is possible to co-opt "spare" bits in the header field to encode the high-level policy, but more generally it is necessary to rely on some form of tunneling. Conveniently, optical networks provide a form of tunneling via their optical channels.

- **Expressiveness:** Certain policies can be expressed in terms of a "one big switch" abstraction, in which only input-output behavior matters [41]. But other policies, such as network function virtualization and middlebox service chaining require paths with multiple segments, or require that paths traverse certain nodes. EdgeNetKAT naturally supports policies based on the "one big switch" abstraction, and can be extended to support segmented paths and path constraints.

The remainder of this chapter presents an example of programming a heterogenous network by utilizing the edge/fabric distinction, and then describes the EdgeNetKAT framework and the techniques used to unravel global programs into edge programs.

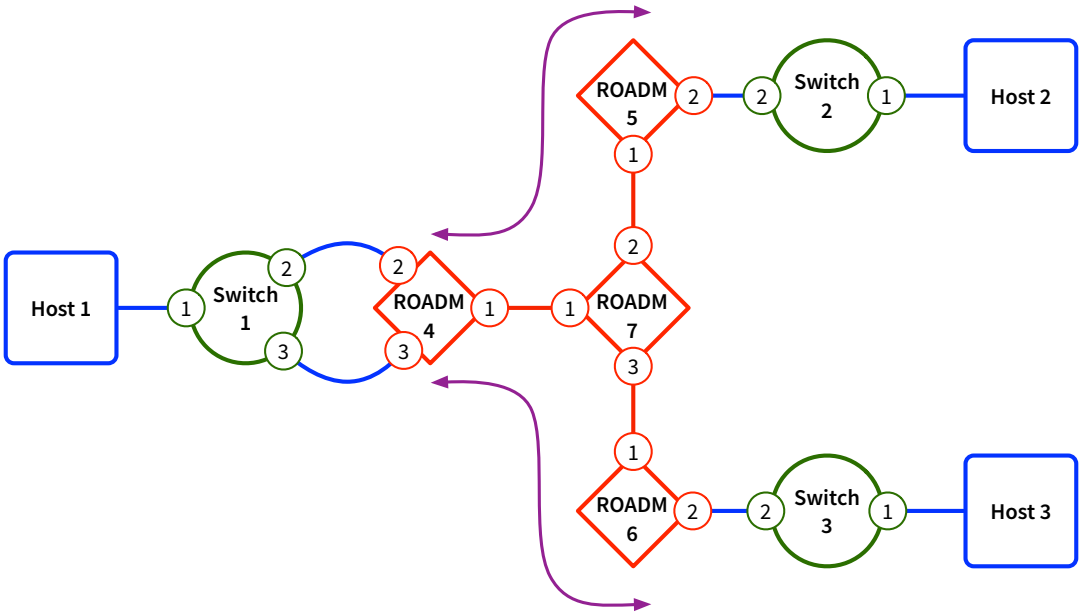


Figure 6.1: A Hybrid Network with a Optical Core and a Packet Edge

6.1 An Edge Programming Example

As an example, consider the hybrid network shown in Figure 6.1. This is the same network as used in Section 5.1, but with an optical core composed of two two optical channels: one connecting port 2 on ROADM 4 to port 2 on ROADM 5, and another connecting port 3 on ROADM 4 to port 2 on ROADM 6.

Recall that Section 5.3 describes Circuit NetKAT—a programming language capable of describing optical channels. The required channels can be implemented with a Circuit NetKAT program as shown in Listing 6.1 below.

```
if roadm=4 and port=2 then
   channel:=1; (r4:1 => r7:1); (r7:2 => r5:1); port:=2
else if roadm=4 and port=3 then
   channel:=2; (r4:1 => r7:1); (r7:3 => r6:1); port:=2
else if roadm=5 and port=2 then
   channel:=1; (r5:1 => r7:2); (r7:1 => r4:1); port:=2
else if roadm=6 and port=2 then
   channel:=2; (r6:1 => r7:3); (r7:1 => r4:1); port:=3
```

Listing 6.1: Optical Channel Configuration in Circuit NetKAT

Note that as with NetKAT, the `if-then-else` construct can be encoded as:

$$\text{if } a \text{ then } p_1 \text{ else } p_2 \quad \triangleq \quad (a \cdot p_1) + (\neg a \cdot p_2)$$

One plausible scenario for such a network is that Host 1 represents a front-end for the network, with Host 2 representing web servers (HTTP) and Host 3 representing email (SMTP) servers. In that case, the network policy would send all HTTP traffic (TCP destination port 80) from Host 1 only to Host 2, and all SMTP traffic (TCP destination port 25) from Host 1 only to Host 3.

To express this policy, we need a concise way of specifying predicates on packet header fields, and forwarding paths through the network. Fortunately, the NetKAT

programming language (also described in Section 5.3), allows us to do just so [5]. A NetKAT program expresses network behavior in terms of functions on packets. that are compiled to flowtables for SDN-capable switches. The intended forwarding behavior is concisely expressed as a NetKAT program in Listing 6.2.

```
if switch=1 and port=1 and tcpDst=80 then
   (s1:1 => s2:1);
else if switch=1 and port=1 and tcpDst=25 then
   (s1:1 => s3:1);
else if switch=2 and port=1 then
   (s2:1 => s1:1);
else if switch=3 and port=1 then
   (s3:1 => s1:1)
```

Listing 6.2: User Policy as a NetKAT Program

Implementing this policy on the given topology presents several challenges. Since reconfiguring the optical channels incurs a large time penalty, we should avoid changing the existing core configuration if possible. In that case, even though the user is only concerned with end-to-end behavior, they would have to understand the details of the optical fabric configuration. Then, instead of the clean policy shown in Figure 6.2, the user would have to manually match up packet switch ports to the optical transceiver ports and the channels they connect to. Finally, the user would need to write another program—either in NetKAT or directly as a forwarding table—that operates on the edge switches and correctly implements the desired forwarding behavior. Although doing all this is feasible in principle, it is quite tedious and error prone process—moreover, it would need to repeated every time the policy changes.

Fortunately, in addition to allowing us to specify network behavior, the NetKAT language and its compiler provides the tools required to automate this rewriting. Us-

ing the optical fabric program (shown in Listing 6.1), we can unravel the user policy into ingress and egress NetKAT programs that affect the edge switches only (as shown in Listings 6.3 and 6.4 respectively). Section 6.2 describes how NetKAT compiler transforms programs into *dyads*—pairs of predicates on packet header fields and modifications to apply to packets. The EdgeNetKAT extension then matches dyads from the policy to those in the fabric.

```
if switch=1 and port=1 and tcpDst=80 then
   vlanId := 1; port := 2
else if switch=1 and port=1 and tcpDst=25 then
   vlanId := 2; port := 3
else if switch=2 and port=1 then
   vlanId := 3; port := 2
else if switch=3 and port=1 then
   vlanId := 4; port := 2
```

Listing 6.3: Generated NetKAT Ingress Program

```
if vlanId=1 and switch=3 and port=2 then
   strip vlan; port := 1
else if vlanId=2 and switch=2 and port=2 then
   strip vlan; port := 1
else if vlanId=3 and switch=1 and port=2 then
   strip vlan; port := 1
else if vlanId=4 and switch=1 and port=3 then
   strip vlan; port := 1
```

Listing 6.4: Generated NetKAT Egress Program

This unraveling of a global program into a pair of edge programs has several benefits. First, the user can ignore the specific configuration of the core fabric and instead

treat it as "one big switch" that connects the edge locations. Instead, the user can write a high-level, global program and let the EdgeNetKAT compiler match optical channels in the fabric to paths implicitly required by the global program. This abstraction means that different entities can be in charge of managing the core and the edge. For example, the Hosts in the example could be datacenters under the control of some kind of web service while the fabric is controlled by a cross-country ISP.

Second, as part of the unraveling process, any predicates (eg., `tcpDst=80`) or modifications specified in the policy are relocated to the edge. Whenever a policy changed, only the edge switches need to be updated. This reduces the overheads associated with implementing policy changes, since the SDN-capable switches can be updated faster than the ROADMs in the core.

Edge compilation embraces the unique features of hybrid networks to increase their programmability while providing important benefits. This approach crucially depends on the path-oriented properties of the NetKAT and Circuit NetKAT programming languages, as described in the next section.

6.2 Compilation to the Edge

Unraveling global network programs into edge programs makes use of the features of the NetKAT programming language and the intermediate data structures of the NetKAT compiler. As described in Section 5.3, NetKAT programs are combinations of predicates on, or modifications to, packet fields. By treating the packet's location in the network (`switch, roadm` or `port`) as a logical packet header, NetKAT can be used to model links or paths through the network.

NetKAT programs are compiled to intermediate representations called Forwarding Decision Diagrams (FDDs), which are used to generate forwarding tables for switches running the OpenFlow protocol [70]. By analyzing the FDD representation of the pro-

gram we can pair up each predicate used to distinguish traffic classes, and the corresponding modifications made to matching packets. We call each such pair a *dyad*. Since switch and port location are logical fields in packet headers, the collected predicates include the starting location, and modifications include the destination location. Thus each dyad denotes the source and sink of a particular traffic class.

Performing this analysis on the user policy gives us the *required* sources and sinks of each traffic class. The same analysis on the fabric gives us sources and sinks of each path *provided* by fabric. The required end-points can be matched to the provided paths using a number of methods, including a simple graph algorithms or a translation to a linear programming problem.

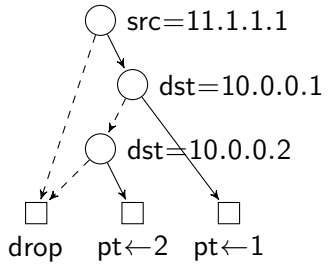
Once policy end-points are matched to fabric paths, we need to distinguish multiple policy traffic classes that are sent across the same fabric paths. At the destination, we may need to separate them out again, either to forward out different ports, or to modify header fields in different ways. To do this, the ingress program uses the policy's predicates to match incoming traffic and applies a unique tag (either VLAN or MPLS) to each packet before forwarding to ports that match suitable paths in the fabric (as in Listing 6.3). Conversely, the egress program matches on the tag at edge locations to perform policy-specified modifications and final forwarding (as in Listing 6.4).

