Abstract

Programmable and Scalable Software-Defined Networking Controllers

Andreas Richard Voellmy
2014

A major recent development in computer networking is the notion of Software-Defined Networking (SDN), which allows a network to customize its behaviors through centralized policies at a conceptually centralized network controller. The SDN architecture replaces closed, vertically-integrated, and fixed-function appliances with general-purpose packet processing devices, programmed through open, vendor-neutral APIs by control software executing on centralized servers. This open design exposes the capabilities of network devices and provides consumers with increased flexibility.

Although several elements of the SDN architecture, notably the OpenFlow standards, have been developed, writing an SDN controller remains highly difficult. Existing programming frameworks require either explicit or restricted declarative specification of flow patterns and provide little support for maintaining consistency between controller and distributed switch state, thereby introducing a major source of complexity in SDN programming.

In this dissertation, we demonstrate that it is feasible to use arguably the simplest possible programming model for centralized SDN policies, in which the programmer specifies the forwarding behavior of a network by defining a packet-processing function as an ordinary algorithm in a general-purpose language. This function, which we call an algorithmic policy, is conceptually executed on every packet in the network and has access to centralized network and policy state. This programming model eliminates the complex and performance-critical task of generating and maintaining sets of rules on individual, distributed switches.

To implement algorithmic policies efficiently, we introduce Maple, an SDN programming framework that can be embedded into any programming language with appropriate support. We have implemented Maple in both Java and Haskell, including an optimizing compiler and runtime system with three novel components. First, Maple's optimizer automatically discovers reusable forwarding decisions from a generic running control program. Specifically, the optimizer observes algorithm execution traces, organizes these traces to develop a partial decision tree for the algorithm, called a trace tree, and incrementally compiles these trace trees into optimized flow tables for distributed switches. Second, Maple introduces state dependency localization and fast repair techniques to efficiently maintain consistency between algorithmic policy and distributed flow tables. Third, Maple includes the McNettle OpenFlow network controller that efficiently executes user-defined OpenFlow event handlers written in Haskell on multicore CPUs, supporting the execution of algorithmic policies that require the central controller to process many packets. Through efficient message processing and enhancements to the Glasgow Haskell Compiler runtime system, McNettle network controllers can scale to handle over 20 million OpenFlow events per second on 40 CPU cores.

Programmable and Scalable Software-Defined Networking Controllers

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Andreas Richard Voellmy

Dissertation Directors: Paul Hudak & Y. Richard Yang

May 2014

Dedicated to my parents, Richard and Alice Voellmy

Contents

List of Figures

List of Tables

Acknowledgements

This dissertation would not have been possible without the support and help of many talented and generous people.

I am extremely grateful to both of my advisors, Paul Hudak and Y. Richard Yang. Paul Hudak's work on Haskell, functional programming, and domain-specific languages inspired me to pursue academic research on programming languages and to obtain my Ph.D. I thank Paul Hudak for giving me this wonderful opportunity, for mentoring me patiently over many years, and for inviting me to join him in applying programming language techniques to solve problems in computer networks.

I owe a special debt of gratitude to Y. Richard Yang for his extraordinary mentorship and collaboration, especially during the latter part of my dissertation. Richard's ability to identify the key challenges of a problem and the major insights of an initial solution were crucial in developing the full potential of our early observations and greatly influenced the way that I approach research problems. His emphasis on clarity helped me develop my technical and presentation skills. I thank Richard for believing in the potential of the work and for encouraging me to be ambitious.

I am grateful to the members of my dissertation committee, Paul Hudak, Y. Richard Yang, Bryan Ford, and Jennifer Rexford, who provided essential guidance and feedback on my papers, talks, and dissertation.

I have been fortunate to collaborate with a number of talented individuals during my time at Yale. I am grateful to Vijay Ramachandran, John Launchbury and the rest of the Galois Inc. Nettle team for collaborating on the original Nettle-BGP project and for bravely agreeing to change the direction of the project to software-defined networking and OpenFlow at a time when this strategy seemed risky. I am also grateful to Ashish Agarwal for the many helpful discussions and contributions to the `nettle-openflow` and `nettle-frp` Haskell libraries, both of which were developed during this initial phase of work.

I am grateful to Nick Feamster, Hyojoon Kim, and Sam Burnett for collaborating on the Resonance project, in which we used functional reactive programming to specify dynamic network security policies.

I am grateful to a number of collaborators on the Maple, McNettle, and Mio projects. I am particularly grateful to Junchang Wang, my primary collaborator on these project, both for his friendship and for his numerous contributions, especially relating to the operating system and hardware challenges that we faced. I thank Michael Nowlan and Lewen Yu for working with me in building the Java-based API to the Maple system. I thank Bryan Ford and Ramakrishna Gummadi for their

advice on the design of McNettle and for providing the servers which made the McNettle work possible. I am grateful to Kazuhiko Yamamoto for collaborating on the Mio project, especially his contributions to the Mio implementation, including the FreeBSD implementation.

I thank Yale Information Technology Services, in particular David Galassi, Richard Beebe, and Dean Baruffi for the many conversations about the design and management of campus networks and for generously lending network equipment that enabled us to perform various key experiments.

I am grateful to the entire Haskell community for designing, building and maintaining fantastic open source software and for building an unusually positive, welcoming and helpful community. In particular, I thank Simon Marlow, Bryan O'Sullivan and Johan Tibell for helping me get started with GHC's runtime system and for providing key feedback on the design and implementation of Mio. Thanks to Edsko de Vries for providing benchmarking tools which helped us evaluate Mio.

This work was supported in part by STTR grants ST061-002 and YU-STTR-C0393 from the Defense Advanced Research Projects Agency, by a gift from Futurewei, and by NSF grants CNS-101720, CNS-0720682, and CCF-0728448. I would also like to thank Yale University and the Department of Computer Science, for supporting me during these years and making this work possible.

I am deeply grateful to my parents and siblings for their love, wisdom, and unwavering support. Facing one particular dilemma, my father counseled me to "have courage". This simple advice became a touchstone for me as I rallied energy to cope with various difficulties. I am grateful to my parents-in-law, Jane and Allan Paulson, who supported me in so many ways.

Finally, I thank my wife Ariana for her love, encouragement and patience during this long process, and for constantly inspiring me with the passion and commitment that she brings to her own work.

Chapter 1

Introduction

A major recent development in computer networking is the notion of Software-Defined Networking (SDN), which allows a network to customize its behaviors through centralized policies at a conceptually centralized network controller. The SDN architecture replaces closed, vertically-integrated, and fixed-function appliances with general-purpose packet processing devices, programmed through open, vendor-neutral APIs by control software executing on centralized servers. This open design exposes the capabilities of network devices and provides consumers with increased flexibility in managing and deploying network services.

In particular, the OpenFlow protocol [41] has made significant progress by establishing (1) flow tables as a standard data-plane abstraction for distributed switches, (2) a protocol allowing the centralized controller to install forwarding rules and query state at switches and for a switch to notify the controller when packets fail to match rules in its switch-local forwarding table. The OpenFlow forwarding abstraction is expressive enough to allow a variety of packet processing functions to be expressed, such as L2 or IP forwarding, multicast forwarding, access control, load balancing, traffic policing, and quality-of-service forwarding. The uniform abstraction provides a simplified interface for centralized network control applications and allows services to be deployed uniformly throughout the network.

The OpenFlow protocol and related systems have provided critical components in realizing the vision that a network operator configures a network by writing a simple, centralized network control program, with a global view of network state. We refer to the programming of the centralized controller as *SDN programming*, and a network operator who conducts SDN programming as an *SDN programmer*, or just programmer.

Despite OpenFlow's progress, a major remaining component in realizing SDN's full benefits is the SDN control system and programming framework: the programming language, programming abstractions, and run-time system. Existing solutions impose a substantial burden by requiring either explicit or restricted, declarative specification of flow patterns, introducing a major source of complexity in SDN programming. For example, SDN programming using NOX [23] requires that a programmer explicitly create and manage flow rule patterns and priorities. Frenetic [21] introduces higher-level abstractions but requires restricted declarative queries and policies as a means for

introducing switch-local flow rules. However, as new use cases of SDN continue to be proposed and developed, a restrictive programming framework forces the programmer to think within the framework's - rather than the algorithm's - structure, leading to errors, redundancy and/or inefficiency.

The central thesis of this dissertation is that *it is feasible,* i.e. *effective and practical, to use what is arguably the simplest and most obvious possible programming model for centralized SDN policies: the programmer specifies the forwarding behavior of a network by defining a packet-processing function, f, as an ordinary algorithm in a general-purpose language that is conceptually executed on every packet in the network and has access to centralized state.* In creating the function f, the programmer would not need to adapt to a new programming model but would instead use standard programming languages to design arbitrary algorithms for forwarding input packets. We refer to this model as *SDN programming of algorithmic policies.* We note that algorithmic and declarative policies are not mutually exclusive. In fact, we show in Chapter 6 that algorithmic policies can be used as an effective implementation technique for declarative policies.

The promise of algorithmic policies is a simple and flexible conceptual model, but this simplicity may introduce performance bottlenecks if naively implemented. Conceptually, in this model, the function f is invoked on every packet, leading to a serious computational bottleneck at the controller; that is, the controller may not have sufficient computational capacity to invoke f on every packet. Also, even if the controller's computational capacity can scale, the bandwidth demand that every packet go through the controller will be impractical. These bottlenecks are in addition to the extra latency of forwarding all packets to the controller for processing [15].

Therefore, an efficient implementation of algorithmic policies must distribute the execution of the packet processing function so that the vast majority of packets are processed locally in SDN switches, rather than at a centralized server. There are three main challenges that we must overcome to accomplish this goal: (1) generate high-quality flow tables, (2) maintain consistency between distributed flow tables and algorithmic policy, and (3) scale centralized execution.

Generate high-quality flow tables. Generating high quality flow tables that handle a large fraction of network traffic is a complex and error-prone task. In particular, OpenFlow switches have limited capacity for rules in their flow tables, ranging from under 1,000 in commodity switches and up to 1,000,000 in state-of-the-art, high-end devices. Since packets that are not handled by flow tables will be diverted to the central controller, it is critical to use available rule capacity to apply local rules to as much traffic as possible. OpenFlow provides several features, including overlapping and prioritized rules that aid in constructing compact rule sets. However, reasoning about the behavior of such rule sets is challenging and generating them to achieve a given high-level policy is error-prone. Errors in flow tables may lead to serious consequences, such as security lapses and inadequate quality of service for mission-critical network applications.

Maintain consistency between flow tables and algorithmic policy. Since

flow table rules do not have access to global state, the network controller must not only generate flow tables, but also maintain them as network conditions and policies change. For example, rules that were installed when all network links were functioning normally may no longer be valid after a link failure. Other rules may implement access control policies and may need to be updated when an administrator changes the policy or a device reaches specified bandwidth limits. Naive approaches, such as removing all rules or recompiling the entire network policy when any network condition changes, are too disruptive and may perform poorly. Targeted and specific updates, such as deleting rules forwarding to a failed port, can perform better, but are more challenging to implement correctly and may introduce errors, as we will show in Section 3.2.

Scale centralized execution. Some algorithmic policies are inherently difficult to distribute to switch flow tables and hence will require more packets to be processed centrally, leading to scalability challenges. For example, Ethane [12] enforces a centralized security policy on each new application (i.e. TCP) flow, requiring at least one centralized execution of the algorithmic policy per TCP flow. Hence, to cope with a variety of packet processing functions or large networks, the central server must scale execution gracefully if the SDN as a whole is to scale.

1.1 Contributions

We introduce Maple, an SDN programming framework consisting of a domain specific language that can be embedded into any programming language with appropriate support. We have implemented Maple in both Java [22] and Haskell [33], including an optimizing compiler and a highly scalable, multicore runtime system. As a result, SDN programmers can enjoy simple, intuitive SDN programming, while achieving high performance and scalability. Maple introduces three novel components to overcome the aforementioned challenges:

Tracing Runtime System. First, Maple introduces a novel SDN *optimizer* that "discovers" reusable forwarding decisions from a generic running control program. Specifically, the optimizer develops a data structure called a *trace tree* that records the invocation of the programmer-supplied f on a specific packet, and then generalizes the dependencies and outcome to *other* packets. As an example, f may read only one specific field of a packet, implying that the policy's output will be the same for any packet with the same value for this field. A trace tree captures the reusability of previous computations and hence substantially reduces the number of invocations of f and, in turn, the computational demand, especially when f is expensive.

The construction of trace trees also transforms arbitrary algorithms to a normal form (essentially a cached data structure), which allows the optimizer to achieve *policy distribution*: the generation and distribution of switch-local forwarding rules, totally transparently to the SDN programmer. By pushing computation to distributed switches, Maple significantly reduces the load on the controller as well as the latency. Its simple, novel translation and distribution technique optimizes individual switch

flow table resource usage. Additionally, it considers the overhead in updating flow tables and takes advantage of multiple switch-local tables to optimize network-wide forwarding resource usage.

State Dependency Tracking. Second, Maple performs run-time state dependency analysis to automatically maintain consistency between an algorithmic policy and switch flow tables in the face of network dynamics and classifier and policy dynamics. In particular, Maple provides a number of data types that can be used to model network state and allows both external programs and the packet-processing function to update this state. Maple records dependencies of cached executions on state components to localize the effects of state changes and uses speculative reevaluation and update minimization to rapidly update flow tables after network or policy changes.

Scalable centralized execution on multicore CPUs. Third, Maple introduces a scalable *OpenFlow message service layer*, called McNettle, to complement flow table policy distribution. McNettle leverages large multicore servers to execute user-defined event processing logic written in Haskell on up to 40 CPU cores, handling networks with thousands of switches and 20 million OpenFlow events per second. McNettle achieves scalable performance by reducing synchronizations in message parsing and memory management and through enhancements to the Glasgow Haskell Compiler (GHC) multicore runtime system. In particular, we develop the Multicore IO Manager (Mio), a component of the GHC runtime system, now part of GHC 7.8.1, that substantially improves the IO performance of many network programs — including OpenFlow servers — written in Haskell.

We prove the correctness of our key techniques, describe a complete implementation of Maple, and evaluate Maple through benchmarks. For example, using HP switches, the Maple optimizer reduces HTTP connection time by a factor of 100 at high load. McNettle scales to 40+ CPU cores, achieving a simulated throughput of over 20 million requests/sec on a single machine.

1.2 Organization

The rest of this dissertation is organized as follows. In Chapter 2 we present relevant background and context for this dissertation. In particular, this chapter presents both traditional and SDN architectures and presents the critical role of the OpenFlow standards in SDN. In addition, it discusses the main approaches available today for structuring the network control plane software of an SDN system.

Chapter 3 introduces the algorithmic policy SDN programming model and compares it to existing approaches for defining network policy. This chapter then presents a high-level overview of the key techniques that enable efficient implementation of the algorithmic policy programming model.

Chapters 4 and 5 describe the Maple tracing runtime system, including detailed algorithms and data structures. Chapter 4 presents the methods used by Maple to generate compact flow tables for all OpenFlow switches in a network, and includes

evaluations of the effectiveness of the generated OpenFlow flow tables and the performance of the Maple tracing runtime system. Chapter 5 presents and evaluates the methods used by Maple to automatically synchronize algorithmic policies with flow tables after network or policy state changes occur.