6.2.1 Forwarding Decision Diagrams and Dyads

NetKAT programs can be compiled to an intermediate representation called a Forwarding Decision Diagram (FDD) [70]. FDDs are generalizations of structures called binary decision diagrams [3]. They are trees where internal nodes represent tests on packet headers, each with a "true" and a "false" branch. Leaf nodes are sets of modifications to packet headers. Figure 6.2 shows an example NetKAT program and the FDD it generates.

A leaf node in the FDD is a set of actions, denoted $\{a_1, \ldots, a_k\}$. An action a maps

```
if src=11.1.1.1 then
 (if dst=10.0.0.1 then port:=1
  else if dst=10.0.0.2 then port:=2
  else false)
else
 false
```



α	β
¬src=11.1.1.1	drop
src=11.1.1.1 ∧ ¬dst=10.0.0.2 ∧ ¬dst=10.0.0.1	drop
src=11.1.1.1 ∧ dst=10.0.0.1	pt←1
src=11.1.1.1 ∧ dst=10.0.0.2 ∧ ¬dst=10.0.0.1	pt←2

Figure 6.2: Example NetKAT program, FDD, and dyads.

fields to values: $\{f_1 \leftarrow n_1, \ldots, f_k \leftarrow n_k\}$ with each field occuring at most once. An internal node, written $(f = n \,?\, d_1 : d_2)$, is specified by a test $f = n$ and two sub-diagrams. If the packet satisfies the test, the true branch (d_1) is evaluated, otherwise the false branch (d_2) is evaluated. FDDs also satisfy well-formedness judgments ensuring that tests appear in a consistent order and do not contradict previous tests to the same field.

The NetKAT primitives $true$, $false$, and $f \leftarrow n$ all compile to simple leaf nodes. The empty action set $\{\}$ drops all packets, while the singleton action set $\{\{\}\}$ contains the identity action $\{\}$, which copies packets unchanged. NetKAT tests $f = n$ compile to a conditional whose branches are the FDDs for $true$ and $false$ respectively. The union operator $(d_1 + d_2)$ traverses d_1 and d_2 and takes the union of the action sets at the leaves. Sequential composition $(d_1 \cdot d_2)$ merges two packet-processing functions into a single function. The FDD Kleene star operator $d*$ is defined using a fixed-point computation, and the well-formedness conditions on FDDs ensure that such a fixed point exists. The compilation process, including the well-formedness conditions, is described in detail in the original paper on the NetKAT compiler [70].

A depth-first search over the FDD lets us collect up pairs of predicates (the conjunction of internal nodes, denoted α) and corresponding modifications (a leaf node,

denoted β) that encode the input-output behavior of the program. That is, by compiling a NetKAT program to an FDD, we can easily produce a compact representation of its forwarding behavior that can be used for further analysis. We call each pair of a predicate α and its corresponding modifications β a *dyad*. Since the analysis form includes the topology and ingress and egress predicates, α includes the starting switch and port, and β includes the destination switch and port. Thus a dyad captures the source and sink of all traffic satisfying a given predicate.

Figure 6.2 shows a simple NetKAT program, the corresponding FDD and the generated dyads. There are four paths from the root node of the FDD to a leaf. The two leftmost paths lead to a `drop` node. The rightmost path checks the source IP `src=11.1.1.1` and destination IP `dst=10.0.0.1` and forwards out port 1. The remaining path checks the source and destination IP addresses, must also check for the negation of the previous destination IP address.

Performing this analysis on the user policy gives us the *required* sources and sinks of each traffic class. The same analysis on the fabric gives us sources and sinks of each path *provided* by fabric. The EdgeNetKAT compiler leverages the FDD structure and the dyads derived from them to determine how required functionality can be mapped onto an existing fabric.

6.2.2 Basic Edge Compilation

The EdgeNetKAT compiler starts with a network policy and generates edge configurations that leverage the fabric for connectivity. The compiler takes as inputs a network policy and a forwarding fabric (both expressed as NetKAT programs), and a set of edge switches to target. Additionally, the compiler assumes knowledge of the physical topology, such as the ingress/egress predicates for both the policy and fabric. From these inputs, the compiler generates ingress and egress NetKAT programs, with the

following properties:

1. **Edge implementation:** Both programs can be implemented entirely on edge switches—i.e., any `switch` predicates in the generated programs only match edge switches.
2. **Ingress classification:** The ingress program implements the same traffic classification as the user policy—i.e., the union of all the αs derived from the user policy's FDD.
3. **Egress modification:** The egress program implements the same modifications to packet header fields as the user policy—i.e., for each α implemented by the ingress program, the egress program must apply the corresponding β to the same traffic class.
4. **Fabric transit:** From each (α, β) pair derived from the policy's FDD, the fabric forwards from the source location in α to the sink location in β.

Together, these properties ensure a "one big switch" abstraction [41]—a combination of edge program and fabric is equivalent to a policy program if they produce the same input/output behavior. Formally, if f and p are the NetKAT programs for the fabric and policy, then $\phi_f = [\![f]\!], \phi_p = [\![p]\!]$ denotes the corresponding packet forwarding functions according to the NetKAT semantics in Figure 5.2. The desired edge forwarding functions are given by ϕ_i (for ingress), and ϕ_o (for egress). We claim that the correctness condition for a compiler implementing the "one big switch" abstraction is captured by the following equivalence:

$$\phi_i \bullet \phi_f \bullet \phi_o \equiv \phi_p$$

Our compiler computes programs i and o such that $\phi_i = [\![i]\!]$ and $\phi_o = [\![i]\!]$, or fails if no such programs exist.

The NetKAT compiler generates Forwarding Decision Diagrams for both the fabric and uers programs. By iterating through the FDD, we convert each program to a set of dyads—pairs of predicates (α) and modifications (β). Recall from Section 6.2.1 that α includes the starting switch and port (sources), and β includes the destination switch and port (sinks). In order to correctly implement a policy using an existing fabric, the sources and sinks required by the policy's dyads need to be matched to those provided by the fabric. The compiler uses two approaches to solving this selection problem.

The first approach is based on simple graph algorithms. The compiler constructs a *connectivity graph* G where the nodes are the sources and sinks of the user policy. There is an edge between two nodes if they are connected via a path in the fabric. We determine this by iterating through the (α, β) pairs for the fabric, and adding an edge to G if the α contains (or is one hop away from) a source and the corresponding β contains the sink (or is one hop away from it). To connect a source and sink, the compiler checks whether there is an edge between them in G.

The second approach is based on a formulation as a linear programming problem whose solution selects a fabric dyad to implement each policy dyad. The compiler generates a sequence of variables $V_{i,j}$ denoting the *possibility* of policy dyad i using the fabric dyad j. If the endpoints of policy dyad i and the fabric dyad j are not identical (or adjacent), then there is a constraint limiting $V_{i,j} = 0$. In the basic case, a single fabric dyad is chosen to implement each policy dyad. This is enforced by a constraint $\sum_{j \in \text{fabric}} V_{i,j} = 1$ for each policy dyad j. The full linear programming formulation is given in Figure 6.3.

Using a linear programming formulation is more powerful than is strictly needed, but it allows for better extensibility. To include additional features such as path constraints, simply requires adding more variables and constraints to the linear programming problem. Without it, the compiler would need custom analyses over the dyads or

	Element	Definition
Input	P	Policy dyads, indexed by i
	F	Fabric dyads, indexed by j
	src(d)	A function (Dyad $\rightarrow$ switch)
	dst(d)	A function (Dyad $\rightarrow$ switch)
Generated	$V_{i,j}$	Policy dyad i possibly implemented by fabric dyad j
Output	$V_{i,j} = 1$	Policy dyad i implemented by fabric dyad j

Minimize	1	since we are not performing any optimization
such that	$\forall i \in \text{P}$	$\sum_{j \in \text{F}} V_{i,j} = 1$
	$\forall i \in \text{P}, \forall j \in \text{F}$	$\text{src}(P_i) \neq \text{src}(F_j) \vee \text{dst}(P_i) \neq \text{dst}(F_j)$: $V_{i,j} = 0$ (or drop $V_{i,j}$ from the problem)

Figure 6.3: Dyad selection as an linear programming problem.

the connectivity graph for the same functionality. Note also that the generic objective function used here could be replaced with network-specific objectives.

After a suitable fabric dyad is found for each policy dyad, the final step is to generate ingress and egress programs that implement the policy using the fabric. Each α in the policy is used as the predicate for a forwarding rule on the source switch, and generate output actions for the rule in two steps. At the source, since more than one stream of traffic may take the same path through the fabric, the ingress program attaches a tag (eg, a VLAN tag) unique to this α to each packet. The collection of these forwarding rules form the required *ingress program*. Similarly,the corresponding β is installed on the corresponding sink, modified to act only on traffic matching the tag attached by the source. These modified βs form the *egress program*.

6.3 Extensions to Edge Compilation

6.3.1 Segmented Path Compilation

The "one big switch" abstraction allows network administrators to specify the endpoints of a particular class of traffic. A natural extension is to allow specifying an entire path, (e.g., $s_1 \Rightarrow s_2 \Rightarrow s_3$) instead of just a source-sink pair. Such a segmented path connects intermediate nodes that are part of the user-controlled edge. For each neighboring pair of nodes in the chain, the compiler would have to find a connecting segment through the fabric. The segments are then chained together to construct the whole path. Just as in dyad matching, a unique tag (e.g., a VLAN tag) differentiates traffic classes and track them across segments. An ingress program that matches the policy's α and attaches the appropriate tag is installed at the start of the path. At each intermediate node, traffic is sent from the fabric to the edge switch, and install *bounce* programs that examine the tag and return traffic to the fabric. Finally, an egress program at the end matches the tag and applies modifications according to the policy's β. Note that rules are installed only on the relevant edge switches, without modifying the fabric connecting them. Thus the core of the network can remain static, reducing the overhead in changing network policy.

Segmented paths are simply an extended form of dyad matching. The compiler finds a fabric dyad to carry the traffic in between each consecutive pair of points. The LP back-end is extended with some additional bookkeeping to reuse the same tag across each segment, and then apply the proper modifications at the end. NetKAT can already describe paths by specifying each hop, as shown in our motivating example in Listing 6.1. This extension allows us to to describe general paths while letting the compiler determine the specific hops. Such policies are useful for applications involving service chaining and middleboxes—e.g., network functions such as firewalls and intru-

```
if dst=backend and tcpDst=80 then
   frontend ==> firewall ==> backend
else if dst=backend and tcpDst=22 then
   frontend ==> backend
else if dst=frontend then
   backend ==> frontend
```

Listing 6.5: Multi-segment program for applying a firewall.

```
if switch=2 and port=1 and dst=frontend then
   vlanId := 3; port := 2
else if switch=1 and port=1 and
        tcpDst=22 and dst=backend then
   vlanId := 2; port := 3
else if switch=1 and port=1 and
        tcpDst=80 and dst=backend then
   vlanId := 1; port := 2

if vlanId=3 and switch=1 and port=3 then
   strip vlan; port := 1
else if vlanId=2 and switch=3 and port=2 then
   strip vlan; port := 1
else if vlanId=1 and switch=2 and port=2 then
   strip vlan; port := 1
else if vlanId=1 and switch=3 and port=2 then
   strip vlan; port := 1
```

Listing 6.6: Ingress and egress program for firewall application.

sion detection are implemented on nodes at various points in the network, and simpler switches in the core of the network move traffic to the required processing nodes.

For example, Listing 6.5 shows a NetKAT program that directs Web requests from a front-end to a back-end through the firewall, but SSH traffic and traffic from back-end to front-end can pass directly through. An optical fabric similar to that in Figure 6.1 could support this program, with the firewall, front-end and backend replacing the hosts. The fabric program would be similar to Listing 6.1. Listing 6.6 shows generated ingress and egress programs. VLAN tags are used to separate different traffic classes. We assume that the intermediate nodes require the original traffic, without tags. Therefore tags are removed and reapplied at the end of every segment.

6.3.2 Compilation With Path Constraints

	Element	Definition
Input	P	Policy dyads, indexed by i
	F	Fabric dyads, indexed by j
	src(d)	Function: Dyad $\rightarrow$ source switch
	dst(d)	Function: Dyad $\rightarrow$ sink switch
	path(d)	Function: Fabric dyad $\rightarrow$ nodes on path
	pcs(d)	Function: Policy dyad $\rightarrow$ path constraints
	$N_{n,j}$	Fabric nodes n used by dyad j
Generated	$V_{i,j}$	Dyad i possibly implemented by dyad j
Output	$V_{i,j} = 1$	Dyad i implemented by dyad j

$$\begin{array}{lll} \text{Minimize} & 1 & \text{since there is no optimization} \\ \text{such that} & & \\ \forall i \in \text{P} & \sum_{j \in \text{F}} V_{i,j} = 1 & \\ \forall i \in \text{P}, \forall j \in \text{F} & \sum_{n \in pcs(i)} N_{n,j} < |pcs(i)| : & V_{i,j} = 0 \\ \forall i \in \text{P}, \forall j \in \text{F} & \text{src}(P_i) \neq \text{src}(F_j) & \\ & \vee\ \text{dst}(P_i) \neq \text{dst}(F_j) : & V_{i,j} = 0 \end{array}$$

Figure 6.4: Dyad selection with path constraints as a linear programming problem.

Segmented paths allow the policy to direct traffic across multiple points on the edge. However they do not provide any control over how the fabric is utilized—the compiler chooses any available fabric dyad with matching end-points. Finer-grained control can be exposed by incorporating path constraints—i.e., instead of allowing arbitrary intermediate nodes on the path that implements a policy, we allow the programmer to specify a number of additional points on the paths that traffic must pass through. To find a matching fabric dyad, we need to consider the entire path represented by the dyad, not just the endpoints. Only the fabric dyads whose paths contain the required number of intermediate nodes can be used to carry the policy traffic. This form of path constraint is particularly useful in the optical domain—optical signals need to be

regenerated after being transmitted for a certain distance. But since regenerators are more expensive, only a certain number of nodes in each path can be regenerators. By specifying that the appropriate regenerators must be visited as points on the path, a policy can ensure that traffic will reach its destination after being regenerated the appropriate number of times.

Starting from the linear programming formulation described in Section 6.2.2, the EdgeNetKAT compiler adds more constraints to capture the fabric's provided paths and the policy's required intermediate nodes. First, we use the NetKAT compiler framework to produce a mapping from fabric dyads to paths (the $path(d)$ function in Figure 6.4). This can be done by symbolically executing the NetKAT program with respect to the given dyad. The policy also produces a mapping from policy dyads to the intermediate nodes required for each dyad. This is represented by the $pcs(d)$ function in Figure 6.4

Each dyad j, and node n in the fabric is represented as a variable $N_{n,j}$. Recall that in the original linear programming formulation includes variables $V_{i,j}$ representing the possibility of policy dyad i being implemented using fabric dyad j. We constrain $V_{i,j} = 0$ unless the dyad endpoints are the adjacent. If policy dyad i also specifies nodes $n_0 \ldots n_k$ as intermediates, we generate further constraints that set $V_{i,j} = 0$ unless all of $N_{n_0,j} \ldots N_{n_k,j}$ are 1. The extended linear programming formulation is shown in Figure 6.4.

6.4 Summary

This chapter tackles the problem of deploying high-level user policies to heterogenous networks. Assuming an optical core network, surrounded by flexible, SDN-enabled edge switches, the EdgeNetKAT compiler unravels high-level global NetKAT programs into programs that only affect edge switches, leveraging the fabric to provide connectivity. To do this, EdgeNetKAT exploits the properties of the NetKAT language and its

compiler data structures such as FDDs. This basic compilation to the edge can be extended to allow users greater flexibility in how the fabric paths are used to implement the global program. EdgeNetKAT provides a practical way to reap the benefits of high-level network programming languages, while operating on realistic hybrid networks.

CHAPTER 7

IMPLEMENTATION AND EVALUATION

This chapter describes the implementation and evaluation of two systems: the Merlin system for programming networks with bandwidth constraints, and the EdgeNetKAT compiler for compiling global NetKAT programs to edge programs.

7.1 Implementation of the Merlin System

The Merlin system is implemented in three parts: a compiler that takes Merlin programs and network topology descriptions and generates configurations for network devices, and a SDN-like controller that installs said configurations on network devices, and a negotiator framework for delegating and verifying Merlin programs.

The compiler component is implemented in about 4000 lines of OCaml. This includes parsers for the Merlin language and network topology descriptions, a representation of the logical topology described in Section 2.4.1, an interface to a MIP problem solver and a code generation engine that converts the results of the MIP solution to specific device configurations.

As described in Section 3.1, solving a MIP problem is a core part of the bandwidth allocation process. For this, compiler uses an external solver: the Gurobi Optimizer [31] to solve constraints, converting the generated problem to Gurobi syntax, and the solution back to a form usable by the code generation engine. However, any MIP solver could be used instead.

The controller component consists of about 100 lines of OCaml and leverages the Frenetic SDN [24] framework to install forwarding rules on OpenFlow switches. The code generator generates Click router [45] to manage software middleboxes, and `ipfilter` and `tc` configurations to control bandwidth on Linux end hosts. The controller also supports installing these scripts on the respective machines. Note that the

design of Merlin does not depend on these specific systems. The code generation and controller components provide clean interfaces for incorporating different backends, allowing for others to instantiate Merlin with alternative device managers.

The Merlin negotiator and verification mechanisms uses standard algorithms for transforming and analyzing predicates and regular expressions. To delegate a policy, Merlin intersects the predicates and regular expressions in each statement with those in the original policy to project out the policy for the sub-network. To verify implications between policies, Merlin uses the Z3 SMT solver [56] to check predicate disjointness, and the Dprle library [33] to check inclusions between regular expressions.

7.2 Evaluation of the Merlin System

The Merlin system has been evaluated on three criteria:

1. The expressiveness of the Merlin policy language.
2. The ability of Merlin to improve end-to-end performance for applications.
3. The scalability of the compiler and negotiator components with respect to network and policy size.

The experiments were run on a cluster of Dell r720 PowerEdge servers with two 8-core 2.7GHz Intel Xeon processors, 32GB ram, and four 1GB NICs. The cluster used a Pica8 Pronto 3290 switch to connect the machines. To test the scalability we ran the compiler and negotiator frameworks on various topologies and policies.

The experiments show that Merlin can effectively provision and configure real-world datacenter and enterprise networks. Merlin concisely expresses rich network policies and can be used to obtain better performance for big-data processing applications, replication systems.

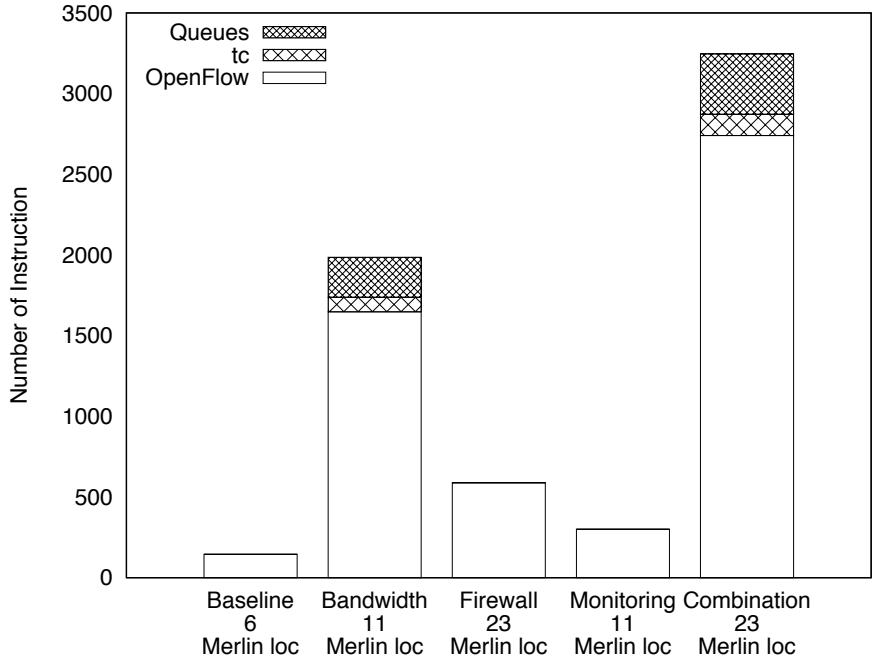


Figure 7.1: Merlin expressiveness with policies for the Stanford campus topology.

7.2.1 Expressiveness

The expressiveness of the Merlin policy language was evaluated by developing several network policies for the 16-switch Stanford campus backbone topology [7]. 24 hosts were added to each of the 12 edge switches in the topology with each pair-wise exchange of traffic between hosts identified as a separate traffic class, producing $82,656$[1]total traffic classes. We then implemented a series of policies in Merlin, and compared the sizes of the Merlin source policies and the outputs generated by the compiler. This comparison measures the degree to which Merlin is able to abstract away from hardware-level details and provide effective constructs for managing a network.