Chapter 6 presents several complete examples of network policies implemented in Maple, including policies that perform broadcasting, shortest path routing, access control with declarative ACLs, and traffic monitoring.

Chapter 7 presents McNettle, a high-performance OpenFlow messaging service written in Haskell. The chapter covers the API and architecture of McNettle and discusses key implementation techniques enabling scalable execution on multicore CPUs. Chapter 8 presents the Multicore IO Manager (Mio), an enhancement ot the Glasgow Haskell Compiler's (GHC) runtime system, which allows programmers to write high-performance, low latency network servers, such as McNettle, in Haskell.

Finally, Chapter 9 summarizes the dissertation, discusses limitations and presents directions for future work.

Chapter 2

Background & Related Work

Software-Defined Networking (SDN) represents a significant departure from traditional network architecture. In particular, SDN proposes an open architecture, in which the network control and forwarding functions are decoupled. This architecture enables automated network control and management to be programmed on standard servers through a uniform interface to physical network elements. Since network control is no longer included in each network element, SDN introduces a new component: the centralized SDN controller.

In this chapter, we review traditional network architecture, present SDN architecture, and introduce key aspects of the OpenFlow protocol, which enables controller-switch interaction through a standardized protocol. In addition, we review current approaches to the organization of the network control plane software for OpenFlow networks.

2.1 Forwarding and Routing

A computer network consists of a collection of devices, called *nodes*, connected by various forms of physical media, such as copper wires or optical fibers. Moreover, most computer networks are packet-switched: endhosts transmit digital information grouped into small packets, with each packet carrying a header that enables a node to process the packet appropriately. Typically, a node must determine how to *forward* a packet towards its ultimate destinations. Hence, the packet header will contain enough information to identify the destination nodes.

In most networks, a distinction is made between two types of nodes: *endhosts* that communicate with each other, and *network elements*, such as switches and routers, that make up the network infrastructure and provide network services to endhosts. For example, Figure 2.1 depicts a simple, wired network with 5 network elements and 4 endhosts.

The primary function of a network is to transport packets, and this task is typically decomposed into two subtasks: *routing* and *forwarding*. The goal of the routing task is to determine — on the basis of the currently available nodes, the state of their interconnections, and perhaps many other factors — the routes that packets should

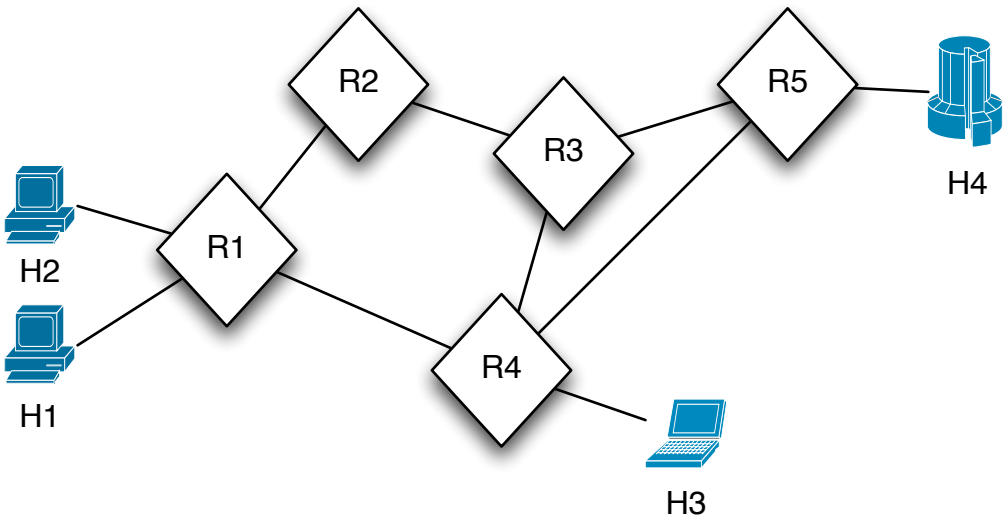


Figure 2.1: A simple network with 5 network elements and 4 endhosts.

follow while traversing the network and to compute how each node in the network should forward traffic, so that the overall forwarding behavior is achieved. Note that the routing task must be running continuously because the state of the network may change at any moment; for example, nodes and links fail without warning or congestion may build up due to unexpected traffic patterns.

While the routing task may involve complex algorithms, the forwarding task, on the other hand, is conceptually very simple: each node must forward packets on the basis of the information provided by the routing task for that node and the labels attached to the packet itself. The interface between the routing and forwarding tasks is a collection of data structures, called the *Forwarding Information Base (FIB)*, one at each switch. The routing process is responsible for maintaining the state of the FIB, while the forwarding process uses the table to determine how to forward any given packet. Figure 2.2 illustrates this organization inside a single network element.

The division of labor into forwarding and routing allows network devices to implement optimized components for each task. In fact, the extreme performance requirements of modern networks demand highly specialized techniques for forwarding packets. For example, a switch must forward each packet on a 10 Gbps port within 10 nanoseconds (ns) on average. At this speed, expensive operations, such as DRAM memory accesses must be almost entirely eliminated and specialized hardware memories and data structures are essential. As a result, forwarding engines are designed around search data structures that optimize the number of memory accesses required to find a value for a given key.

2.1.1 Traditional Architecture

Clearly, part of the forwarding task must be executed on each network element. In traditional network architecture, the routing task is also executed on each network element independently, even though each individual element may need information

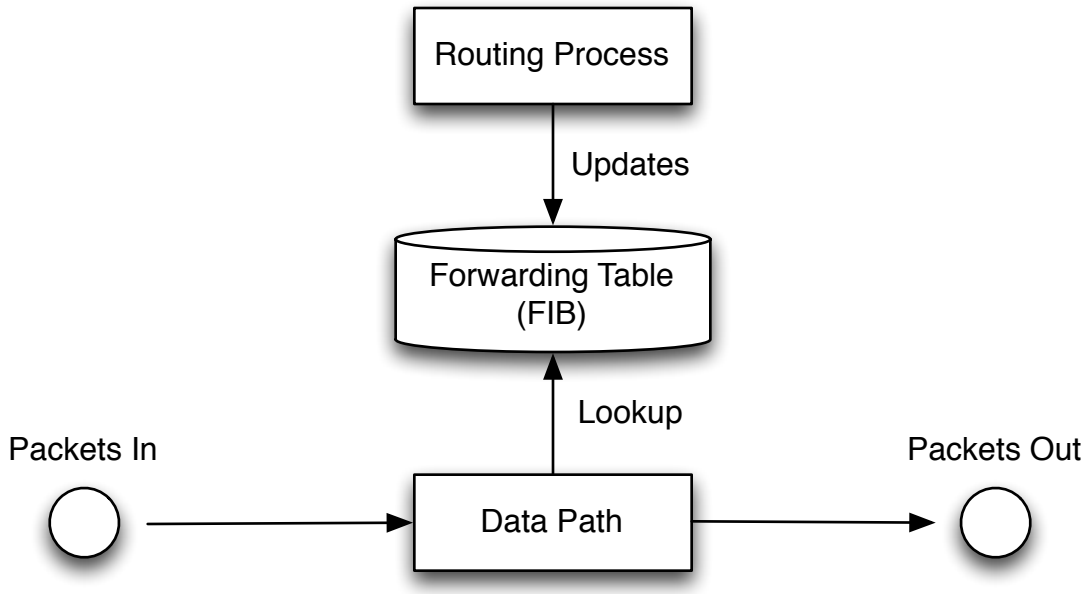


Figure 2.2: Routing and forwarding tasks in a network element.

derived from other elements in order to perform its role in correctly forwarding packets. Typically, this is done by having each network element announce all the local information that any other network elements may need in order to determine the global state. For example, each network element may announce to all other network elements which neighbors it is directly connected to and the destination addresses of all directly connected endhosts. Each network element then independently calculates the desired routes, determines its local role in forwarding packets along those routes, and then configures its local FIB accordingly. Since state can change at any time, network elements announce updates to their local state and receive updates from other elements.

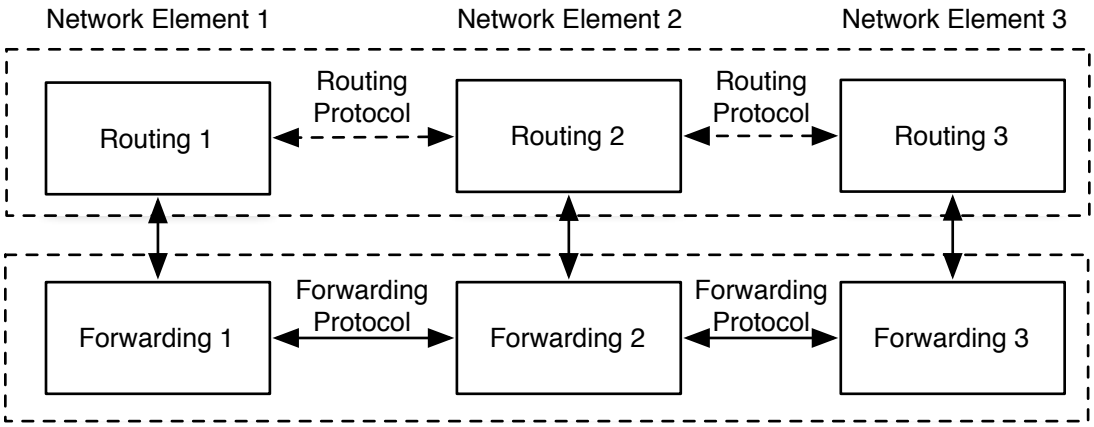


Figure 2.3: Forwarding and Control planes in a traditional network.

Figure 2.3 illustrates this design with three network elements, with each element divided into routing and forwarding components. The interface between the routing and forwarding processes of a single network element includes messages sent by the routing process to configure the FIB of the element, but also allows the routing process to send and receive packets to and from the routing processes of other elements. Conceptually, the routing and forwarding tasks each form a *layer*, or *plane*, across all the devices in the network. That is, the routing processes of all nodes logically communicate and coordinate among themselves to configure the forwarding processes while the forwarding processes of all routers cooperate to move packets from one location to another.

This symmetric design, in which each router plays an equal role in the network, contributes substantially to the robustness of modern network systems. In such a system, a single node may fail without impacting the rest of the nodes beyond the loss of connectivity caused by that failure.

With this design, the network elements interact via the protocols between routing elements. Therefore, the FIB of each network element and the interface between the routing process and the FIB can be proprietary and device-specific. This has the advantage that device makers can freely innovate within their devices without breaking inter-operability. On the other hand, the use of proprietary interfaces constrains innovation in networks, since users may not implement new routing methods directly on devices. Instead, a new protocol must be established and implemented by multiple vendors. Since multiple vendors are involved, this implementation typically requires international standardization, which can take years to establish.

2.1.2 SDN Architecture

SDN proposes to establish an industry-wide standard "forwarding abstraction" for the FIB and a standardized protocol allowing a higher-level process to configure the FIB and to interact with the forwarding element. The forwarding abstraction and protocol would allow new network management methods (e.g. routing algorithms) to be implemented on devices immediately, without requiring an expensive and slow standardization process. This change promises to greatly accelerate the pace of innovation in networks.

The SDN forwarding plane abstraction should be flexible enough so that a variety of network control algorithms can be implemented with it. For example, it would not suffice to have a forwarding plane that only forwards packets based on Ethernet destination address, since some applications may need to forward on the basis of source addresses as well. Similarly, it would not be sufficient if the FIB only allows the routing process to associate a single outgoing port with a given packet, since some applications may need the ability to broadcast or multicast.

Furthermore, if the forwarding abstraction is general enough to support a variety of functions beyond simple L2 or L3 forwarding, for example access control rules and traffic monitoring, then many specialized network devices may be eliminated. The use of more generalized packet processing devices may lead to simplified network management, improved utilization of equipment and decreased capital expenses for

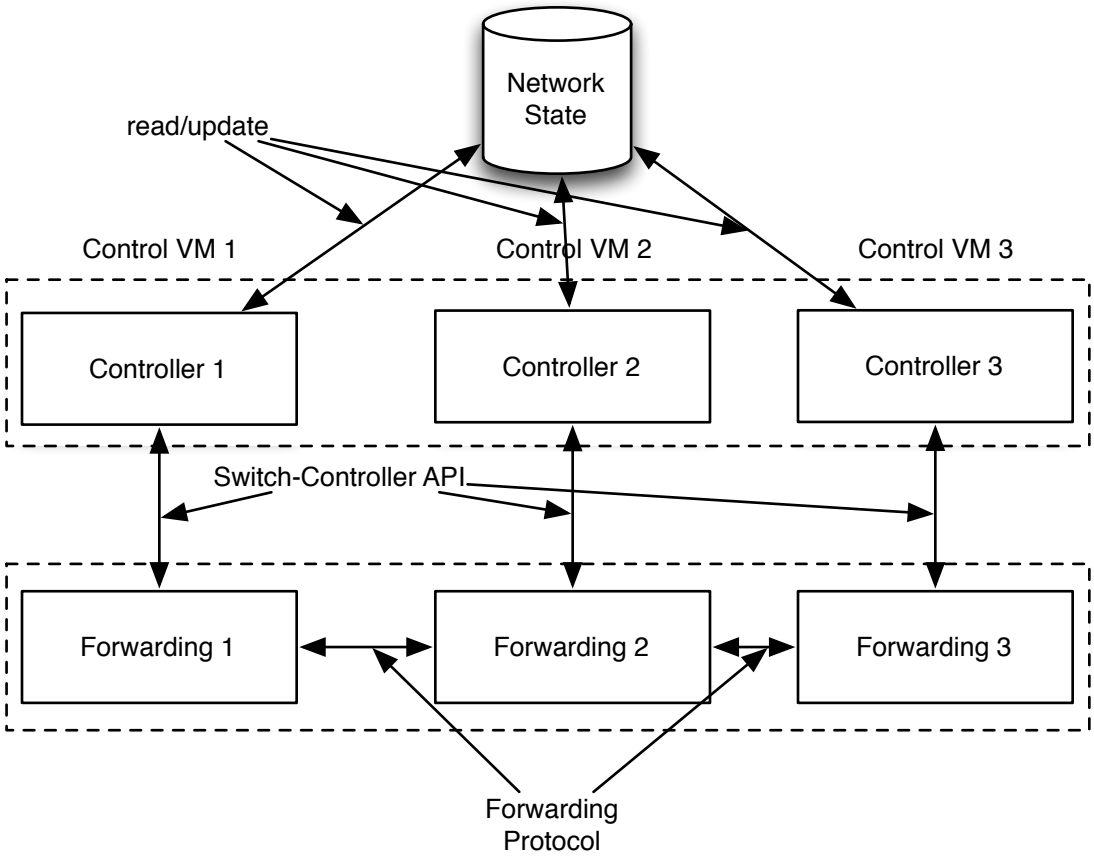


Figure 2.4: System architecture for SDN with virtualized routing functions.

network operators.