The Merlin policies implemented are as follows:

1. *Baseline.* This policy creates pair-wise forwarding rules for all hosts in the net-

[1]With 24 hosts for each of 12 switches: $(24 * 12)^2 - (24 * 12) = 82,656$.

work. The policy is restricted to only forwarding, and does not use network functions or specify bandwidth constraints. It therefore provides a baseline measurement of the number of low-level instructions that would be needed in almost any non-trivial application. The Merlin policy is only 6 lines long and compiles to 145 OpenFlow rules.

2. *Bandwidth.* This policy augments the basic connectivity by providing 10% of traffic classes a bandwidth guarantee of 1Mbps and a cap of 1Gbps. Such a guarantee could be useful, for example, to prioritize emergency messages sent to students. This policy required 11 lines of Merlin code, but generates over 1600 OpenFlow rules, 90 TC rules and 248 queue configurations. The number of OpenFlow rules increased dramatically due to the bandwidth guarantees, which required provisioning separate forwarding paths for a large collection of traffic classes.

3. *Firewall.* This policy assumes the presence of a middlebox that filters incoming web traffic. The baseline policy is altered to forward all packets matching a particular pattern (e.g., `tcp.dport = 80`) through the middlebox. This policy requires 23 lines of Merlin code, but generates over 500 OpenFlow instructions.

4. *Monitoring.* This policy attaches middleboxes to two switches and partitions the hosts into two sets of roughly equal size. Hosts connected to switches in the same set may send traffic to each other directly, but traffic flowing between the sets must pass through a middlebox. This policy is useful for filtering traffic from untrusted sources, such as student dorms. This policy required 11 lines of Merlin code but generates 300 OpenFlow rules, roughly double the baseline.

5. *Combination.* This policy augments the baseline with a filter for web traffic, bandwidth guarantees for some traffic classes, and a monitoring policy for some hosts. This policy is expressed in only 23 lines of Merlin code, but generates over 3000 instructions for network devices, including OpenFlow and `tc` rules,

and queue configurations.

The results of this experiment are depicted in Figure 7.1. Overall, using Merlin significantly reduces the effort, in terms of lines of code, required to provision and configure network devices for a variety of real-world management tasks.

7.2.2 Application Performance

The second set of experiments explore Merlin's ability to express policies beneficial for real-world applications. Specifically, they show that bandwidth provisioning and function placement improves the performance of data center applications. They also provide a proof-of-concept that Merlin can effectively manage datacenter networks.

Hadoop Hadoop is a popular open-source implementation of MapReduce [17] is widely-used for data analytics. A Hadoop computation proceeds in three stages: the system (i) applies a *map* operator to each data item to produce a large set of key-value pairs; (ii) *shuffles* all data with a given key to a single node; and (iii) applies the *reduce* operator to values with the same key. The many-to-many communication used in the shuffle phase often results in heavy network load, making Hadoop jobs sensitive to background traffic. In practice, this background traffic can come from a variety of sources. For example, some applications, such as system monitoring tools [75, 74], network overlay management [37], and even distributed storage systems [75, 18], use UDP-based gossip protocols to update state. A sensible network policy would be to provide guaranteed bandwidth to Hadoop and best-effort service to UDP traffic.

```
[ x : (ip.src = 192.168.1.1/16 and ip.dst = 192.168.1.1/16 and
       ip.proto = 0x06 and tcp.dst = 50060)
      -> .* ], min(x,100MB/s)
```

Listing 7.1: Merlin Program for Hadoop Application.

The Merlin program used to implement the guarantee for Hadoop traffic is shown in Listing 7.1. To show its impact, we ran a Hadoop job that sorts 10GB of data from a corpus of open source texts (e.g., Shakespeare's plays, etc.), and measured the time to complete it on a cluster with four servers. The cluster was configured so that all servers could act as mappers or reducers. The experiment ran under three configurations:

1. *Baseline.* Hadoop had exclusive access to the network.
2. *Interference.* The `iperf` tool injects UDP packets, simulating background traffic.
3. *Guarantees.* We again injected background traffic, but guaranteed 90 percent of the capacity for Hadoop.

With exclusive network access, the Hadoop job finished in 466 seconds. With background traffic causing network congestion, the job finished in 558 seconds, a roughly 20% slow down. With bandwidth guarantees, the job finished in 500 seconds, corresponding to the 90% allocation of bandwidth.

Deep Packet Inspection Merlin programs can also improve application performance through the careful placement of network functions. To demonstrate how placement can impact performance, we designed an experiment inspired by recent work on moving middlebox functionality to end-hosts [20]. We measured traffic latency for a network under two possible configurations: one in which all traffic is routed through a Deep Packet Inspection (DPI) middlebox, and one in which DPIfunctionality is implemented on end hosts. We implemented the DPIfunctionality using Click [45].

The experiment network consisted of five machines connected to a single switch. Two machines were used to generate traffic for which we measured latency. Two machines were used to generate background traffic. The final machine acted as the DPI middlebox.

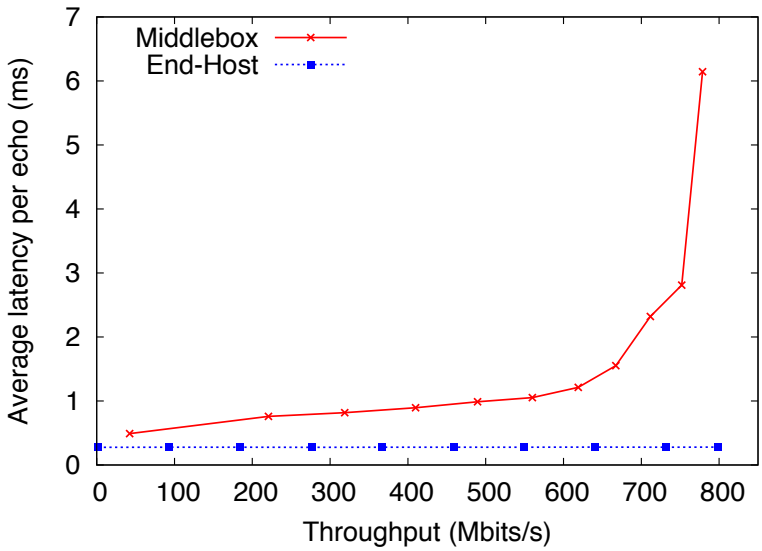


Figure 7.2: A Merlin policy inspired by ETTM [20] showing that a centralized middlebox implementation of deep packet inspection has higher latency than an end-host implementation.

We measured the latency for sending traffic under increasing network load. To place load on the network, we used the two background traffic machines. The client side forked n processes, each continually sending data to the server. We increased the amount of traffic by increasing the number of processes running concurrently. We computed the traffic throughput on these machines, and increased the load until we were unable to measure an increase in throughput. Traffic was generated using the datacenter traffic distribution identified by Greenberg et al. [29]. To measure latency, we used a second pair of machines. The client side sent 1000-byte probes to the server, and the server sent them back.

For each step of increasing load, we took 1000 latency measurements, and computed the 90th percentile. This eliminates extreme outliers due to packet loss and retransmission. We ran each experiment three times, and report the average results.

The results are shown in Figure 7.2. When the network is lightly loaded, the per-

formance of both systems are comparable. This is as expected, since the computation performed by end-hosts and the middlebox element are the same. The extra overhead for the middlebox case, about 0.25ms, is due to the extra hops (to and from the middlebox) that each packet needs to travel. When the network is heavily loaded, at about 800 Mbits/second, the effects of the middlebox bottleneck become manifest. The latency spikes to over 6 milliseconds, and we see an increasing number of packet drops and retransmissions. In contrast, the latency for the end-host setup stays constant at around 0.26 milliseconds. This is a 95% reduction in latency when the network is heavily loaded.

The results are exactly as one would expect, since one configuration suffers from a central bottleneck, while the other network configuration distributes the computation. However, in this case, using the shortest-path heuristic, Merlin compute an optimal placement for the network function.

Summary

7.2.3 Compilation and Verification

The scalability of the Merlin compiler and verification framework depends on both the size of the network topology and the number of traffic classes. Our third set of experiments evaluate the scalability of Merlin under a variety of scenarios.

Compiler The compilation time of the Merlin compiler was measured on three different sets of network topologies.

1. *Topology Zoo.* The Internet Topology Zoo [35] dataset contains 262 topologies that represent a large diversity of network structures. We treated each node in the Topology Zoo graph as a switch, and attached one host to each switch. The

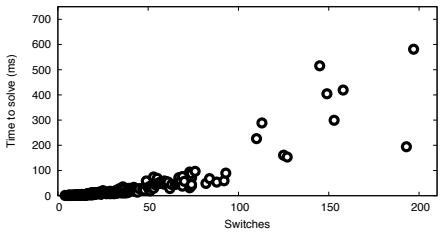


Figure 7.3: Compilation times for Internet Topology Zoo.

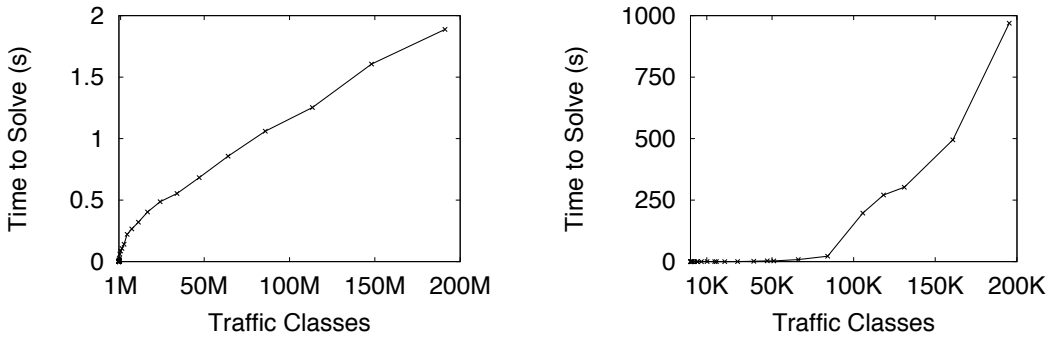


Figure 7.4: Compilation times for an increasing number of traffic classes in a balanced tree topology for (a) all pairs connectivity, (b) 5% of the traffic with guaranteed priority.

topologies have an average size of 40 switches, with a standard deviation of 30 switches. We measured the compilation time needed by Merlin to determine pair-wise forwarding rules for all hosts in each topology. In other words, the program provides basic connectivity for all hosts in the network. The results are shown in Figure 7.3.

2. *Balanced Trees.* We used the NetworkX Python software package [59] to generate balanced tree topologies. In a balanced tree, each node has n children, except the leaves. We treated internal node as switches, and leaf nodes as hosts. We varied the depth of the tree from 2 to 3, and the fanout (i.e., number of children) over a range of 2 to 24, to give us trees with varying numbers of hosts and switches. We identified each pair-wise exchange of traffic between hosts as a

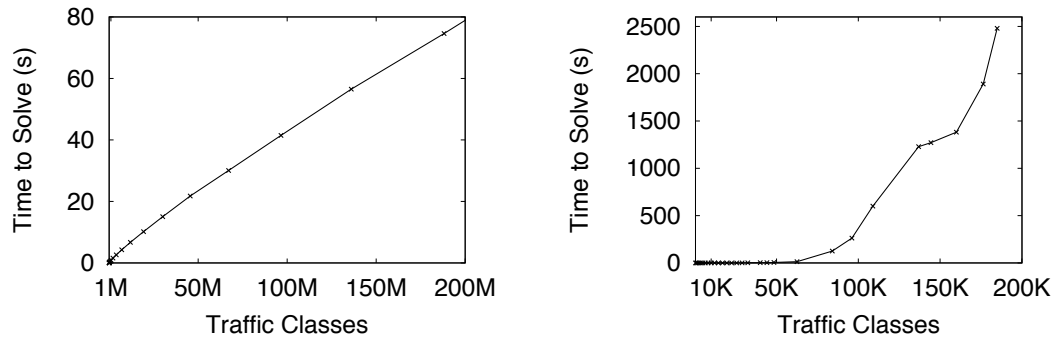


Figure 7.5: Compilation times for an increasing number of traffic classes in a fat tree topology for (a) all pairs connectivity, (b) 5% of the traffic with guaranteed priority.

separate traffic class. We measured the compilation time for two different programs for an increasing number of traffic classes. Figure 7.4 (a) shows the time to provide pair-wise connectivity with no guarantees, and Figure 7.4 (b) shows the time to provide connectivity when 5% of the traffic classes receive bandwidth guarantees.

3. *Fat Trees.* Finally, we used the NetworkX package to generate fat tree topologies [4]. A fat tree contains a set of *pods*. Each pod of size n has two layers of $n/2$ switches. To each switch in a lower layer, we attached two hosts. Each pair-wise exchange of traffic between hosts is a separate traffic class. We increased the pod size n to create larger numbers of traffic classes. Figure 7.5 (a) shows the compilation time to provide pair-wise connectivity with no guarantees, and Figure 7.5 (b) shows the time to provide connectivity when 5% of the traffic classes receive bandwidth guarantees. To provide more detail for fat tree topologies, Figure 7.6 shows a sample of topology sizes and solution times for various traffic classes, along with a finer-grained accounting of compiler time.

The results in Figure 7.3 show that for providing basic connectivity, Merlin scales well on a diverse set of topologies. The compiler finished in less than 50ms for the

Traffic Classes	Hosts	Switches	LP cons. (ms)	LP soln. (ms)	Best-Effort (ms)
870	30	45	25	22	33
8010	90	80	214	160	36
28730	170	125	364	252	106
39800	200	125	1465	1485	91
95790	310	180	13287	248779	222
136530	370	180	27646	1200912	215
159600	400	180	29701	1351865	212
229920	480	245	86678	10476008	451

Figure 7.6: Number of traffic classes, topology sizes, and details of compilation time for fat tree topologies with 5% of the traffic classes with guaranteed bandwidth.

majority of topologies, and less than 600ms for all but one of the topologies. To improve the readability of the graph, we elided the largest topology, which has 754 switches and took Merlin 4 seconds to compile. In practice, we expect that this task would be computed offline.

Figures 7.4 and 7.5 show the impact of bandwidth guarantees on compilation time. As expected, the guarantees add significant overhead. The worst case scenario that we measured, shown in Figure 7.5 (b), was a network with $184,470$ total traffic classes, with $9,224$ of those classes receiving bandwidth guarantees. Merlin took around 41 minutes to find a solution. Merlin finds solutions for 100 traffic classes with guarantees in a network with 125 switches in under 5 seconds.

Figure 7.6 shows more detail about where the compiler time is spent. The linear programming construction column measures how long it takes to create the linear programming problem. Our prototype implementation writes the problem to a file on disk before invoking the solver in a separate process. So, much of this time is attributed to string allocations and file I/O. The LP solution column measures how long it takes the solver to find a solution to the linear programming problem. As expected, this is where most of the time is spent as we increase the problem size. The Best-Effort solution column measures how long it takes to find paths with best-effort guarantees for the remaining traffic. The compiler spends little time finding paths that do not

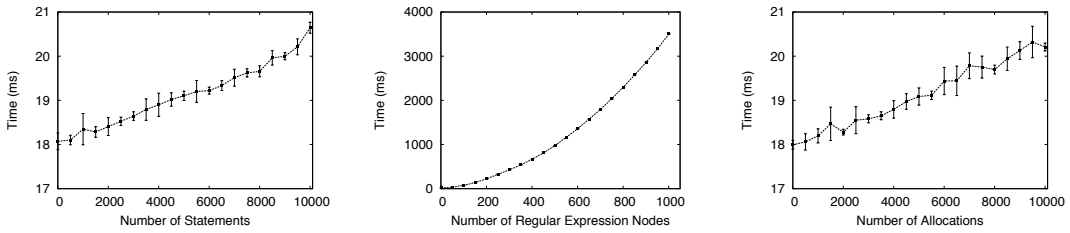


Figure 7.7: Time taken to verify a delegated program for an increasing number of delegated predicates, increasingly complex path expressions, and an increasing number of bandwidth allocations.

provide guaranteed rates.

Verifying negotiators Delegated Merlin programs can be modified by negotiators in three ways: by changing the predicates, the path expressions, or the bandwidth allocations. We ran three experiments to benchmark our negotiator verification runtime for these cases. First, we increased the number of additional predicates generated in the delegated program. Second, we increased the complexity of the path expressions in the delegated program. The number of nodes in the path expression's abstract syntax tree is used as a measure of its complexity. Finally, we increased the number of bandwidth allocations in the delegated program. For all three experiments, we measured the time needed for negotiators to verify a delegated program against the original program. We report the mean and standard deviation over ten runs.

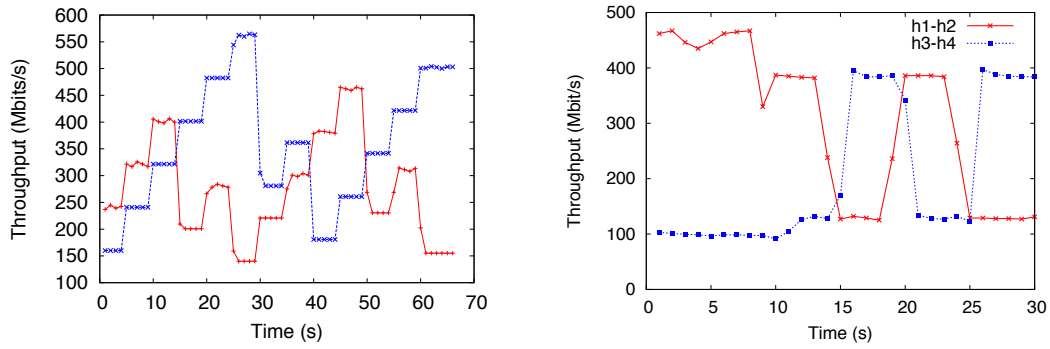
The results, shown in Figure 7.7, demonstrate that program verification is extremely fast for increasing predicates and allocations. Both scale linearly up to tens of thousands of allocations and statements and complete in milliseconds. This shows that Merlin negotiators can be used to rapidly adjust to changing traffic loads. Verification of path expressions has higher overhead. It scales quadratically, and takes about 3.5 seconds for an expression with a thousand elements. However, since path expressions denote paths through the network, it is unlikely that we will encounter path expres-

sions with thousands of nodes in realistic deployments. Moreover, we expect path constraints to change relatively infrequently compared to bandwidth constraints.

Dynamic adaptation Merlin negotiators support a wide range of resource management schemes. We implemented two common approaches: *additive-increase, multiplicative decrease* (AIMD), and *max-min fair-sharing* (MMFS). Both implementations required two components: a negotiator which ran on the same machine as the SDN controller, and end-host software, which monitors per-host bandwidth usage, and sends requests to the negotiator.

With AIMD, the end-host components send requests to the negotiator to incrementally increase their bandwidth allocation. The negotiator maintains a mapping of hosts to their current bandwidth limits. When the negotiator receives a new request, it attempts to satisfy the demand. If, however, satisfying the demand violates the global program, it then exponentially reduces the allocation for the host. After computing the new allocations, the negotiator generates updated Merlin programs, which are processed by the compiler to generate new `tc` commands that are installed on the end-hosts.

With MMFS, the end-host components declare resource requirements ahead of time by sending demands to the negotiator. The negotiator maintains a mapping of hosts to their demands. When the negotiator receives a new demand, it re-allocates bandwidth for all hosts. It does this by attempting to satisfy all demands starting with the smallest. When there is not enough bandwidth available to satisfy any further demands, the left-over bandwidth is distributed equally among the remaining tenants. Once the new allocations are computed, the negotiator generates a new program that reflects those allocations. The new program is processed by the compiler to generate new queue configurations for switches, and `tc` commands for end hosts. The queue configurations ensure that satisfied demands are respected, and the `tc` commands ensure that the



(a) (b)

Figure 7.8: (a) AIMD and (b) MMFS dynamic adaptation.

remaining traffic does not exceed the allocation specified by the original policy.

Figure 7.8 (a) shows the bandwidth usage over time for two hosts using the AIMD strategy. Figure 7.8 (b) shows the bandwidth usage over time for four hosts using the MMFS negotiators. Host h1 communicates with h2, and h3 communicates with h4. Both experiments were run on our hardware testbed.