The introduction of a forwarding abstraction and a networked configuration protocol enables a further benefit: forwarding elements can be simplified by removing the higher-level routing processes from the network element. Instead, those functions can be implemented on standard computational elements, such as commodity x86 servers, that run routing algorithms remotely and that use a standard controller-switch protocol to configure the forwarding element. This simplification may lead to streamlined network elements and lower costs for high-performance network hardware. In addition, standard approaches to building fault-tolerant services in data centers can be applied to build the routing plane.

The forwarding abstraction and protocol allows for a variety of approaches for implementing the network control system. Figure 2.4 illustrates one possible organization in which the controller for each switch has been replicated into a virtualized environment and with each switch's controller having access to a global database to support global coordination. Other designs are possible as well: for example, the controller could be a single program running on a centralized server controlling all devices in the network.

Priority	Match	Action
2	`tcp_dst:22`	`forward [1]`
1	`eth_dst:0xabababababab`	`forward []`
1	`eth_dst:0xcdcdcdcdcdcd`	`forward [2,3]`
0	`eth_dst:0xefefefefefef`	`set(eth_src=0); forward [1]`

Figure 2.5: Example OpenFlow flow table.

2.1.3 OpenFlow

The OpenFlow standard has recently emerged as the de facto industry standard SDN forwarding abstraction and switch-controller protocol. An OpenFlow switch initially establishes a communication channel over a TCP connection to a single remote *controller* at a specific IP address and TCP port. The OpenFlow protocol is then used to exchange information between controller and switch and to allow the controller to configure the switch's *flow table*[1], which determines its forwarding behavior.

A flow table consists of a collection of prioritized rules, where each rule consists of a match condition that defines which packets the rule applies to, and an action that defines how packets that match this rule should be processed. When a switch receives a packet, it finds the highest priority rule whose match condition matches the packet. It then executes the actions associated with the selected rule. If no rules match the packet, the packet is encapsulated in an OpenFlow message and sent to the controller.[2] Figure 2.5 shows a small example flow table with four rules. The first rule has the highest priority and is tested first. It applies to packets whose TCP destination port field has value 22. The remaining three rules apply to packets with Ethernet destinations `0xabababababab`, `0xcdcdcdcdcdcd`, and `0xefefefefefef`, respectively. The action for the first rule is to forward to the router's physical port having port number 1. The action for the second rule is to forward the packet out of no ports, i.e. to drop the packet. The action of the third rule directs the switch to forward the packets out of *both* ports 2 and 3. The action for the fourth rule first sets the MAC source address of the packet to `0x000000000000` and then forwards the packet to port number 1.

As illustrated in the preceding example, OpenFlow flow table rules can match on packet header fields at various layers. OpenFlow protocol version 1.0 allows match conditions to match on 12 packet header fields, including fields from layers 2, 3, and 4. A rule that includes values for all 12 fields is called an *exact* rule, while a rule that places no condition on one or more fields is called a *wildcard* rule. OpenFlow rules also allows a wide variety of actions, including forwarding to one or more ports, modifying packet header fields, and placing a packet in a previously configured queue. Using these capabilities, a network controller can program L2 or L3 destination-based

1. This applies to OpenFlow version 1.0. Later versions of OpenFlow support multiple flow tables organized in a pipeline.

2. In addition, the switch may choose to buffer the packet and send the controller an identifier for the packet and the packet header, rather than the full packet.

Message Type	Sender	Description
Request features	c	Controller requests switch to describe its configuration (i.e. port information)
Request stats	c	Controller requests port or flow statistics, such as byte and packet counters.
Send packet	c	Send a packet, specified either as a sequence of bytes or by reference to a packet buffered at the switch.
Flow modification	c	Insert, update, or delete flow table entries
Features reply	s	Description of the features (i.e. ports) of the switch.
Stats reply	s	Report of port or flow statistics of the switch.
Packet In	s	Notification that the given packet arrived at the switch either because it failed to match in flow table or it matched a rule whose action directed the packet to the controller.
Flow removed	s	A flow entry was removed due to timer expiration or flow entry delete command.

Figure 2.6: Summary of key OpenFlow messages. The sender column indicates whether the controller (c) or the switch (s) sends the given message.

routing, or can implement customized classifications to perform policy-based routing and QoS forwarding. Researchers have used OpenFlow to implement a variety of interesting network control features, including load balancing routing algorithms in data centers [4], server load balancing [71], campus network access control policies [47], network support for live virtual machine migration, and network debuggers [24].

An OpenFlow switch also maintains various traffic statistics, including per-port byte and packet counters and per-flow table entry byte and packet counters. In addition, each flow table entry can be configured with both hard and soft timeouts. If a rule has a hard timeout value t, then the switch must remove the rule from the flow table after t seconds have elapsed since the rule's installation. If a rule has a soft timeout value t, then the switch must remove the rule after the first period of t seconds during which no packets match the rule.

Figure 2.6 summarizes the key OpenFlow messages. In particular, the controller may request information from a switch, may command the switch to send a packet or may command the switch to modify its flow table. The switch can respond to requests for information and can indicate when a flow entry is removed.

When a packet does not match any rules in the flow table, the packet is sent to the network controller for the switch. This notification is called a packet-in message and indicates to the controller that further rules are needed in the switch to handle a particular type of packet. This situation arises frequently since flow tables often do not include rules to cover all possible packets. For instance, in the example flow table of Figure 2.5, a packet with TCP destination port 23 and Ethernet destina-

String	Value
01*11*	x
01*10*	y
1*	z

Figure 2.7: Example TCAM.

tion `0x121212121212` would not match any rules. In this situation, an OpenFlow forwarding element will notify the controller of the event. The notification, called a *packet-in event*, includes a prefix of the arriving packet contents (enough to include the common layer 2,3, and 4 packet headers) along with some extra information, such as the physical port on which the packet arrived. Having received this notification, the controller may decide to add a new rule to the flow table to handle this type of packet. This packet-in feature allows a controller to detect actual network traffic, allowing it to remove rules that are not being used and install rules that are actively needed.

2.1.4 OpenFlow Switch Implementation

The flexibility of the OpenFlow flow table forwarding abstraction presents an implementation challenge for forwarding devices. Well-known algorithms for L2 forwarding based on source and destination MAC addresses or L3 forwarding based on longest prefix matching on IP destination addresses are not general enough to apply to the flow table model. A naive linear search of the flow table from highest priority to lowest priority would be too slow to forward packets at 10 Gbps or higher, as required in many networks today.

As a result, most OpenFlow switches use a specialized form of memory, a Ternary Content Addressable Memory (TCAM), to perform packet classification in a single memory access cycle. A TCAM is a memory chip where each entry includes a ternary string of some fixed length n and some associated information. A ternary string of length n is a string of n symbols where each symbol is either 0, 1, or *. Figure 2.7 shows an example TCAM memory with ternary strings of length 6. Most importantly, a TCAM supports fast lookup of values associated to binary strings. Suppose v is a binary vector of length n. Then v matches a ternary string w if, for each $i \in \{0 \ldots n-1\}$, $v_i = w_i$ or if $w_i = *$. Given v, the TCAM compares v against each TCAM entry and outputs the value associated with the entry in the highest (or lowest, depending on the implementation) position whose ternary string matches v, or outputs a signal indicating that no entry matches v. For example, the string "010110" matches the first entry in the table of Figure 2.7, while string "011010" matches the final entry. A TCAM can perform this search in parallel and can output the matching value in a single memory cycle, typically 10 nanoseconds. Hence, TCAMs can support line rate packet classification and forwarding at rates of 10 Gbps.

While TCAMs are very powerful, they require more transistors per stored bit than SRAM and are more power hungry than other memories. As a result, TCAM memory is expensive and most switches today have limited amount of space in their TCAM

memories. Most devices available today have TCAMs with capacities supporting hundreds to a few thousand OpenFlow rules. High-end devices are now becoming available that have capacity for up to 1 million flow table entries, but these are expensive and not commonly deployed in today's networks.

As a result of limited TCAM capacity, it is important for network controllers to use flow table capacity efficiently. If the network controller requires more rules for its forwarding policy than the available capacity, it will need to handle packets for some flows at the controller. While algorithms to swap different rules into the flow table based on demand may allow more packets to be handled in the switch under these conditions, it is clear that requiring more rules than there is capacity will lead to some degree of network performance degradation.

2.1.5 ForCES

The Forwarding and Control Element Separation (ForCES) IETF working group defined the ForCES architecture, model, protocol and various extensions. The ForCES work, like OpenFlow, attempts to separate the control and forwarding data planes of various network devices, such as IP routers, switches and firewalls. The ForCES work (which predates OpenFlow by several years) therefore shares many similarities with OpenFlow and could also be used as the controller-switch component of an implementation of the SDN architecture. Although it is more general than OpenFlow, ForCES appears to be less widely deployed than OpenFlow, which is now available on a number of network devices. In this dissertation, we focus entirely on programming flow tables, which can also be implemented on ForCES. Several documents provide detailed comparisons of OpenFlow and ForCES [60, 72].

2.2 Network Control System

Network control systems for OpenFlow networks typically consist of several distinct software layers, as illustrated in Figure 2.8. The lowest component in the stack, which we call the *OpenFlow messaging service* manages connections with OpenFlow switches and implements the basics of the OpenFlow protocol. The next layer in the stack consists of user-defined event-handler logic that receives and sends OpenFlow messages to devices via the OpenFlow messaging service. By allowing multiple switches to connect to the messaging service, the user-defined logic can control an entire collection of switches using a single, centralized program.

The user-defined logic typically provides a so-called *North-bound API*, which allows network administrators to configure parameters of the network at runtime. For example, a network control system may allow an administrator to enter and edit a list of endhosts which are allowed to access the network. Typically north-bound APIs are exposed through HTTP server following REST [19] principles. This allows programmatic access to the administrative parameters of a network, which can be used to implement graphical user interfaces (GUIs), command-line interfaces (CLIs), or administrative scripts.

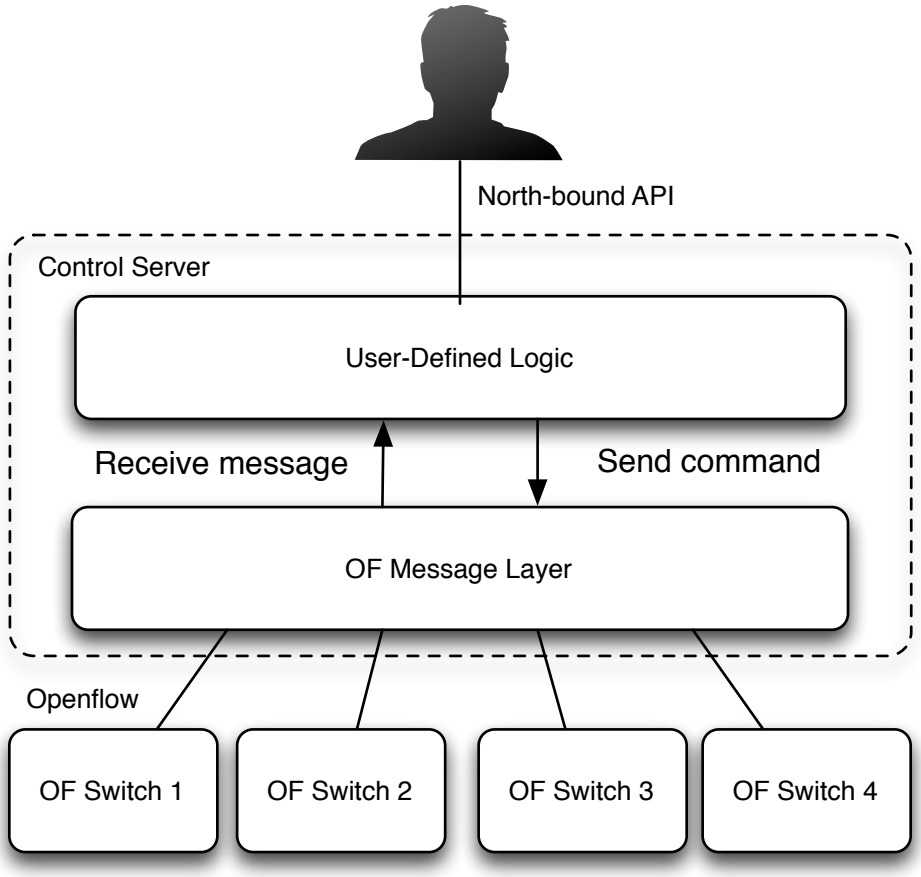


Figure 2.8: Software architecture for a typical OpenFlow network control system.

2.2.1 OpenFlow Messaging Service

All OpenFlow network controllers include a software layer, which we call the *OpenFlow messaging service* that implements the basic aspects of the OpenFlow network protocol. In particular, the message service implements parsers and serializers for converting between binary representations of OpenFlow messages and logical representations of those messages in the host programming language of the message service. For example, Floodlight [20] is an OpenFlow messaging service, implemented in Java, which converts OpenFlow messages to and from Java objects that logically represent the information contained in OpenFlow messages. In addition, the messaging service typically implements other low-level features, such as version negotiation, maintaining working connections with the OpenFlow switches by sending and responding to keep-alive messages of the OpenFlow protocol, and auto-generating message transaction numbers.

Perhaps most importantly, the message layers provide a mechanism by which a user can execute user-defined program logic whenever an OpenFlow message is received and methods for sending OpenFlow messages to switches, in response to either OpenFlow or external events.

Many OpenFlow message services exist, primarily distinguished by the language in which they are implemented and the language in which user-defined logic can be written (typically the same as the implementation language). In particular, NOX [23] is implemented in C++ and offers C++ and Python APIs for message handling; Beacon [8], Floodlight [20], and OpenDaylight [50] are implemented in Java and offer a similar API for Java. Ryu [58] and POX [55] are implemented in Python and allow user-defined logic expressed in Python.

2.2.2 User-Defined Logic

We now review several existing approaches for organizing the user-defined logic on top of the OpenFlow messaging layer.

Reactive Controllers

Many controllers install rules only in reaction to packet-in messages from switches. Such controllers are named *reactive* controllers. In particular, the Ethane [12] system is a prominent early SDN system that used the reactive approach. In Ethane, the packet header of the first packet of every new application flow was diverted to the network controller, which checks a security policy against the packet. If the first packet header passes the security policy, all packets of the same application flow, and hence having the same packet headers, are allowed and flow table entries to route the application flow are installed. Otherwise, flow table entries to drop packets in the application flow are installed.

Reactive controllers often treat the OpenFlow switches under their control as a cache for the packet-in event handler maintained at the controller. These controllers often use only exact match flow entries and install rules with a timeout to ensure that

cached entries (i.e. flow table rules) are removed periodically. Rule timeouts ensure that rules that are no longer consistent with the control plane state, which may change in response to various network and administrative events, are eventually removed and new rules are eventually installed. Such controllers typically use flow table space poorly, leading to large flow tables and a large number of packet-in events and poor performance. Furthermore, the use of timeouts to remove invalid entries limits the response time of the network to events such as link failures and host mobility. These controllers are therefore not applicable in networks with stringent requirements on link failure recovery time.