7.2.4 Summary

Overall, these experiments show that the Merlin language can concisely express real-world programs, and that the Merlin system is able to generate code that achieves the desired outcomes for applications on real hardware. Merlin can provide connectivity for large networks quickly and our mixed-integer programming approach used for guaranteeing bandwidth scales to large networks with reasonable overhead. Negotiators allow the network to quickly adapt to changing resource demands, while respecting the global constraints imposed by the policy. Thus, Merlin strikes a good balance between expressivity and performance on real-world networks and applications.

7.3 Implementation of the EdgeNetKAT System

We have developed a prototype implementation of our edge compilation framework as well as a supporting simulation and testing system. The dyad matching algorithm, segmented paths and path constraint extensions are implemented atop the NetKAT compiler. The iteration over FDDs to produce dyads is implemented as a 400-line OCaml module. A 300-line OCaml module serializes the linear programming problems to a format understood by the Gurobi solver. This serialization module is used by both the dyad matching back-end (100 lines of OCaml) and the path constraints back-end (115 lines). Syntax extensions to support Circuit NetKAT and segmented paths are another 300 lines of OCaml.

We also develop a simulator for hybrid packet-optical networks as in Section 5.1. The open-source Linc-OE software switch simulates ROADMs, and supports an API based on the OpenFlow 1.3 protocol to set up the optical channels. The packet switches are simulated using the Mininet simulator [49], and we developed a 200-line Python extension to embed Linc-OE switches in Mininet. To control the switches in the network, we developed a packet-optical OpenFlow controller in Java using the OpenFlowJ library (500 lines of Java). The controller accepts switch forwarding tables as emitted by compiler and installs them on the simulated switches. The examples in Sections 5.2 and 6.1 have been tested using this simulator.

Finally, we developed a set of tools to generate optical fabrics and test policies based on network topologies. We use these tools to perform scalability tests on EdgeNetKAT using a real-world topology, as described in the next section.

Figure 7.9: CORONET 60-node optical topology [6].

7.4 Evaluation of the EdgeNetKAT System

We evaluate the EdgeNetKAT compiler on CORONET (Figure 7.9), a real-world optical topology with 60 nodes that is representative of current carrier networks [6]. The network stretches across the continental United States with three link-diverse cross-continental paths. We use this topology to generate realistic optical fabrics and policies and measure the compiler's performance on a variety of inputs. All experiments were run on Dell r620 servers, equipped with two eight-core 2.60 GHz E5-2650 Xeon CPUs and 64 GB of RAM running Ubuntu 14.04.1 LTS.

7.4.1 Topologies, Fabrics, and Policies

To generate the optical topology, we place a ROADM at each node in the physical topology. Then we attach packet switches to the ROADMs on the two coasts, using

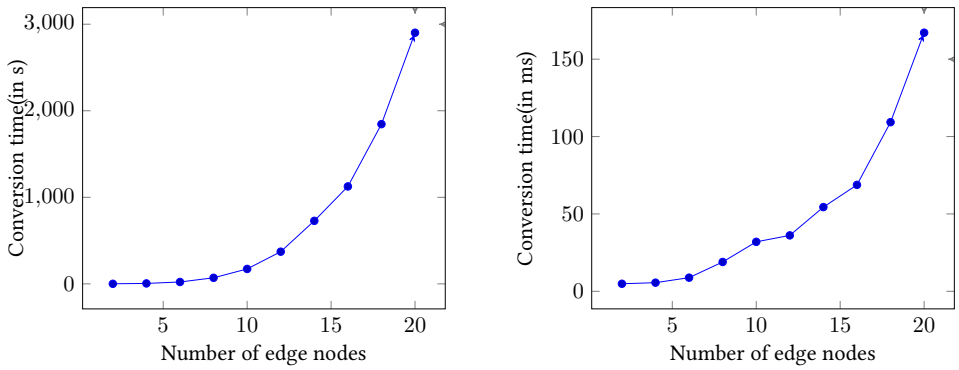
a configurable number of transponder ports. The packet switches constitute a flexible edge network to use as the target for the generated edge programs. Given this topology, we generate a range of fabrics and edge policies.

To generate the fabric, we choose a subset of ROADM nodes located along either coast. From this subset, we randomly choose pairs of nodes (R_e, R_w), such that each is on an opposite coast. We then find three paths between R_e and R_w such that each path goes over a different physically disjoint cross-continental path. For each such path, we create a unique optical channel between R_e and R_w. Using these channels, we generate a NetKAT program to implement the fabric. The program matches traffic incoming on the transponder ports (connected to the packet switches) and places them on an appropriate channel. The channels are forwarded across the network using the appropriate paths. At the egress, the program matches optical channels and outputs them to a transponder port. By increasing the number of coastal ROADMs that we connect via optical channels we can scale the size of the fabric. Using this fabric, we can generate policies to test the various parts of the EdgeNetKAT system.

To test the scalability of the basic dyad matching functionality, we generate policies to provide cross-country connectivity. We start with pairs of nodes (P_e, P_w) such that each is a packet switch on opposite coasts. We use NetKAT predicates (as described in Section 5.3) to separate out and forward a number of traffic classes between each P_e and P_w. By connecting increasing number of edge nodes, we can increase the size and complexity of the policy.

7.4.2 Dyad Generation Scalability

The first step in producing the required edge programs is to convert the fabric and the policy from NetKAT programs into dyads for the later stages. This computation is the same irrespective of how the dyads are matched. Figure 7.10(a) shows the time taken



(a) NetKAT Fabric to Dyad conversion. (b) NetKAT Policy to Dyad conversion.

Figure 7.10: Dyad conversion scalability.

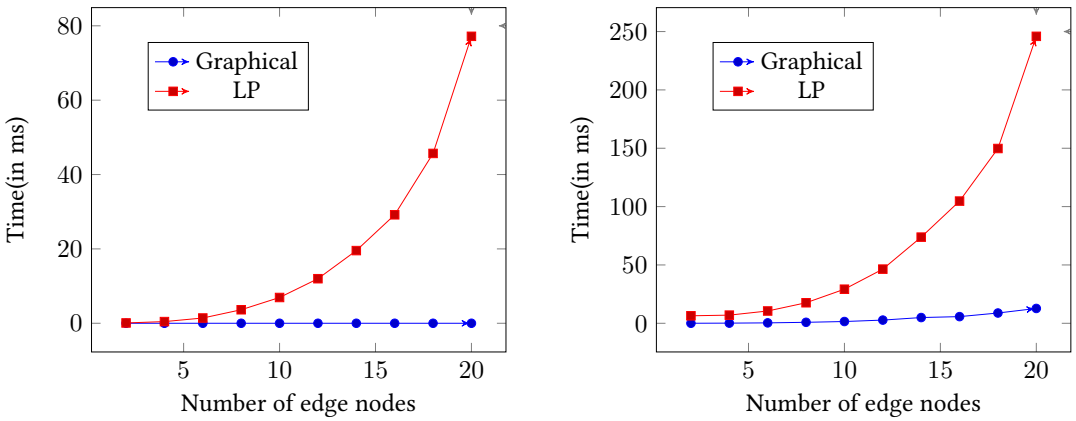
to convert the fabric into dyads using the NetKAT compiler framework. Figure 7.10(b) shows the time taken to convert the policy into dyads.

The time taken to produce the dyads for the fabric is the largest—nearly an hour for the largest fabrics. This time dominates the conversion time for the policy (less than 200 ms). However, one of the motivating constraints is that the fabric is rigid and changes infrequently, while the policy may change often. Thus the dyad conversion for the fabric could easily be performed just once and then cached for subsequent changes to the policy. Moreover, we believe its performance can be further improved by adding further optimizations to the NetKAT compiler.

7.4.3 Dyad Matching Scalability

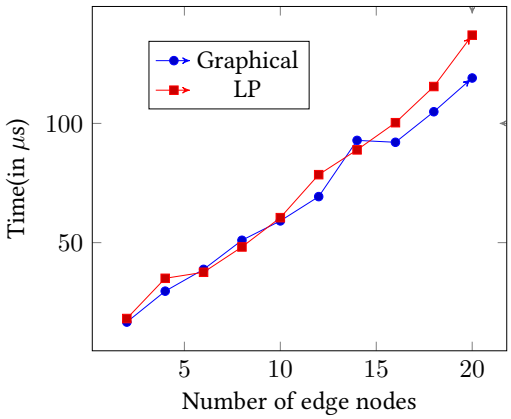
The results of running the dyad-matching back-end on increasing fabric and policy sizes is shown in Figure 7.11. Each graph plots the number of coastal nodes on the X-axis, and the time taken to complete each stage of the matching process on the Y-axis. Figures 7.11(a-c) show the time taken to generate the matching problem, the time to solve the problem, and the time to generate the NetKAT edge programs to implement the found matching respectively.

The graphical matching approach completes in microseconds, while generating the



(a) Time to create matching problem.

(b) Time to solve matching problem.



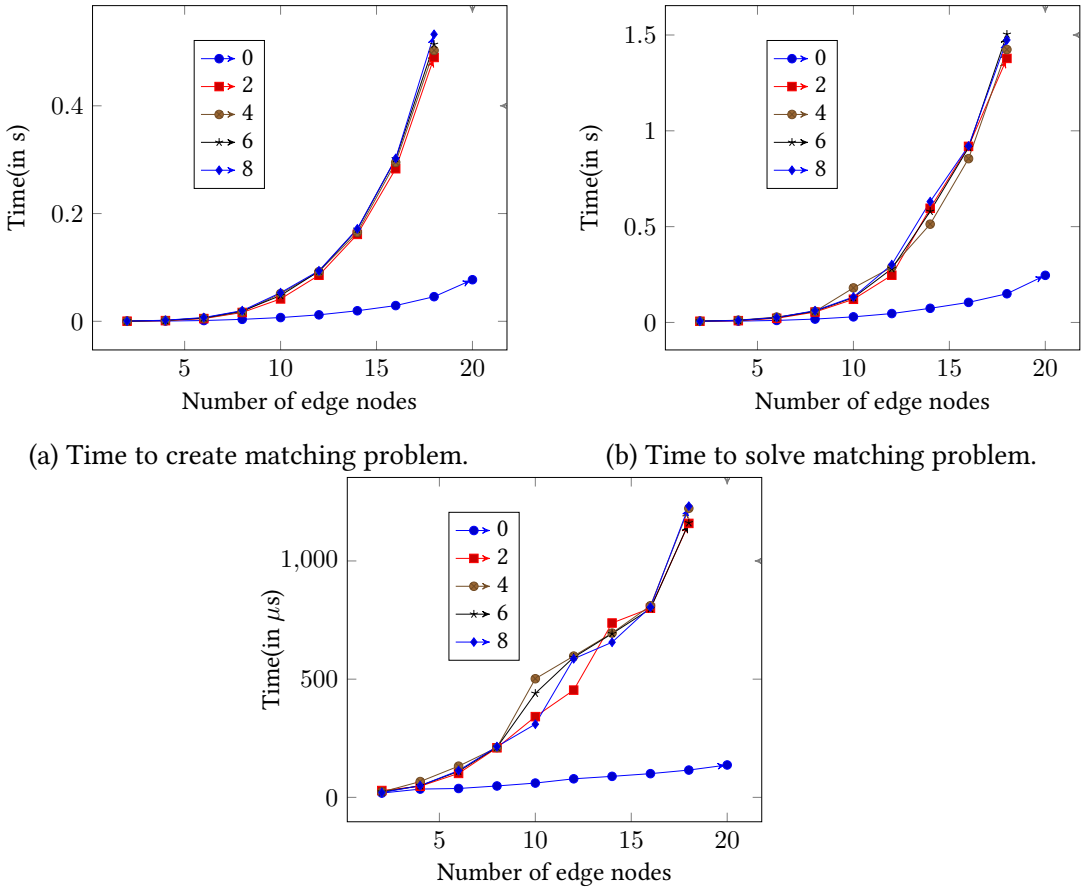
(c) Time to generate edge programs.

Figure 7.11: Scalability of linear programming and graphical dyad matching.

linear programming problem and solving it takes only hundreds of milliseconds. In either case, even very complex changes to the policy across the entire topology can be handled in a small amount of time. The time penalty paid for the linear programming approach is made up for by flexibility—generating the extra linear constraints for implementing path constraints takes only 10 lines of OCaml.

7.4.4 Path Constraint Scalability

We can now extend the basic matching evaluation with path constraints. Starting from the same fabric and policy setup, we add an increasing number of intermediate nodes



(a) Time to create matching problem.

(b) Time to solve matching problem.

(c) Time to generate edge programs.

Figure 7.12: Scalability of matching with path constraints.

to the policy. For each bicoastal pair of nodes (P_e, P_w) we add an increasing number of intermediate points drawn from one of the physical paths connecting them. Figures 7.12(a-c) show the time taken to generate the matching problem, the time to solve the problem, and the time to generate the NetKAT edge programs to implement the found matching respectively. The baseline case (0 intermediate nodes) is the dyad matching approach described in the previous section.

Using a linear programming solver allows us to support path constraints by simply adding more variables and constraints. However, adding this functionality considerably increases the size of the linear programming problems, increasing the time taken to generate, solve and recover the solution. Though the time penalty is significant in

going from the basic to the smallest number of constraints, there are no additional penalty to increasing the number of intermediate nodes. This is to be expected, since the majority of the linear programming problem is composed of variables and constraints representing the fabric paths, rather than the policy's intermediate nodes.

7.4.5 Summary

The process for generating edge programs from NetKAT fabric and policy programs can be broken down into four stages—preprocessing the programs into dyads, formulating the matching problem, solving the matching problem and finally generating edge programs from the solutions. The preprocessing step converts the programs into their dyad representation. Applying this step to the fabric takes the longest amount of time (on the order of multiple minutes to an hour). However, since our use cases are networks where the core fabric is meant to be rigid and change rarely, this step would only be performed infrequently. Formulation and solution takes on the order of milliseconds for the basic case, and less than a second if we include path constraints. Generating the final edge programs takes even less time.

These experiments suggest that EdgeNetKAT could be used to generate edge programs from network policies, provided the fabric is stable for a relatively long period of time. This is a reasonable assumption for both the optical circuit networks as well as other fabrics involving legacy, non-SDN devices [50, 12]. Thus, EdgeNetKAT provides a practical method for flexible management of heterogeneous networks.

CHAPTER 8

RELATED WORK

The preliminary design for Merlin in a workshop paper [71], and then described the approach in more detail, including an experimental evaluation, in a conference paper [72]. An expanded journal with an expanded discussion of the optimization problem and negotiator design, additional examples, and more evaluation is under review, and has been used as the basis of the Merlin-related sections in this document.

The EdgeNetKAT system leverages a number of theoretical advances made in previous work on NetKAT [5]. In addition we rely on the existing compiler infrastructure for NetKAT to produce the FDDs that our analysis takes as input [70]. This work has been published as conference paper that has been the basis for the text on heterogeneous networks and the description and evaluation of the EdgeNetKAT system.

SDN controller frameworks ONIX was an influential early SDN controller that offered a number of features including slicing and virtualization [48]. These features are also realized in more recent work on VMware NSX [47]. Exodus translates SDN policies into configurations that can be installed on legacy devices [57]. Fibbing developed approaches for implementing SDN-like control using distributed routing protocols [76].

Middleboxes and Packet Processing Functions SIMPLE [64] is a framework for controlling middleboxes. SIMPLE attempts to load balance the network with respect to TCAM and CPU usage. Like Merlin, it solves an optimization problem, but it does not specify the programming interface to the framework, or how policies are represented and analyzed.

The APLOMB [68] system allows network operators to specify middlebox processing services that should be applied to classes of traffic. The actual processing of packets is

handled by virtual machines deployed in a cloud-based architecture. Merlin is similar, in that policies allow users to specify packet-processing functions. However, Merlin does not directly target cloud-services. Moreover, Merlin allocates paths with respect to bandwidth constraints, while APLOMB does not.

E2 [61] is a framework that implements common functionality for packet-processing applications. Similar to Merlin, it provides a declarative policy-language for traffic management. E2 differs from Merlin, though, in that it is focused on Network Function (NF) management. Users provide NF descriptions, which allow the E2 software to handle tasks such as placement, scaling, and service interconnection.

Bandwidth Control Systems A number of systems in recent years have investigated mechanisms for providing bandwidth caps and guarantees [8, 69, 62, 38], implementing traffic filters [36, 66], or specifying forwarding policies at different points in the network [25, 28, 55, 32]. Merlin builds on these approaches by providing a unified interface and central point of control for switches, middleboxes, and end hosts.

Network programming languages There is a large body of work on domain-specific programming languages for SDN. Examples include Frenetic [25], Nettle [77], Pyretic [54], NetCore [55], NetKAT [5], Maple [78], and FlowLog [58], among others. The compilers for these languages translate high-level programs into switch-level forwarding rules—i.e., they assume that the network consists of SDN switches that can be frequently reprogrammed in response to changing conditions.

However, these languages are limited in that they do not allow programmers to specify middlebox functionality, allocate bandwidth, or delegate policies. An exception is the PANE [23] system, which allows end hosts to make explicit requests for network resources like bandwidth. Unlike Merlin, PANE does not provide mechanisms for partitioning functionality and delegation is supported at the level of individual flows,

rather than entire policies.

The Merlin compiler implements a form of program partitioning. This idea has been previously explored in a variety of other domains including secure web applications [15], and distributed computing and storage [52].

Optical networks The flexibility of SDNs is attractive for optical networks as well. Recent work has identified a set of challenges that are different from that in packet networks [30]. In particular, signal attenuation at long distances is a significant problem that requires the careful placement of regenerator nodes. Our path constraints extension finds paths connecting the required regenerator, while previous work has tackled the problem of where in a network to place these regenerators in the first place [42].

A number of recent efforts have developed architectures and control abstractions for hybrid packet-optical networks. REACToR incorporates optical circuits into data center network with the goal of improving performance without sacrificing control [51]. Another system, OWAN jointly manages the optical and packet layers to optimize bulk data transfers in wide-area networks [39]. Our work, which focuses on mechanisms for implementing programs at the edge, is complementary to these efforts.

Edge-fabric distinction Several existing systems are also based on an edge-fabric distinction. An early paper by Casado et al. [12] proposed an architecture based on a fabric service model. The same paper discussed the problem of mapping policies to the edge but did not propose a solution. Panopticon addresses incremental deployment of SDNs using data structures called Solitary Confinement Trees, which are similar to our fabrics [50]. However, Panopticon focuses on L2/L3 packet networks in small to medium scale enterprises, rather physically heterogenous deployments. As mentioned above, Felix is also based on an edge-fabric distinction and adopts a similar approach based on NetKAT and FDDs [14].

Software synthesis for networks Several recent systems have applied ideas from program synthesis to networks. Software synthesis is attractive because it offers the promise of finding general solutions to a wide class of problems, rather than relying on ad-hoc and possibly brittle solutions to particular problems. Genesis and SyNet synthesize device-level configurations from high-level policies that incorporate path and traffic engineering constraints [73, 21]. The EdgeNetKAT approach is also based on synthesis, but we exploit domain-specific knowledge to produce dyads by analyzing NetKAT programs, that are then fed to a solver to compute a matching. By doing so, we cut down the space of possible solutions that a solver has to investigate.

CHAPTER 9

CONCLUSION

This dissertation describes the development of a series of domain-specific languages for managing modern networks based on treating paths through the network as the core construct for describing network-wide policies. This expands on work on path-based network programming languages such as FatTire and NetKAT to support network policies involving bandwidth constraints, heterogenous network devices, and multiple administrative domains. These languages include Merlin (supporting bandwidth and delegation), Circuit NetKAT (supporting optical circuit networks) and the EdgeNetKAT compiler for NetKAT (supporting hybrid packet-optical networks). The compilers and runtimes for these languages have been tested by implementing a variety of modern applications on realistic networks. Together these languages show that it is possible to manage modern networks and specify realistic policies using path-centric, high-level, domain-specific languages.

There are a number of avenues for future work. The Merlin compiler currently uses an integrality constraint in the generated linear programming problem to produce solutions with a single path for each traffic class. Future work could leverage approaches to multi-commodity flow that take advantage of multiple paths. Though the Merlin language is designed to support delegating policies from administrators to tenants, the compiler under the control of the administrator must verify such refined policies before they are installed on the network. Using new hardware advances such as Intel's SGX, it may be possible to remove dependence on a central compiler by allowing tenants to run a hardware-attested compiler that verifies and installs refined policies.