Component-based Message Service

Some OpenFlow messaging services, such as NOX and Floodlight, provide mechanisms to allow for multiple user-defined components to register event handlers to execute when OpenFlow messages are received. These systems typically allow the programmer to specify the relative order in which the registered components are invoked after receiving an event. In addition, each component handler function returns a code that indicates whether components later in the order should be executed. Unfortunately, these mechanisms do not provide a useful abstraction, since the programmer assembling the components must understand each component's features in order to decide on the order of component handlers and the appropriate return action of each handler. The components can easily interfere with each other by performing state-changing actions on switches, such as flow modifications, which interfere with another component's operation. For example, a component which learns about the topology by injecting and observing LLDP packets will not operate properly if some other component adds flow table rules that drop LLDP packets.

SNAC & FML

SNAC [61] and Flow-based Management Language (FML) [25] are declarative policy languages for configuring and managing enterprise networks, implemented using NOX. An FML program is a collection of non-recursive logic rules that define when certain facts hold of a flow, and ultimately define which forwarding actions should be applied to packets of a given flow. FML policies are more concise than OpenFlow controllers written in many other frameworks. On the other hand, FML is more restrictive than OpenFlow message services. In particular, FML policies can not express policies that involve change of controller state.

The implementation of FML includes interesting algorithms that allow high performance policy lookup. In particular, FML builds a lookup tree from the input policy which allows the collection of rules that apply to a given packet to be located quickly. However, FML's implementation only generates exact match flow table entries, leading to poor use of flow table space, a high rate of packet-in events, and poor network performance.

Frenetic

Frenetic [21] provides an OpenFlow programming framework, based on the principles of functional reactive programming. The key innovation of Frenetic is that it allows users to express policies that observe network packets and policies that forward network packets independently. The Frenetic runtime system then compiles flow table entries that ensure that both the queries over network packets and the forwarding policies are implemented correctly.

Frenetic requires users to express forwarding policies using a language of match conditions and actions, much like OpenFlow. In addition, observations of network packets must be expressed in terms of query operators that combine match conditions with stream processing operators and functional reactive programming combinators. Frenetic therefore requires users to program in terms of a specialized, declarative query language and match-action flow patterns, which can be a burden on SDN programmers.

In addition, Frenetic's runtime system generates only exact match rules, leading to poor use of flow table capacity.

NetCore

The NetCore [44] language supports a declarative language for expressing the forwarding policy of a single switch. In particular, the language provides logical operators, such as conjunction, disjunction and negation, allowing policies to be composed via Boolean connectives. Like Frenetic, NetCore requires that users express their policy in a specialized, declarative domain specific language of flow patterns. Unlike Frenetic, NetCore's compilation algorithm effectively uses wildcard rules and can generate compact flow tables.

NetCore supports dynamic policies in which forwarding behavior depends on packet history. However, the NetCore compiler only generates flow rules when it can be shown that the flow rule will never need to be revised in the future. It therefore cannot install flow table rules in networks with host mobility or topology dynamics, for example. Furthermore, NetCore requires that the user write an "invariant predicate" for each dynamic policy indicating when the policy becomes invariant. Incorrectly specifying the invariant prediate may lead to controller errors.

Pyretic

Pyretic [45] is an extension of NetCore which adds a richer set of packet actions and adds a sequential composition operator on policies. The addition of sequential composition of policies allows Pyretic to extend NetCore from specifying the behavior of a single switch to specifying the behavior of an entire network of OpenFlow switches.

Pyretic, like NetCore, uses queries over packets to receive packets at the controller, but uses a different approach than NetCore for the specification of dynamic policies. In particular, Pyretic uses a single policy variable to hold the current policy; policy can be changed by updating this policy variable. Typical programs therefore register

queries over packets to execute user-defined callback functions, which in turn update the policy variable with a newly calculated network policy.

The Pyretic runtime system offers both reactive and proactive compilers. The proactive compiler compiles a Pyretic policy to rules for all switches in the network. This whole network policy compiler is invoked whenever the policy variable is updated, which typically occurs whenever a host location is updated or a topology event occurs. Due to the high computational overhead of compiling the entire network policy, Pyretic copes poorly with network and policy dynamics.

2.2.3 North-bound APIs

While most north-bound APIs are implemented using an HTTP server following REST principles, the precise resources and operations supported are specific to each network control system. However, certain prominent applications have defined important north-bound APIs. In particular, OpenStack [51], an open source data center management system specifies a north-bound API to a network controller that implements the network virtualization services required by the data center's configuration. This north-bound API, called Neutron [49], allows OpenStack software to specify tenants, networks, IP subnets, and virtual ports.

2.2.4 Scaling Network Controllers

The introduction of a centralized network control plane may introduce scaling bottleneck when many events must be processed by the centralized control plane. Some systems attempt to improve scaling by reducing the number of events that must be processed by the controller. In particular, Devoflow [15] increases scalability by refactoring the OpenFlow API, reducing the coupling between centralized control and centralized visibility.

NOX-MT [66], Beacon [8], and Maestro [11] scale network controllers using multicore servers. In particular, NOX-MT modifies NOX to better utilize multicore CPUs by batching system calls and by using Boost [10] C++ libraries for IO notification and threading. Beacon [18] uses one OS thread per core and statically assigns each switch to be handled by a particular core when it connects to the controller. Tootoonchian [66] demonstrates that NOX-MT, Beacon, and Maestro process 2 million, 100 thousand, and 300 thousand OpenFlow events respectively per second when controlling 64 emulated switches using up to 8 OS threads. Later measurements by Voellmy [69] demonstrated that NOX-MT scales to 5 million events per second using 10 CPU cores and Beacon scales to 13 million flows per second at 20 CPU cores.

Other systems provide multi-server implementations of OpenFlow network controllers, in order to load balance event load across multiple control servers. Onix [29] partitions network state across multiple distributed controllers, alleviating scalability and fault-tolerance concerns, but compromising the attractive simplicity of the centralized model. ElastiCon [16] implements a distributed controller that supports migration of switch-specific controllers to different servers in response to controller event load.

2.3 Summary

In this chapter, we have introduced the SDN architecture, which proposes a separation of the forwarding and control planes of a network. We have introduced the OpenFlow protocol as an emerging standard SDN protocol for implementing controller-switch interaction. We then presented the typical organization of OpenFlow control planes and current approaches to programming network controllers. We described the two approaches for expressing network policy: (1) specify imperative event handlers on top of an OpenFlow messaging service, and (2) express policy using declarative domain-specific languages for describing flow patterns and queries over packet arrivals.

In this dissertation, we develop a programming model for SDN control, called algorithmic policies, that allows programmers to express network policy by writing an imperative packet processing computation in a general-purpose language that will be invoked on *every* packet passing through the network. This alleviates the need for declaring flow patterns and queries to program network policy. In addition, we develop novel techniques that automatically implement arbitrary packet processing functions efficiently on OpenFlow switches. In the next chapter, we introduce algorithmic policies and present an overview of our solution.

Chapter 3

Algorithmic Policies

Software-Defined Networking offers the appeal of a simple, centralized programming model for managing complex networks. However, challenges in managing low-level details, such as setting up and maintaining correct and efficient forwarding tables on distributed switches, often compromise this conceptual simplicity.

This chapter motivates the Maple algorithmic policy programming model by examining the challenges that arise in programming simple network control policies and demonstrating how the algorithmic policy programming model eliminates those difficulties from the SDN programmer. We then give an overview of the novel techniques that Maple uses to efficiently and scalably implement algorithmic policies.

3.1 Challenge: Generating Correct, Efficient Flow Tables

To motivate algorithmic policies, we consider a simple network policy on a network consisting of a single switch with four ports and three hosts at known, fixed locations (i.e. ports), as shown in Figure 3.1. The example policy consists of two parts. First, an access control policy: TCP packets with destination port 22 should be dropped. Second, a forwarding policy: other packets should be sent to the location of the host identified by the destination mac address.

We can express this policy with the ordinary, Python-like program shown in Listing 3.1, which defines a table mapping mac addresses to ports and defines a forwarding function that describes how each packet should be forwarded. We consider this program to be an *algorithmic policy*. It consists of a program that defines how an incoming packet should be forwarded in the network and corresponds to how a programmer might write the packet forwarding program as an ordinary program running on the switch, handling each incoming packet. The return value of the function is simply a list of ports that the packet should be forwarded out of. If the list is empty, the packet is forwarded out of no ports (i.e. it is dropped). In general, other packet processing functions, such as packet modifications, can be performed. However, in this chapter, we consider only forwarding actions that forward to a subset of ports on a switch, and we therefore simplify our actions to consist of just lists of ports.

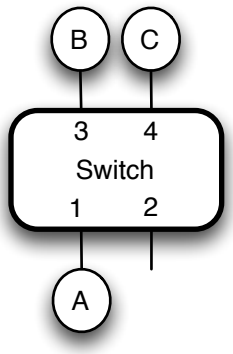


Figure 3.1: Simple single switch network.

Listing 3.1: Simple algorithmic policy with fixed set of hosts and host locations.

```
1 portForMac[A] = 1
2 portForMac[B] = 3
3 portForMac[C] = 4
4
5 def forward(pkt):
6   if 22 == pkt.tcp_dst_port:
7     return []
8   else:
9     return [portForMac[pkt.eth_dst]]
```

Listing 3.2: Naive reactive controller.

```
portForMac[A] = 1
portForMac[B] = 3
portForMac[C] = 4

def onPacketIn(pkt):
  if 22 == pkt.tcp_dst:
    installRule({'priority':1, 'match':{'tcp_dst':pkt.tcp_dst}, 'action':[]})
  else:
    action = [portForMac[pkt.eth_dst]
    rule = {'priority':0, 'match':{'eth_dst':pkt.eth_dst}, 'action':action}
    installRule(rule)
    send(pkt, action)
```

Of course, OpenFlow switches cannot execute the above algorithmic policy. Instead, a programmer must specify the policy as a collection of OpenFlow rules. In this example, we can use the following flow table to implement the algorithmic policy:

Priority	Match	Action
1	`tcp_dst:22`	`forward []`
0	`eth_dst:A`	`forward [1]`
0	`eth_dst:B`	`forward [3]`
0	`eth_dst:C`	`forward [4]`

Note that the rule at priority level 1 serves two purposes. First, it executes the policy to drop packets to port 22. Second, it prevents packets with port 22 from matching any of the rules at priority 0, since it is matched first. This second function is critical, since the rules at priority level 0 do not specify that they should apply to *non port 22* traffic. In general, OpenFlow does not support negated conditions, and we therefore *must* encode negation by way of prioritization of rules.

Although this case was simple, in general, it can be challenging to produce a flow table corresponding to a given policy, and it would therefore be useful to have a method of generating such tables automatically from the packet forwarding function. In fact, there is a common style of programming an OpenFlow controller which attempts to accomplish exactly this. In a so-called "reactive" controller, the programmer writes a function to handle `packet-in` messages sent to the controller. This packet-in handler treats the switch as a cache and the packet-in message as a request to fill the cache to handle a particular type of traffic. The packet-in handler accomplishes this task by both forwarding the packet and adding rules to handle similar packets in the future. In addition, the controller assumes (or enforces) that the switch begins with an empty flow table. Listing 3.2 shows a naive translation of our algorithmic policy into a reactive-style controller. In this code, the controller adds a rule to the switch using the `installRule()` library function and commands the switch to forward the packet using the `send()` library function.

Priority	Match	Action
1	`tcp_dst:22`	`forward []`
0	`eth_dst:B`	`forward [3]`

Figure 3.2: Forwarding table after invoking controller of Figure 3.3 with packet `p1`.

Although this program is similar to our original algorithmic policy in Figure 3.1, it does require that the programmer analyze the program to generate appropriate rules. In particular, the programmer must determine both the match condition (i.e. which fields to specify and which to leave unspecified) and the priority of each rule. A mistake in either could lead to packets forwarded incorrectly. Moreover, the assignment of priorities is not modular; it requires global analysis of the forwarding policy. To see this, suppose the `else` branch of the program was modified and subsequently required more priority levels to implement. In this case, the rules for the `if` branch would need to use a priority level higher than 1 and would need to be revised.

Even worse, this program has a bug! To see the bug, suppose the first packet `p1` to arrive at the switch is a TCP packet with `eth_src:A`, `eth_dst:B` and `tcp_dst:80`. Since the switch's flow table is empty, the packet is diverted to the controller. The controller executes the `onPacketIn` function on `p1`, forwards it and installs a rule for normal traffic. The switch flow table after handling this packet is as follows:

Priority	Match	Action
0	`eth_dst:B`	`forward [3]`

Suppose now that a subsequent packet with `tcp_dst:22` and `eth_dst:B` arrives at the switch. Although we intend for this packet to be dropped, it in fact matches the single rule currently in the switch and is therefore incorrectly forwarded to port 3. Although we intended the installed rule to match only packets not having `tcp_dst` equal to 22, our naive reactive controller neglected to install the higher priority rules enforcing this condition. Hence, we see that a naive approach to translating algorithmic policies to reactive controllers will not suffice.

3.1.1 Fix 1: Handle Overlapping Rule Dependencies

One way to prevent the above problem from occurring is to ensure that if any rules are installed for forwarding normal traffic, then the higher priority rule for access control is installed as well. The program in Listing 3.3 implements this strategy. This program introduces a boolean flag, `aclRulesInstalled` to indicate whether the program has installed the access control rule yet. Then, on each packet, the program installs the access control rule and sets the flag to `True`, if it has not yet been installed. Figure 3.2 shows the switch flow table after packet `p1` arrives at the switch.

Producing this program imposes a burden on the programmer. The program has become substantially more complex with both proactive and reactive rule installation. Furthermore, it suggests that we can only apply the reactive approach to the lowest

Listing 3.3: First bug fix to the naive controller in Listing 3.2.

```
portForMac[A] = 1
portForMac[B] = 3
portForMac[C] = 4

aclRulesInstalled = False

def installACL():
  if not aclRulesInstalled:
    installRule({'priority':1, 'match':{'tcp_dst':22}, 'action':[]})
    aclRulesInstalled = True

def onPacketIn(pkt):
  installACL()
  if 22 == pkt.tcp_dst:
    return
  else:
    action = [portForMac[pkt.eth_dst]]
    installRule({'priority':0, 'match':{eth_dst:pkt.eth_dst}, 'action':action)
    send(pkt, action)
```

priority rules. This is unfortunate since inserting flow rules on demand has benefits in many cases when the number of rules required for a policy would not all fit into the flow table simultaneously.

3.1.2 Fix 2: Avoid Overlapping Rules

A second way to fix the bug of the first reactive controller (Figure 3.2) is to avoid the use of overlapping rules at multiple priority levels. More concretely, our reactive controller could match *all* fields of the packet, ensuring that no other rules could apply to the packet (since any other rule must differ in at least one field). This flow table management strategy is in fact generic, since we always perform the same actions: calculate a match condition that matches all possible attributes of the packet header and use priority 0. We can therefore write this controller as follows:

```
portForMac[A] = 1
portForMac[B] = 3
portForMac[C] = 4

def onPacketIn(pkt):
  mch = exactMatchFromPacket(pkt)
  action = forward(pkt)
  installRule({'priority':0, 'match':mch, 'action':action})
  send(pkt, action)
```

In this example, we used a helper function `exactMatchFromPacket()` which computes a match condition that matches every field of the packet. After receiving a few packets, the switch flow table might look as follows:

Prio	Match	Action
0	`eth_src:A,eth_dst:C,eth_type:0x0800,...,tcp_dst:22`	`forward []`
0	`eth_src:A,eth_dst:B,eth_type:0x0800,...,tcp_dst:22`	`forward []`
0	`eth_src:B,eth_dst:A,eth_type:0x0800,...,tcp_dst:80`	`forward [1]`
0	`eth_src:B,eth_dst:A,eth_type:0x0800,...,tcp_dst:81`	`forward [1]`
0	`eth_src:B,eth_dst:A,eth_type:0x0800,...,tcp_dst:82`	`forward [1]`

Note that all rules can use the same priority level (0), since they are all non-overlapping. In addition, note that there are now multiple rules for a given destination pair (e.g. packets to `A`) while our original flow table used just one rule per destination.

This program does not suffer from the bugs of the first reactive controller, since rules generated for the `else` branch never overlap rules generated for the `if` branch. However, the program may perform poorly, because it uses only exact match rules, which match at a very granular level. Every new flow will fail to match at switch flow tables, leading to a roundtrip to the controller. To make matters worse, it may not be possible to completely cover active flows with such granular rules, since rule space in switches is typically limited. As a result, rules may need to be frequently evicted to make room for new rules. To handle this, typical reactive controllers using exact match rules add each rule with a short timeout (typically a few seconds) to ensure that rules are frequently removed. This can negatively impact performance by causing packets to be diverted to the controller with higher frequency and by imposing a higher load on the switch's control processor to handle the frequent forwarding state changes.

3.1.3 Summary

Comparing the example programs using the current models with the algorithmic policy in Figure 3.1 at the beginning of this section, we see the unnecessary burden that current models place on programmers, forcing them to consider issues such as match granularity and rule dependencies.

3.2 Challenge: Handling Network Dynamics

In the previous section we assumed that the hosts in the network and their locations are fixed and known at compile time. Both of these assumptions are unrealistic, as new devices are brought on to enterprise networks on a daily basis and devices may be moved from one port to another port or even to another switch. To handle this situation, we must write a policy that handles network dynamics.

Consider the example network in Figure 3.1 again. To avoid hard-coding the device identities and locations of devices, we instead *learn* the identities and locations

Listing 3.4: Learning switch expressed as an algorithmic policy.

```
1 def forward(pkt):
2   portForMac[pkt.eth_src] = pkt.in_port
3   if 22 == pkt.tcp_dst:
4     return []
5   elif pkt.eth_dst in portForMac:
6     return [portForMac[pkt.eth_dst]]
7   else:
8     return allPorts.remove(pkt.in_port)
```

of devices from the packets they send into the switch. For example, we can infer that host `A` is on port 1 when `A` sends its first packet on port 1. Listing 3.4 demonstrates how we can easily extend our original algorithmic policy to accomplish this. In particular, since we program it from the perspective of a function seeing every packet, we simply update the `portForMac` table on every packet.

In addition to the update of the `portForMac` table in the first line of the function, we now handle the case when the location of the destination of a given packet is not known. In this case, we broadcast to all ports except the incoming port. This case did not arise in the previous controllers, since the location of each host was known at all times.

This program is simple, intuitive, and effective: it learns a host's location when the host sends its very first packet. Unfortunately, implementing this in mainstream SDN programming systems introduces many complexities, which we now analyze more closely.

3.2.1 Lost State Updates

Suppose we simply amend the program in Figure 3.3 to learn host locations and to handle the case when packets are broadcast out of all ports, to obtain the controller shown in Listing 3.5.

Unfortunately, this program also has a bug. After receiving packet `p1`, the `portForMac` table will include a single entry with key `A` and value `1` and the switch's flow table will be as shown in Figure 3.3. Suppose now that a packet `p2` with `eth_src:C` and `eth_dst:B` arrives in the network. Given the current flow table, `p2` will be forwarded by the previously installed rule for `eth_dst:B`. In particular, the packet is not forwarded to the controller and the `portForMac` table will therefore not be updated with the location of `C`, even though `C` has sent at least one packet. Similarly, if `B` now sends packets to `tcp_dst:22`, the packet is dropped by the switch and the controller fails to learn the location of host `B`. As a result, forwarding will continue to flood packets to `B`, violating the desired policy.

These problem arise because our algorithmic policy now changes state when seeing the first packet from host `C`. Therefore, the generated flow table must ensure that the controller receives at least the first packet from host `C`. If it fails to do this, the

Listing 3.5: Controller of Listing 3.3 extended with host location learning.

```
1  aclRulesInstalled = False
2
3  def installACL():
4    if not aclRulesInstalled:
5      installRule({'priority':1, 'match':{'tcp_dst':22}, 'action':[]})
6      aclRulesInstalled = True
7
8  def onPacketIn(pkt):
9    portForMac[pkt.eth_src] = pkt.in_port
10   installACL()
11   if 22 == pkt.tcp_dst:
12     return
13   else:
14     if pkt.eth_dst in portForMac[pkt.eth_dst]:
15       action = [portForMac[pkt.eth_dst]]
16     else:
17       action = allPorts.remove(pkt.in_port)
18     installRule({'priority':0, 'match':{'eth_dst':pkt.eth_dst}, 'action':action})
19     send(pkt, action)
```

Priority	Match	Action
1	`tcp_dst:22`	`forward []`
0	`eth_dst:B`	`forward [2,3,4]`

Figure 3.3: Forwarding table after invoking controller of Figure 3.5 with packet `p1`.

Listing 3.6: Fix to the controller in Listing 3.5 to avoid lost host location table updates.

```
aclRulesInstalled = []

def installACL(eth_src):
  if eth_src not in aclRulesInstalled:
    installRule({'priority':1, 'match':{'eth_src':eth_src,'tcp_dst':22}, 'action':[]})
    aclRulesInstalled.add(eth_src)

def onPacketIn(pkt):
  installACL(pkt.eth_src)
  if 22 == pkt.tcp_dst:
    return
  else:
    if pkt.eth_dst in portForMac:
     action = [portForMac[pkt.eth_dst]]
    else:
     action = allPorts.remove(pkt.in_port)
    rule = {'priority':0, 'match':{'eth_src':pkt.eth_src,'eth_dst':pkt.eth_dst}, 'action':action}
    installRule(rule)
    send(pkt, action)
```

system may become stuck in an incorrect state indefinitely.

One way to resolve the specific problems just discussed is to install rules matching on `eth_src` address and to only install rules for a sender after at least one packet has been observed from the sender. With this change, we only install rules that handle packet for senders whose locations have already been discovered and whose packets would lead to no further state change in the algorithmic policy. Hence, these rules are safe to install. After making this change, we obtain the code shown in Listing 3.6.

Notice that we also extended the match condition used in the access control rules to include a match on the `eth_src` field, in order to avoid having access control rules mask packets that would lead to the controller learning about a change in the `portForMac` table. We therefore generalized the `aclRulesInstalled` flag to a set of hosts and add access control rules for a given sender whenever normal rules are needed for the sender.

Figure 3.4 shows the resulting forwarding table with our modified controller after receiving packet `p1`. We see that all rules apply only to packets sent by `A` and hence the controller will correctly receive the first packet from either `B` or `C`.

3.2.2 Inconsistent Forwarding Rules

Suppose we also attempt to support host mobility. For example, consider the transition in Figure 3.5 where host `A` moves from port 1 to port 2. First, we observe that our previous controller does not correctly handle this state change. In particular, if

Priority	Match	Action
1	`eth_src:A,tcp_dst:22`	`forward []`
0	`eth_src:A,eth_dst:B`	`forward [2,3,4]`

Figure 3.4: Forwarding table after invoking controller of Figure 3.6 with packet `p1`.

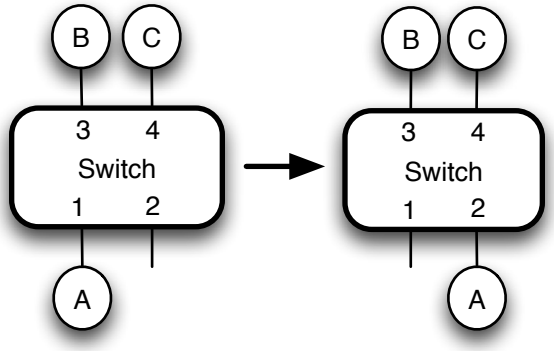


Figure 3.5: Example of host mobility in the simple topology.

the switch's flow table is in the state shown in Figure 3.4 before `A` moves to port 2, then the switch will continue to send packets from `A` to ports `[2,3,4]` rather than to the controller and hence the controller will not learn of the updated location of `A`. Therefore, packets from any other host to `A` will be misrouted to port 1 indefinitely, leading to `A` essentially losing connectivity.

To ensure that the controller receives at least the first packet from a host when it moves to a new port, we also specialize each rule to match on the incoming port of the packet. After doing this, the flow table produced after receiving `p1` will be:

Priority	Match	Action
1	`in_port:1,eth_src:A,tcp_dst:22`	`forward []`
0	`in_port:1,eth_src:A,eth_dst:B`	`forward [2,3,4]`

Now, when host `A` moves to port 2, its next packet, will be sent to the controller, since no rule applies to packets with `in_port:2` and hence the controller will correctly learn the new location of `A`.

Unfortunately, further problems exists. To see these, suppose `B` sent a packet to `A` prior to `A` 's move to `B`. Then the flow table prior to the move would be:

Priority	Match	Action
1	`in_port:1,eth_src:A,tcp_dst:22`	`forward []`
1	`in_port:3,eth_src:B,tcp_dst:22`	`forward []`
0	`in_port:1,eth_src:A,eth_dst:B`	`forward [2,3,4]`
0	`in_port:3,eth_src:B,eth_dst:A`	`forward [1]`

Now suppose that after the move to port 2, `A` sends packet `p3` to `B`, resulting in the following flow table:

Priority	Match	Action	
1	`in_port:1,eth_src:A,tcp_dst:22`	`forward []`	wasted
1	`in_port:2,eth_src:A,tcp_dst:22`	`forward []`	
1	`in_port:3,eth_src:B,tcp_dst:22`	`forward []`	
0	`in_port:1,eth_src:A,eth_dst:B`	`forward [2,3,4]`	wasted
0	`in_port:2,eth_src:A,eth_dst:B`	`forward [3]`	
0	`in_port:3,eth_src:B,eth_dst:A`	`forward [1]`	wrong

Specifically, the controller adds one rule for access control for host `A` on port 2 and a rule for other packets from `A` to `B` on port 2.

Unfortunately, the updated table is neither efficient nor correct. In particular, 2 rules no longer have any effect, since host A is no longer located at port 1. These wasted rules lead to inefficient usage of flow table memory, a resource which is often a critical bottleneck. More seriously, the rule for traffic from `B` to `A` misroutes traffic, leading to an indefinite period where C and A fail to communicate and thereby leading to poor network performance for some hosts.

Many real-world controllers attempt to solve these problems by using time-outs to remove flow table entries after some time. Unfortunately, this can lead to prolonged periods (typically several seconds) where incorrect policy is applied.

Other controllers use ad-hoc solutions to the problem. For example, one might consider a simple fix specific to host mobility: whenever a host A changes its location (*i.e.*, the incoming port is different from the previously recorded incoming port), delete all rules to or from host A.

The problem of this revision, however, is that it is neither efficient nor correct in a more realistic setting such as a multi-switch topology. First consider efficiency. An interpretation of the fix is that the controller applies this strategy at all switches, and hence any rule containing A at any switch is deleted. Consider a large network where a host B at a remote switch is communicating with A. Assume that A changes port but is still at the same switch. Then a large part of the path between A and B does not change, and hence deleting all rules at all of the intermediate switches is unnecessary, resulting in unnecessary disruption.

Next consider correctness. The reason that a switch has a rule containing A may not be related with routing using location table at all, but access control. For example, the controller may want to limit host A to a part of the network. Hence, it sets up flow rules containing A at the flow tables at a few switches to form a boundary for A. Hence, when A changes its location within its allowed part of the network, it should not result in the deletion of all policies containing A in the whole network, as this will break the boundary policy, for example when a remaining flow rule (*e.g.*, based on IP) may allow A to go through. In other words, updating switch flow tables needs precision to avoid unintended damages.

3.2.3 Summary

The examples of this section demonstrate that manual management of flow tables in the presence of network dynamics can lead to serious bugs, including lost state

Listing 3.7: Algorithmic policy with bad hosts policy variable.

```
1 monitorFlag = True
2 portForMac = {A:1,B:3,C:4}
3
4 def forward(pkt):
5  if (monitorFlag and pkt.eth_src==1):
6    return [portForMac[pkt.eth_dst], 4]
7  else:
8    return [portForMac[pkt.eth_dst]]
```

updates, incorrect forwarding, and inefficient flow table usage. Correctly handling network dynamics can substantially complicate controller logic, greatly obscuring the simple algorithmic policy which we are implementing.

3.3 Challenge: Handling Policy Dynamics

Network controllers should handle not only physical state changes but also configuration state changes. In particular, realistic network controllers maintain configuration state and forward packets according to the configuration state. As an example, consider a network control policy consists of both a forwarding policy and a monitoring policy: all packets are forwarded normally to their next hop on the path to the destination. In addition, packets from host 1 are sent to a special monitoring device at specific host C located at port 4. Furthermore, an administrator can modify a Boolean flag at runtime which controls where the monitoring should occur or not. If the flag is false, the described monitoring policy is not implemented.

Listing 3.7 shows the algorithmic policy that implements this logic. In particular, it declares a boolean flag `monitorFlag` and makes use of this in the forwarding function. The administrator will be permitted to alter the value of the `monitorFlag` at any time. Several possible mechanisms for implementing this are possible, such as a REST server which modifies this flag in response to HTTP requests.

To illustrate the issues that can be caused by configuration state changes, consider the example on the left hand side of Figure 3.5 again. Suppose that `monitorFlag=True` and that A sends a packet to B and B sends packets to C. Then the forwarding table may be:

Priority	Match	Action
0	`in_port:1,eth_src:A,eth_dst:B`	`forward [3,4]`
0	`in_port:3,eth_src:B,eth_dst:C`	`forward [4]`

Suppose that the administrator now sets `monitorFlag=False`. How should the above forwarding table be updated? One possibility is to remove all rules having the monitoring device, `C`, as destination. However, this would lead to the deletion of the second rule, which is incorrect, since it is handling a flow from `B` to `C`, unrelated to

monitoring. Another possibility is to delete all rules sending to the port at which the monitoring device is located, namely port 4. This would correctly delete the first rule, which is used for monitoring, but would also incorrectly delete the second rule. A final possibility is to delete all rules using more than one port. Unfortunately, this highly specific condition is unlikely to be correct when monitoring functionality is combined with other network policies.

3.3.1 Summary

Most network controllers provide a north-bound API (NBI) that allows operators to specify runtime parameters of the network. Moreover, real world north-bound APIs can be complex: Neutron [49], a north-bound API for a network controller implementing network virtualization for OpenStack cloud deployments consists of 3 major entity types, with over 29 complex attributes, with each entity referring to one or more entities of other types. In addition, the mapping from policy state to forwarding rule state can be complex, resulting from the combination of several policies. Therefore, handling changes in policy state can be challenging, and improper handling can lead to efficiency and/or correctness issues.