For EdgeNetKAT, there are a number of interesting extensions: fault tolerance, load balancing and more path constraints such as node or edge disjointness. Implementing these extensions will undoubtedly require further development of the linear program-

ming based back-end, or exploring techniques based on counter-example guided inductive synthesis (CEGIS). Finally, the compiler could generate more than just the NetKAT edge programs. There may be cases where an existing fabric cannot be used to implement a given policy. In such cases, the compiler could suggest minimal extensions to the fabric required to support the policy, or conversely, precisely locate portions of the policy that the fabric cannot support.

BIBLIOGRAPHY

[1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice Hall, June 1972.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.

[3] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 63–74, August 2008.

[5] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *Symposium on Principles of Programming Languages*, pages 113–126, January 2014.

[6] Monarch Network Architects. Coronet optical topology. Available at `http://www.monarchna.com/topology.html`, October 2006.

[7] Automatic test packet generation. `https://github.com/eastzone/atpg`.

[8] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 242–253, August 2011.

[9] Cynthia Barnhart, Christopher A Hane, and Pamela H Vance. Using Branch-and-Price-and-Cut to Solve Origin-Destination Integer Multicommodity Flow Problems. *Operations Research*, 48(2):318–326, March 2000.

[10] Shrutarshi Basu, Nate Foster, Hossein Hojjat, Paparao Palacharla, Christian Skalka, and Xi Wang. Life on the edge: Unraveling policies into configurations. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '17, pages 178–190, Piscataway, NJ, USA, 2017. IEEE Press.

[11] Andrei Z. Broder, Alan M. Frieze, and Eli Upfal. Static and Dynamic Path Selection on Expander Graphs: A Random Walk Approach. In *Symposium on Theory of Computing*, pages 531–539, May 1997.

[12] Martin Casado, Teemu Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A Retrospective on Evolving SDN. In *HotSDN*, pages 85–90, August 2012.

[13] Amit Chakrabarti, Chandra Chekuri, Anupam Gupta, and Amit Kumar. Approximation Algorithms for the Unsplittable Flow Problem. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 51–66, September 2002.

[14] Haoxian Chen, Nate Foster, Jake Silverman, Michael Whittaker, Brandon Zhang, and Rene Zhang. Felix: Implementing traffic measurement on end hoses using program analysis. In *ACM SIGCOMM Symposium on Software-Defined Networking Research (SOSR)*, March 2016.

[15] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure Web Applications via Automatic Partitioning. In *Symposium on Operating Systems Principles*, pages 31–44, October 2007.

[16] Julia Chuzhoy and Shi Li. A Polylogarithmic Approximation Algorithm for Edge-Disjoint Paths with Congestion 2. In *IEEE Symposium on Foundations of Computer Science*, pages 233–242, October 2012.

[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating Systems Design and Implementation*, pages 137–150, December 2004.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Symposium on Operating Systems Principles*, pages 205–220, October 2007.

[19] Yefim Dinitz, Naveen Garg, and Michel X. Goemans. On the Single-Source Unsplittable Flow Problem. *Combinatorica*, 19(1):17–41, January 1999.

[20] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. ETTM: A Scalable Fault Tolerant Network Manager. In *Symposium on Networked Systems Design and Implementation*, pages 7–21, March 2011.

[21] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-wide configuration synthesis. *CoRR*, abs/1611.02537, 2016.

[22] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C.

Mogul. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In *Symposium on Networked Systems Design and Implementation*, pages 533–546, April 2014.

[23] Andrew Ferguson, Arjun Guha, Chen Liang, Rodrigo Fonseca, and Shriram Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 327–338, August 2013.

[24] Nate Foster, Arjun Guha, et al. The Frenetic Network Controller. In *The OCaml Users and Developers Workshop*, September 2013.

[25] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *International Conference on Functional Programming*, pages 279–291, September 2011.

[26] Alan M. Frieze. Disjoint Paths in Expander Graphs via Random Walks: A Short Survey. In *Workshop on Randomization and Approximation Techniques in Computer Science*, pages 1–14, October 1998.

[27] Aaron Gember, Prathmesh Prabhu, Zainab Ghadiyali, and Aditya Akella. Toward Software-Defined Middlebox Networking. In *Workshop on Hot Topics in Networks*, pages 7–12, October 2012.

[28] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet Routing. *SIGCOMM Computer Communication Review*, 39(4):111–122, August 2009.

[29] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 51–62, 2009.

[30] Steven Gringeri, Nabil Bitar, and Tiejun J. Xia. Extending software defined network principles to include optical transport. *IEEE Communications Magazine*, 51(3):32–40, March 2013.

[31] Gurobi Optimization Inc. The Gurobi optimizer. `http://www.gurobi.com`.

[32] T. Hinrichs, N. Gude, M. Casado, J. Mitchell, and S. Shenker. Practical Declarative

Network Management. In *Workshop: Research on Enterprise Networking*, pages 1–10, 2009.

[33] Pieter Hooimeijer. Dprle decision procedure library. `http://www.cs.virginia.edu/~ph4u/dprle/`.

[34] John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[35] The Internet Topology Zoo. `http://www.topology-zoo.org`.

[36] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. Implementing a Distributed Firewall. In *Conference on Computer and Communications Security*, pages 190–199, November 2000.

[37] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. T-Man: Gossip-based Fast Overlay Topology Construction. *Computer Networks*, 53(13):2321–2339, January 2009.

[38] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *Symposium on Networked Systems Design and Implementation*, pages 297–312, April 2013.

[39] Xin Jin, Yiran Li, Da Wei, Siming Li, Jie Gao, Lei Xu, Guangzhi Li, Wei Xu, and Jennifer Rexford. Optimizing bulk transfers with software-defined optical WAN. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 87–100, August 2016.

[40] Dilip Antony Joseph, Arsalan Tavakoli, Ion Stoica, Dilip Joseph, Arsalan Tavakoli, and Ion Stoica. A Policy-aware Switching Layer for Data Centers. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 51–62, August 2008.

[41] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *International Conference on Emerging Networking Experiments and Technologies*, pages 13–24, December 2013.

[42] Inwoong Kim, Paparao Palacharla, Xi Wang, Qiong Zhang, Daniel Bihon, Mark D. Feuer, and Sheryl L. Woodward. Regenerator predeployment in cn-roadm net-

works with shared mesh restoration. *J. Opt. Commun. Netw.*, 5(10):A213–A219, Oct 2013.

[43] Jon Kleinberg and Ronitt Rubinfeld. Short Paths in Expander Graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 86–95, October 1996.

[44] Jon M. Kleinberg. Single-Source Unsplittable Flow. In *IEEE Symposium on Foundations of Computer Science*, pages 68–77, October 1996.

[45] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *Transactions on Computer Systems*, 18(3):263–297, August 2000.

[46] Stavros G. Kolliopoulos and Clifford Stein. Approximation Algorithms for Single-Source Unsplittable Flow. *SIAM Journal on Computing*, 31(3):919–946, June 2001.

[47] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Jesse Gross Igor Ganichev, Natasha Gude, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, , Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. Network virtualization in multi-tenant datacenters. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014*, pages 203–216, April 2014.

[48] Teemu Koponen, Martín Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010*, pages 351–364, October 2010.

[49] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[50] Dan Levin, Marco Canini, Stefan Schmid, Fabian Schaffert, and Anja Feldmann. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *2014 USENIX Annual Technical Conference, USENIX ATC '14*, pages 333–345. USENIX Association, June 2014.

[51] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with REACToR. In *NSDI*, pages 1–15, 2014.

[52] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A Platform for Secure Sistributed Computation and Storage. In *ACM SIGOPS European Workshop*, pages 321–334, October 2009.

[53] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[54] Christopher Monsanto et al. Composing Software-Defined Networks. In *Symposium on Networked Systems Design and Implementation*, pages 1–13, April 2013.

[55] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A Compiler and Run-time System for Network Programming Languages. In *Symposium on Principles of Programming Languages*, pages 217–230, January 2012.

[56] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[57] Tim Nelson, Andrew D. Ferguson, Da Yu, Rodrigo Fonseca, and Shriram Krishnamurthi. Exodus: Toward automatic migration of enterprise network configurations to sdns. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15*, pages 13:1–13:7. ACM, June 2015.

[58] Tim Nelson, Michael Scheer, Andrew D. Ferguson, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. In *Symposium on Networked Systems Design and Implementation*, April 2014.

[59] NetworkX. `https://networkx.github.io`.

[60] Haruko Okamura and Paul D. Seymour. Multicommodity Flows in Planar Graphs. *Journal of Combinatorial Theory, Series B*, 31(1):75–81, 1981.

[61] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for nfv applications. In *Symposium on Operating Systems Principles*, pages 121–136, October 2015.

[62] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 187–198, August 2012.

[63] Puppet. `http://puppetlabs.com`.

[64] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 27–38, August 2013.

[65] Mark Reitblatt, Marco Canini, Nate Foster, , and Arjun Guha. FatTire: Declarative Fault Tolerance for Software Defined Networks. In *HotSDN*, August 2013.

[66] Martin Roesch. Snort—Lightweight Intrusion Detection for Networks. In *Conference on System Administration*, pages 229–238, November 1999.

[67] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Symposium on Networked Systems Design and Implementation*, pages 24–38, April 2012.

[68] 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 *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 13–24, August 2012.

[69] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *Workshop on Hot Topics in Cloud Computing*, pages 1–8, June 2010.

[70] Steffen Smolka, Spiridon Eliopoulos, Nate Foster, and Arjun Guha. A fast compiler for NetKAT. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015*, pages 328–341. ACM, September 2015.

[71] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the Network with Merlin. In *Workshop on Hot Topics in Networks*, November 2013.

[72] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Merlin: A Language for Provisioning Network Resources. In *International Conference on Emerging Networking Experiments and Technologies*, December 2014.

[73] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *POPL*, pages 572–585, 2017.

[74] Rajagopal Subramaniyan, Pirabhu Raman, Alan D. George, Matthew A. Radlinski, and Matthew A. Radlinski. GEMS: Gossip-Enabled Monitoring Service for Scalable Heterogeneous Distributed Systems. *Cluster Computing*, 9(1):101–120, January 2006.

[75] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *Transactions on Computer Systems*, 21(2):164–206, February 2003.

[76] Stefano Vissicchio, Olivier Tilmans, Laurent Vanbever, and Jennifer Rexford. Central Control Over Distributed Routing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015*, pages 43–56. ACM, August 2015.

[77] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. In *13th PADL*, volume 6539 of *LNCS*, pages 235–249, 2011.

[78] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 87–98, August 2013.