3.4 Algorithmic Policies in Maple

The core objective of Maple is to offer an SDN programmer the abstraction that a general-purpose program f, which specifies the routing of each packet, defined by the programmer runs "from scratch" on a centralized controller for every packet entering the network, hence removing low-level details, such as distributed switch flow tables, from the programmer's conceptual model. We call this function an algorithmic policy. We now make this notion more precise and establish some notation.

A Maple program consists of three components: (1) a state space S, (2) a forwarding function (packet processing function) f, and an event handler (3) g:

State Space, S: The state space S of a Maple control program consists of a number of instances of forwarding data structures. These state components include both system-maintained data structures, such as the set of operational links in the network, but also user-defined and maintained data structures, such as the `portForMac` and `badHost` tables used in the examples of this chapter. More precisely, we can model the state space as a collection of forwarding data structure instances $D_1, D_2, \ldots, D_n$ which includes both built-in and user-defined components. In particular, the forwarding data structures include basic mutable variables, sets, and maps (e.g. associative arrays). Listing 3.8 lists the forwarding data structures and their operations, using Java-like type declarations to indicate the intended polymorphic type of each operation.

Forwarding function, `f`: The forwarding function $f : (Packet, S) \rightarrow (Route, S)$, defines how any packet is routed in the network in any state and how the processing of a packet alters the state of the system. We write $f^s(p)$ to denote the application of

Listing 3.8: Forwarding data structures and their operations.

```
MapleVariable<K>
MapleVariable<K> mapleVariable(String name, K initialValue);
K read(MapleVariable<K>);
void write(MapleVariable<K>, K);

MapleSet<K>
MapleSet mapleSet(String name);
boolean contains(MapleSet<K>, K);
void insert(MapleSet<K>, K);
void delete(MapleSet<K>, K);

MapleMap<K,V>
MapleMap mapleMap(String name);
V lookup(MapleMap<K,V>, K);
void insert(MapleMap<K,V>, K, V);
void delete(MapleMap<K,V>, K);
```

function f to packet p in state s and write f^s to denote the packet function in state s.

In practice, the programmer defines an imperative procedure f that accepts a packet as an argument and accesses S through references to state components (e.g. `badHosts`) rather than through arguments. Likewise the final state of the computation is not part of the return value, but is rather indicated implicitly through the operations performed on state components during the computation.

The return value of a policy `f` is (1) a forwarding path, which specifies whether the packet should be forwarded at all and if so how and (2) an updated state. For multicast, the forwarding path may be a tree instead of a linear path. The return of `f` specifies global forwarding behavior through the network, rather than hop-by-hop behavior. In addition, the returned route can indicate a transformation, such as rewriting source and destination MAC addresses, to apply to a packet before delivering the packet to the destination.

Except that it must conform to the signature, `f` may use arbitrary algorithms to classify the packets (*e.g.*, conditional and loop statements) and compute forwarding actions (*e.g.*, graph algorithms).

State transition function, `g`: The state transition function $g : (event, S) \rightarrow S$, defines how the state of the system changes as a result of various events, such as link up or down events, switch up or down events, etc. In practice the function g is (like f) specified by an imperative procedure which receives the event as an argument and which accesses the state components of S through references rather than through an procedure argument. In particular `g` may execute operations from the forwarding data structures to change the state of the system. The purpose of g is to allow controllers to maintain state components, such as next hop tables, in response to events.

Listing 3.9: Bad host example with forwarding function computing network routes.

```
badHostSet = mapleSet("BadHosts")

def forward(pkt):
 if pkt.eth_src in badHostSet or pkt.eth_dst in badHostSet:
   return nullRoute;
 else:
   return shortest(pkt)

def shortest(pkt):
 srcLoc = hostLoc[pkt.eth_src]
 dstLoc = hostLoc[pkt.eth_dst]
 switches = switches()
 links = links()
 route = myShortestPath(switches,links,srcLoc.switchID,dstLoc.switchID)
 return unicast(srcLoc,dstLoc,route)
```

3.4.1 Example

In this section, we show a complete, basic example algorithmic policy in Python to illustrate the components of the programming model. We consider a policy that forwards based on shortest path between source and destination hosts. To calculate shortest paths, the policy accesses the built-in state components for the set of switches, the set of links, and the mapping of hosts to network locations. In addition, the policy accesses a set of hosts which it considers to be bad hosts and which should not send or receive traffic. The set of bad hosts can be modified at runtime by an administrator.

The complete forwarding function `forward`, shown in Listing 3.9, starts by accessing the `badHostSet` to determine whether the sender or receiver are bad hosts. If so, the packets are dropped, using the `nullRoute` route. Otherwise, the function calls `shortest(pkt)`, accesses the host location table and topology state, computes the shortest path using a user-defined algorithm and returns the resulting route with the `unicast` route function. In particular, notice that the policy does not mention details about how to update forwarding table state given either changes in network topology or bad host state changes.

3.5 Maple Architecture

A naive implementation of the algorithmic policy abstraction would simply invoke the function f on every packet at a central server. This, however, would yield unusable performance and scalability.

The key component of Maple is an optimizer, or tracing runtime, which automatically discovers reusable (i.e. cacheable) algorithmic policy executions at runtime, offloads work to switches when possible, and invalidates cached policy executions due to environment changes. The tracing runtime is implemented over a run-time sched-

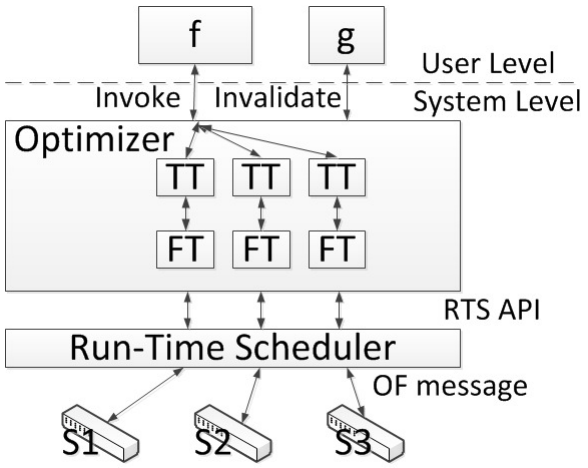


Figure 3.6: Maple system components.

uler, or scheduler for short, implemented using McNettle, which provides scalable execution of policy "misses" generated by the many switches in a large network on multicore hardware. Figure 3.6 illustrates the positions of the two components.

In addition, Maple allows a set of higher-level tools to be built on top of the basic abstraction: (1) deployment portal, which imposes constraints on top of Maple, in forms such as best practices, domain-specific analyzers, and/or higher-level, limited configuration interfaces to remove some flexibility of Maple; and (2) policy composer, which an SDN programmer can introduce.

This section sketches a high-level overview of the techniques, leaving technical details to subsequent chapters.

3.5.1 Optimizer

Although a policy `f` might in principle follow a different execution path and yield a different result for every packet, in practice many packets—and often many flows—follow the same or similar execution paths in realistic policies. For example, consider the example algorithmic policy in Figure 3.4, which we denote as `f` here. `f` assigns the same path to two packets if they match on source and destination MAC addresses and neither has a TCP port value 22. Hence, if we invoke `f` on one packet, and then a second packet arrives, and the two packets satisfy the preceding condition, then the first invocation of `f` is reusable for the second packet. The key objective of the optimizer is to leverage these reusable algorithm executions.

Recording reusable executions

The technique used by Maple to detect and utilize reusable executions of a potentially complex program `f` is to record the essence of its decision dependencies: the data accesses (*e.g.*, reads and assertions) of the program on related inputs. We call a sequence of such data accesses a *trace*, and Maple obtains traces by logging data accesses made by `f`.

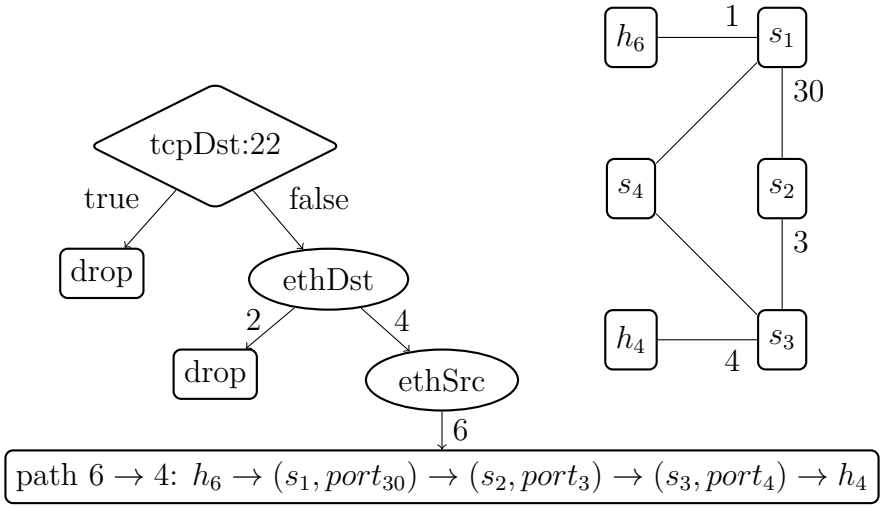


Figure 3.7: An example trace tree. Diamond indicates a test of some condition, circle indicates reading an attribute, and rectangles contain return values. tcpDst denotes TCP destination; ethDst and ethSrc denote Ethernet destination and source. Topology of the network is shown on the right.

As an example, assume that the log during one execution of a function f is follows: (1) the only data access of the program is to apply a test on TCP destination port for value 22, (2) the test is true, and (3) the program drops the packet. One can then infer that if the program is again given an arbitrary packet with TCP destination port 22, the program will similarly choose to drop the packet.

The key data structure maintained by the optimizer is a collection of such data access traces represented as a *trace tree*. Figure 3.7 is a trace tree from 3 traces of a program `f`. In one of these traces, the program first tests TCP destination port for 22 and the result is false (the right branch is false). The program then reads the field value of Ethernet destination (=4) and Ethernet source (=6), resulting in the program's decision to forward packets from host 6 to host 4 along the shortest path between these two hosts. For concreteness, Figure 3.7 shows the details of how a path is represented. Assume that host 6 is attached at switch s_1/port 1, and host 4 at switch s_3/port 4. Figure 3.7 shows the detail of the path from host 6 to host 4 as: from host 6 (s_1/port 1) to output port 30 of s_1, which is connected to switch s_2; s_2 sends out at port 3, which is connected to s_3, where host 4 is attached at port 4. The trace tree abstracts away the details of `f` but still retains its output decisions as well as the decisions' dependencies on the input packets.

Utilizing distributed flow tables

Merely caching prior policy decisions using trace trees would not make SDN scalable if the controller still had to apply these decisions centrally to every packet. Real scalability requires that the controller be able to "push" many of these packet-level decisions out into the flow tables distributed on the individual OpenFlow switches to make quick, on-the-spot per-packet decisions.

To achieve this goal, the optimizer maintains, logically, a trace tree for each switch, so that the leaves for a switch's trace tree contain the forwarding actions required for that switch only. For example, for the trace tree shown in Figure 3.7, the switch-specific trace tree maintained by Maple for switch s_1 has the same structure, but includes only port actions for switch s_1 at the leaves (*e.g.*, the right most leave is labeled only port 30, instead of the whole path).

Given the trace tree for a switch, the optimizer compiles the trace tree to a prioritized set of flow rules, to form the flow table of the switch. In particular, there are two key challenges to compile an efficient flow table for a switch. First, the table size at a switch can be limited, and hence it is important to produce a compact table to fit more cached policy decisions at the switch. Second, the optimizer will typically operate in an online mode, in which it needs to continuously update the flow table as new decisions are cached. Hence, it is important to achieve fast, efficient flow table updates. To address the challenges, our optimizer introduces multiple techniques: (1) it uses incremental compilation, avoiding full-table compilation; (2) it optimizes the number of rules used in a flow table, through both switch-local and network-wide optimizations on switch tables; and (3) it minimizes the number of priorities used in a flow table, given that the update time to a flow table is typically proportional to the number of priority levels [59].

Keeping trace trees, flow tables up-to-date

Just as important as using distributed flow tables efficiently is keeping them up-to-date, so that stale policy decisions are not applied to packets. Specifically, the decision of `f` on a packet depends on not only the fields of the packet, but also other variables. For example, accesses by `f` through the `State` argument to access the network topology will also generate dependency, which Maple tracks. Hence, trace trees record the dependencies of prior policy decisions not only on packet fields but also Maple-maintained and user-defined environment state such as network topology and policy settings.

Maple provides a general-purpose and flexible mechanism that allows algorithmic policies to record dependencies on external state components and for programs to indicate when the values of particular external state components change. This mechanism is used internally by Maple itself to track dependencies on Maple-maintained state, such as dependencies on port and link operational status and host locations. Furthermore, the mechanism is extensible: it allows algorithmic policies to depend on user-defined state components. This mechanism is used to implement a variety of abstract data types, such as variables, sets, and associative arrays, which implement data-type specific dependency tracking using the Maple-provided mechanism. Both system event handlers and user-defined functions (*i.e.*, `g` in Figure 1) can execute queries and operations on these user-defined state objects.

3.5.2 Multicore Scheduler

Even with efficient distributed flow table management, some fraction of the packets will "miss" the cached policy decisions at switches and hence require interaction with the central controller. Thus, controller-side processing of misses must scale gracefully if the SDN as a whole is to scale. Maple leverages the multicore scalability of McNettle to provide scalable and high-performance event processing for user-defined functions that induce a large number of control events.

3.6 Summary

This chapter introduced the algorithmic policy programming model, which allows programmers to specify network behavior through a centralized program with global visibility that processes every packet entering the network. The program is expressed as an ordinary program in a general-purpose language, using a convenient API to access packet and state attributes. This chapter also provided an overview of Maple's tracing runtime system, which provides an efficient implementation of algorithmic policies on OpenFlow network elements. In particular, the tracing runtime discovers the dependencies of a computation on packet and state components, constructs and maintains a partial decision tree representing an algorithmic policy's logic in the current network and policy state, and generates compact flow tables for all switches in the network that correctly implement the algorithmic policy.

Chapter 4

Automatic Generation of Compact Flow Tables

In this chapter, we present the optimizer, highlighting the construction of trace trees and methods for converting trace trees to flow tables. To aid understanding, we present the ideas in steps, from basic ideas to optimizations.

4.1 Basic Concepts

Trace tree: A trace tree provides an abstract, partial representation of an algorithmic policy. We consider packet attributes $a_1, \ldots a_n$ and write $p.a$ for the value of the a attribute of packet p. We write $dom(a)$ for the set of possible values for attribute a: $p.a \in dom(a)$ for any packet p and attribute a.

Definition 1 (Trace Tree) *A* trace tree (TT) *is a rooted tree where each node t has a field $type_t$ whose value is one of* $\mathbf{L}$ *(leaf),* $\mathbf{V}$ *(value),* $\mathbf{T}$ *(test), or* $\mathbf{\Omega}$ *(empty) and such that:*

1. *If $type_t = \mathbf{L}$, then t has a $value_t$ field, which ranges over possible return values of the algorithmic policy. This node represents the behavior of a program that returns $value_t$ without inspecting the packet further.*

2. *If $type_t = \mathbf{V}$, then t has an $attr_t$ field, and a $subtree_t$ field, where $subtree_t$ is an associative array such that $subtree_t[v]$ is a trace tree for value $v \in keys(subtree_t)$. This node represents the behavior of a program that if the supplied packet p satisfies $p.attr_t = v$, then it continues to $subtree_t[v]$.*

3. *If $type_t = \mathbf{T}$, then t has an $attr_t$ field, a $value_t$ field, such that $value_t \in dom(attr_t)$, and two subtree fields t_+ and t_-. This node reflects the behavior of a program that tests the assertion $p.attr_t = value_t$ of a supplied packet p and then branches to t_+ if true, and t_- otherwise.*

4. *If $type_t = \mathbf{\Omega}$, then t has no fields. This node represents arbitrary behavior (*i.e., *an unknown result).*

Algorithm 1: SEARCHTT(t, p)

```
1  while true do
2      if type_t = Ω then
3          return NIL;
4      else if type_t = L then
5          return value_t;
6      else if type_t = V ∧ p.attr_t ∈ keys(subtree_t) then
7          t ← subtree_t[p.attr_t];
8      else if type_t = V ∧ p.attr_t ∉ keys(subtree_t) then
9          return NIL;
10     else if type_t = T ∧ p.attr_t = value_t then
11         t ← t_+;
12     else if type_t = T ∧ p.attr_t ≠ value_t then
13         t ← t_−;
```

Given a TT, one can look up the return value of a given packet, or discover that the TT does not include a return value for the packet. Algorithm 1 shows the SEARCHTT algorithm, which defines the semantics of a TT. Given a packet and a TT, the algorithm traverses the tree, according to the content of the given packet, terminating at an **L** node with a return value or an **Ω** node which returns NIL.

Flow table (FT): A pleasant result is that given a trace tree, one can generate an OpenFlow flow table (FT) efficiently.

To demonstrate this, we first model an FT as a collection of *FT rules*, where each FT rule is a triple $(priority, match, action)$, where *priority* is a natural number denoting its priority, with the larger the value, the higher the priority; *match* is a collection of zero or more (packet attribute, value) pairs, and *action* denotes the forwarding action, such as a list of output ports, or ToController, which denotes sending to the controller. Matching a packet in an FT is to find the highest priority rule whose *match* field matches the packet. If no rule is found, the result is ToController. Note that FT matches do not support negations; instead, priority ordering may be used to encode negations (see below).

Trace tree to forwarding table: Now, we describe BUILDFT(t), a simple algorithm shown in Algorithm 2 that compiles a TT rooted at node t into an FT by recursively traversing the TT. It is simple because its algorithm structure is quite similar to the standard in-order tree traversal algorithm. In other words, the elegance of the TT representation is that one can generate an FT from a TT using basically simple in-order tree traversal.

Specifically, BUILDFT(t), which starts at line 1, first initializes the global *priority* variable to 0, and then starts the recursive BUILD procedure. Note that each invocation of BUILD is provided with not only the current TT node t, but also a match parameter denoted m, whose function is to accumulate attributes read or positively

tested along the path from the root to the current node t. We can see that BUILDFT(t) starts BUILD with m being *any* (*i.e.*, match-all-packets). Another important variable maintained by BUILD is *priority*. One can observe an invariant (lines 8 and 16) that its value is increased by one after each output of an FT rule. In other words, BUILD assigns each FT rule with a priority identical to the order in which the rule is added to the FT.

BUILD processes a node t according to its type. First, at a leaf node, BUILD emits an FT rule (at line 7) with the accumulated match m and the designated action at the leaf node. The priority of the FT rule is the current value of the global variable *priority*. Second, at a **V** node (line 9), BUILD recursively emits the FT rules for each subtree branch. Before proceeding to branch with value v, BUILD adds the condition on the branch (denoted $attr_t : value$) to the current accumulated match m. We write $m_1 \wedge m_2$ to denote the intersection of two matches.

The third case is a **T** node t. One might think that this is similar to a **V** node, except that a **T** node has only two branches. Unfortunately, flow tables do not support negation, and hence BUILD cannot include the negation condition in the accumulated match condition when recursing to the negative branch. To address the issue, BUILD uses the following techniques. (1) It emits an FT rule, which we call the *barrier* rule for the t node, with action being ToController and match m_t being the intersection of the assertion of the **T** node (denoted as $attr_t : value_t$) and the accumulated match m. (2) It ensures that the barrier rule has a higher priority than any rules emitted from the negative branch. In other words, the barrier rule prevents rules in the negated branch from being executed when m_t holds. (3) To avoid that the barrier rule blocks rules generated from the positive branch, the barrier rule should have lower priority than those from the positive branch. Formally, denote r_b as the barrier rule at a **T** node, r_- a rule from the negative branch, and r_+ a rule from the positive branch. BUILD needs to enforce the following ordering constraints:

$$r_- \rightarrow r_b \rightarrow r_+, \tag{4.1}$$

where $r_1 \rightarrow r_2$ means that r_1 has lower priority than r_2.

Since BUILD increases priority after each rule, enforcing the preceding constraints is easy: in-order traversal, first negative and then positive. One can verify that the run-time complexity of BUILDFT(t) is $\mathcal{O}(n)$, where n is the size of the trace tree rooted at t.

Example: We apply BUILDFT to the root of the trace tree shown in Figure 3.7, for switch s1 (*e.g.*, leaves containing only actions pertaining to s1, such as drop and port 30). Since the root of the tree is a **T** node, testing on TCP destination port 22, BUILD, invoked by BUILDFT, goes to line 12. Line 13 recursively calls BUILD on the negative (right) branch with match m still being *any*. Since the node (labeled ethDst) is a **V** node on the Ethernet destination attribute, BUILD proceeds to visit each subtree, adding a condition to match on the Ethernet destination labeled on the edge to the subtree. Since the subtree on edge labeled 2 is a leaf, BUILD (executing at line 7) emits an FT rule with priority 0, with a match on Ethernet destination 2. The action is to drop the packet. BUILD then increments the priority, returns to

Algorithm 2: BUILDFT(t)

```
1  Algorithm BUILDFT(t)
2      priority ← 0;
3      BUILD(t,any);
4      return;
5  Procedure BUILD(t,m)
6      if type_t = L then
7          emitRule(priority, m, value_t);
8          priority ← priority + 1;
9      else if type_t = V then
10         for v ∈ keys(subtrees_t) do
11             BUILD(subtrees_t[v], m ∧ (attr_t : v));
12     else if type_t = T then
13         BUILD(t_−, m);
14         m_t = m ∧ (attr_t : value_t);
15         emitRule(priority, m_t, ToController);
16         priority ← priority + 1;
17         BUILD(t_+, m_t);
```

the parent, and visits the subtree labeled 4, which generates an FT rule at priority level 1 and increments the priority. After returning from the subtree labeled 4, BUILD backtracks to the **T** node (tcpDst:22) and outputs a barrier rule with priority 2, with match being the assertion of the node: matching on TCP destination port 22. BUILD outputs the final FT rule for the positive subtree at priority 3, dropping packets to TCP destination port 22. The final FT for switch s1 is:

```
[ (3, tcp_dst_port=22       , drop),
  (2, tcp_dst_port=22       , toController),
  (1, eth_dst=4 && eth_src=6, port 30),
  (0, eth_dst=2             , drop)]
```

If one examines the FT carefully, one may observe some inefficiencies in it, which we will address in Section 4.3. A key at this point is that the generated FT is correct:

Theorem 1 (FT correctness) *tree and* BUILDFT(*tree*) *encode the same function on packets.*

4.2 Trace Tree Augmentation

With the preceding basic concepts, we now describe our tracing runtime system, to answer the question: how does Maple transparently generate a trace tree from an arbitrary algorithmic policy?

Maple packet access API: Maple builds trace trees with a simple requirement from algorithmic policies: they access the values of packet attributes and perform boolean assertions on packet attributes using the Maple packet access API:

```
readPacketField :: Field -> Value
testEqual       :: (Field,Value) -> Bool
ipSrcInPrefix   :: IPPrefix -> Bool
ipDstInPrefix   :: IPPrefix -> Bool
```

The APIs simplify programming and allow the tracing runtime to observe the sequence of data accesses and assertions made by a policy. A language specific version of Maple can introduce wrappers for these APIs. For example, `pkt.eth_src()` used in Section 2 is a wrapper invoking `readPacketField` on Ethernet source.

Trace: Each invocation of an algorithmic policy that uses the packet access API on a particular packet generates a *trace*, which consists of a sequence of *trace items*, where each trace item is either a *Test* item, which records an assertion being made and its outcome, or a *Read* item, which records the field being read and the read value. For example, if a program calls `testEqual(tcpDst, 22)` on a packet and the return is false, a *Test* item with assertion of TCP destination port being 22 and outcome being false is added to the trace. If the program next calls `readPacketField(ethDst)` and the value 2 is returned, a *Read* item with field being Ethernet destination and value being 2 will be appended to the trace. Assume that the program terminates with a returned action of drop, then drop will be set as the returned action of the trace, and the trace is ready to be added to the trace tree.

Augment trace tree with a trace: Each algorithmic policy starts with an empty trace tree, represented as Ω. After collecting a new trace, the optimizer *augments* the trace tree with the new trace. The AugmentTT(t, *trace*) algorithm, presented in Algorithm 3, adds a new trace *trace* to a trace tree rooted at node t. The algorithm walks the trace tree and the trace in lock step to find the location at which to extend the trace tree. It then extends the trace tree at the found location with the remaining part of the trace. The algorithm uses *head*(*trace*) to read the first item of a trace, and *next*(*trace*) to remove the head and return the rest. The algorithm uses a straightforward procedure TraceToTree(*trace*), which we omit here, that turns a linear list into a trace tree.

Example: Figure 4.1 illustrates the process of augmenting an initially empty tree. The second tree results from augmenting the first tree with trace *Test*(*tcpDst*, 22; False), *Read*(*ethDst*; 2); action=drop. In this step, AugmentTT calls TraceToTree at the root. Note that TraceToTree always places an Ω node in an unexplored branch of a **T** node, such as the t_+ branch of the root of the second tree. The third tree is derived from augmenting the second tree with the trace *Test*(*tcpDst*, 22; False), *Read*(*ethDst*; 4), *Read*(*ethSrc*, 6); action=*port 30*. In this case, the extension is at a **V** node. Finally, the fourth tree is derived by augmenting the third tree with trace *Test*(*tcpDst*, 22; True); action=*drop*. This step fills in the positive branch of the root.

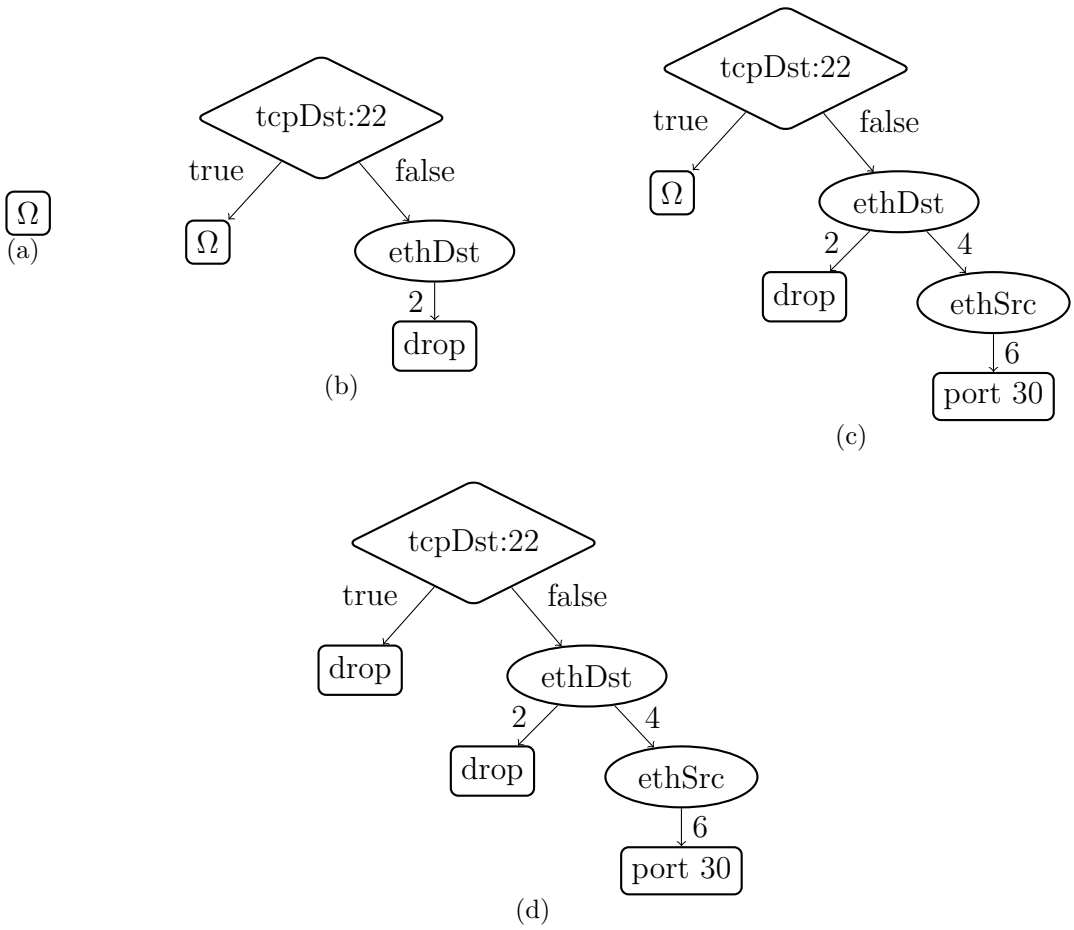


Figure 4.1: Augmenting a trace tree for switch s_1. Trace tree starts as empty ($\mathbf{\Omega}$) in (a).

Correctness: The trace tree constructed by the preceding algorithm returns the same result as the original algorithmic policy, when there is a match. Formally, we have:

Theorem 2 (TT Correctness) *Let t be the result of augmenting the empty tree with the traces formed by applying the algorithmic policy f to packets $pkt_1 \ldots pkt_n$. Then t safely represents f in the sense that if* SEARCHTT(t, pkt) *is successful, then it has the same answer as $f(pkt)$.*

Optimization: trace compression: A trace may have redundancy. Specifically, although the number of distinct observations that a program `f` can make of the packet is finite, a program may repeatedly observe or test the same attribute (field) of the packet, for example during a loop. This can result in a large trace, increasing the cost of tracing. Furthermore, redundant trace nodes may increase the size of the trace tree and the number of rules generated.

Maple applies COMPRESSTRACE, Algorithm 4, to eliminate both read and test redundancy in a trace before applying the preceding augmentation algorithm. In

Algorithm 3: AUGMENTTT($t, trace$)

```
1  if type_t = Ω then
2      t ← TRACETOTREE(trace);
3      return;
4  repeat
5      item = head(trace); trace ← next(trace);
6      if type_t = T then
7          if item.outcome is true then
8              if type_{t+} = Ω then
9                  t_+ ← TRACETOTREE(trace);
10                 return;
11             else
12                 t ← t_+ ;
13         else
14             if type_{t−} = Ω then
15                 t_− ← TRACETOTREE(trace);
16                 return;
17             else
18                 t ← t_− ;
19     else if type_t = V then
20         if item.value ∈ keys(subtree_t) then
21             t ← subtree_t[item.value];
22         else
23             subtree_t[item.value] ← TRACETOTREE(trace);
24             return;
25 until;
```

particular, the algorithm tracks the subset of packets that may follow the current trace. When it encounters a subsequent data access, it determines whether the outcome of this data access is completely determined by the current subset. If so, then the data access is ignored, since the program is equivalent to a similar program that simply omits this redundant check. Otherwise, the data access is recorded and the current subset is updated.

4.3 Rule & Priority Optimization

Motivation: With the basic algorithms covered, we now present optimizations. We start by revisiting the example that illustrates BUILDFT at the end of Section 4.1. Let $\texttt{FT}_\texttt{B}$ denote the example FT generated. Consider the following FT, which we refer

Algorithm 4: compressTrace()

```
for next access entry on attribute a do
    if range(entry) included in knownRange then
        ignore
    else
        update knownRange
```

as $\texttt{FT}_0$:

```
[ (1, tcp_dst_port=22         , drop),
  (0, eth_dst==4 && eth_src==6, port 30),
  (0, eth_dst==2              , drop)]
```

One can verify that the two FTs produce the same result. In other words, the example shows that buildFT has two problems: (1) it may generate more flow table rules than necessary, since the barrier rule in $\texttt{FT}_\texttt{B}$ is unnecessary; and (2) it may use more priority levels than necessary, since $\texttt{FT}_\texttt{B}$ has 4 priorities, while $\texttt{FT}_\texttt{0}$ has only 2.

Reducing the number of rules generated for an FT is desirable, because rules are often implemented in TCAMs where rule space is limited. Reducing the number of priority levels is also beneficial, because TCAM update algorithms often have time complexity linear in the number of priority levels needed in a flow table [59].

Barrier elimination: We start with eliminating unnecessary barrier rules. buildFT outputs a barrier rule for each **T** node t. However, if the rules emitted from the positive branch t_+ of t is *complete* (*i.e.*, every packet matching t_+'s match condition is handled by t_+'s rules), then there is no need to generate a barrier rule for t, as the rules for t_+ already match all packets that the barrier rule would match. One can verify that checking this condition eliminates the extra barrier rule in $\texttt{FT}_\texttt{B}$.

Define a general predicate $isComplete(t)$ for an arbitrary tree node t: for an **L** node, $isComplete(t) = true$, since a buildFT derived compiler will generate a rule for the leaf to handle exactly the packets with the match condition; for a **V** node, $isComplete(t) = true$ if both $|subtrees_t| = |dom(attr_t)|$ and $isComplete(subtree_v)$ for each $v \in keys(subtrees_t)$; otherwise $isComplete(t) = false$ for the **V** node. For a **T** node t, $isComplete(t) = isComplete(t_-)$. We define $needsBarrier(t)$ at a **T** node t to be true if t_+ is not complete and t_- is not $\mathbf{\Omega}$, false otherwise.

Priority minimization: Minimizing the number of priorities is more involved. But as we will show, there is a simple, efficient algorithm achieving the goal, without the need to increase priority after outputting every single rule, as buildFT does.

Consider the following insight: since rules generated from different branches of a **V** node are disjoint, there is no need to use priority levels to distinguish them. Hence, the priority increment from 0 to 1 by buildFT for the example at the end of Section 4.1 is unnecessary. The preceding insight is a special case of the general insight: one can assign arbitrary ordering to two rules if their match conditions are disjoint.

Combining the preceding general insight with the ordering constraints shown in Equation (4.1) in Section 4.1, we define the minimal priority assignment problem as choosing a priority assignment P to rules with the minimal number of distinct priority values:

$$\begin{aligned} &\underset{P}{\text{minimize}} \quad |P| \\ &\text{subject to} \quad (r_i \rightarrow r_j) \wedge (r_i, r_j \text{ overlap}) : P(r_i) < P(r_j) \end{aligned} \tag{4.2}$$

One may solve Problem (2) using topological ordering. Specifically, construct a directed acyclic graph (DAG) $G_r = (V_r, E_r)$, where V_r is the set of rules, and E_r the set of ordering constraints among the rules: there is an edge from r_i to r_j iff $r_i \rightarrow r_j$ and r_i, r_j overlap. Initialize variable *priority* to 0. Assign rule priorities as follows: select all nodes without any incoming edges, assign them *priority*, delete all such nodes, increase *priority* by 1, and then repeat until the graph is empty.

An issue of the preceding algorithm is that it needs to first generate all rules, compute their ordering constraints, and then assign priorities. However, Maple's trace tree has special structures that allow us to make simple changes to BUILDFT to still use in-order trace tree traversal to assign minimal priority levels. The algorithm that we will design is simple, more efficient, and also better suited for incremental updates.

Define a weighted graph $G_o(V_o, E_o, W_o)$ for a trace tree to capture all ordering constraints. Specifically, V_o is the set of all trace tree nodes. All edges of the trace tree, except those from a **T** node t to t_-, belong to E_o as well. To motivate additional edges in E_o, consider all ordering constraints among the rules. Note that the edges of a trace tree do not include constraints of the form $r_- \rightarrow r_b$, as defined in Equation (1). To introduce such constraints in G_o, extend E_o to include *up-edges* from rule-generating nodes (*i.e.*, **T** nodes with barriers or **L** nodes). Identifying the up-edges from a rule-generating node n is straightforward. Consider the path from the tree root to node n. Consider each **T** node t along path. If n is on the negative branch of t and the intersection of the accumulated match conditions of t_+ and n is nonempty, then there should be an up-edge from n to t. One can verify that G_o is a DAG.

To remove the effects of edges that do not convey ordering constraints, we assign all edges weight 0 except the following:

1. For an edge from a **T** node t to t_+, this edge has $w(e) = 1$ if t needs a barrier and $w(e) = 0$ otherwise;

2. For each up-edge, the weight is 1.

Example: Figure 4.2 shows G_o for Figure 4.1(d). Dashed lines are up-edges of E_o. Edges that are in both the TT and E_o are shown as thick solid lines, while edges that are only in TT are shown as thin solid lines. E_o edges with weight 1 are labeled $w = 1$. The `drop` leaf in the middle has an up-edge to the root, for example, because its accumulated match condition is *ethDst* : 2, which overlaps the root's positive subtree match condition of *tcpDst* : 22. The E_o edge to the positive subtree of the root has weight 0, because the root node is complete and hence no barrier is needed for it.

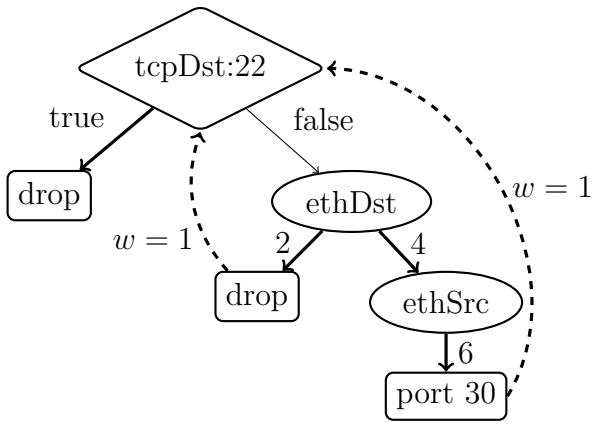


Figure 4.2: Order graph G_o for trace tree of Figure 4.1(d).

Algorithm: The algorithm based on G_o to eliminate barriers and minimize priorities is shown in optBuildFT. The algorithm traverses the TT in-order, the same as buildFT. Each TT node t has a variable *priority*(t), which is initialized to 0, and is incremented as the algorithm runs, in order to be at least equal to the priority of a processed node x plus the link weight from x to t. Since all non-zero weight edges flow from right to left (*i.e.*, negative to positive) in G_o, in-order traversal guarantees that when it is time to output a rule at a node, all nodes before it in the ordering graph have already been processed, and hence the node's priority is final.

Example 1: Consider executing optBuildFT on the example of Figure 4.2. The algorithm begins by processing the negative branch of the root. At the *ethDst* **V** node, optBuildFT updates the priority of its two children with priority 0, since its priority is 0 and the edge weights to its children are 0. The algorithm then processes the *drop* leaf for *ethDst* : 2, emits a rule at priority 0, and updates the root priority to 1, since its priority is 0 and its up-edge weight is 1. The algorithm then processes the *ethDst* : 4 branch where it processes the leaf in a similar way. After backtracking to the root, optBuildFT skips the barrier rule, since its positive subtree is complete, and updates the priority of its positive child to be 1 (its priority), since the edge weight to the positive child is 0. Finally, it proceeds to the positive child and outputs a drop rule at priority 1.

Example 2: Consider the following policy that tests membership of the IP destination of a packet in a set of IP prefixes:

```
f(p):  if p.ipDstInPrefix(103.23.0.0/16):
          if p.ipDstInPrefix(103.23.3.0/24):
            return a
          else:
            return b
       if p.ipDstInPrefix(101.1.0.0/16):
         return c
       if p.ipDstInPrefix(101.0.0.0/13):
         return d
```

Algorithm 5: optBuildFT(t)

```
1  Algorithm optBuildFT(t)
2      optBuild(t,any);
3      return;
4  Procedure update(t)
5      for (t, x) ∈ E_o do
6          priority(x) = max(priority(x), weight(t, x) + priority(t));
7  Procedure optBuild(t,m)
8      if type_t = L then
9          emitRule(priority(t), m, value_t);
10         update(t);
11     else if type_t = V then
12         update(t);
13         for v ∈ keys(subtrees_t) do
14             optBuild(subtrees_t[v], m ∧ (attr_t : v));
15     else if type_t = T then
16         build(t_−, m);
17         m_t = m ∧ (attr_t : value_t);
18         if needsBarrier(t) then
19             emitRule(priority(t), m_t, ToController);
20         update(t);
21         optBuild(t_+, m_t);
```

```
return e
```

An example trace tree of this program is shown in Figure 4.3, which also shows the G_o edges for this TT. By avoiding barrier rules and by using the dependency graph to limit priority increments, optBuildFT generates the following, optimal prioritization:

```
2, ipDst:103.23.3.0/24 --> a
2, ipDst:101.1.0.0/16  --> c
1, ipDst:101.23.0.0/16 --> b
1, ipDst:101.0.0.0/13  --> d
0, *                   --> e
```

4.4 Efficient Insertion

We now adapt the algorithms of the preceding sections to become incremental algorithms, which typically update rules by examining only a small portion of a trace

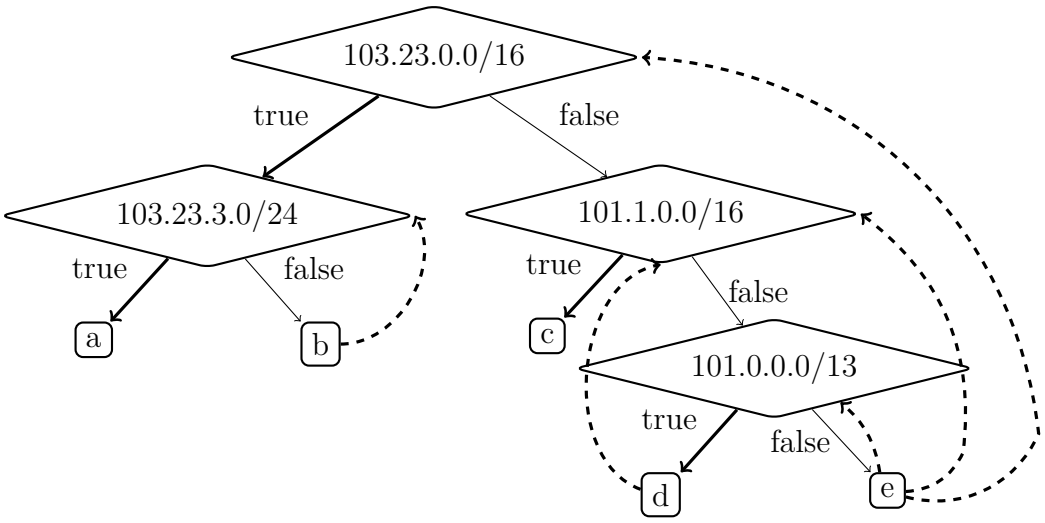


Figure 4.3: Trace tree from an algorithmic policy implementing IP prefix matching.

tree, rather than compiling the entire trace tree "from scratch". Maple allows efficient updates because of information maintained in G_o, which we call node annotations of the tree.

First consider augmenting a trace tree with a new trace. We modify AugmentTT to accumulate the identities of **T** nodes when the trace follows the negative branch. After attaching the new trace, we build rules starting with the priority stored at the attachment point. Since building rules for the new trace may increase the priorities of some of the **T** nodes accumulated above, we modify augmentation to backtrack towards the root and rebuild any positive sub-branches of those **T** nodes along this path whose priorities have increased. Note that the method avoids recompiling branches at **V** nodes other than the branch being augmented. It also avoids recompiling negative branches of positive **T** ancestors of the augmentation point.

4.5 Optimization for Distributed Flow Tables

Maple further optimizes flow table usage through network-wide optimizations, by considering network properties. Below we specify two such optimizations that Maple conducts automatically.

Elimination of ToController **at network core**: To motivate the idea, consider converting the global trace tree in Figure 2 to switch trace trees. Consider the trace tree for a switch s4, which is not on the path from host 6 to 4. If a packet from 6 to 4 does arrive at switch s4, a general, safe action should be ToController, to handle the exception (*e.g.*, due to host mobility). Now Maple uses network topology to know that switch s4 is a core switch (*i.e.*, it does not have any connection to end hosts). Then Maple is assured that the exception will never happen, and hence there is no need to generate the ToController rule for switch s4. This is an example of the general case that core switches do not see "decision misses", which are seen only by edge switches. Hence, Maple does not install ToController rules on core switches and still achieves

Algorithm 6: RouteAggregation(*ft*)

```
1 Safe elimination of rules in ft that have action ToController;
2 for each destination d mentioned in ft do
3     act = action of lowest priority rule in ft overlapping destination d ;
4     p = priority of highest priority rule that overlaps packets to d and agrees
      with act and such that all lower priority rules overlapping with packets to d
      agree with act ;
5     Delete rules at priority ≤ p matching on destination d from ft;
6     emitRule(p, matchForDest(d), act);
```

correctness. Maple can conduct further analysis on network topologies and the trace trees of neighboring switches to remove unnecessary rules from a switch.

Route aggregation: The preceding step prepares Maple to conduct more effectively a second optimization which we call route aggregation. Specifically, a common case is that routing is only destination based (*e.g.*, Ethernet, IP or IP prefix). Hence, when the paths from two sources to the same destination merge, the remaining steps are the same. Maple identifies safe cases where multiple rules to the same destination can be replaced by a single, broader rule that matches on destination only. Algorithm 6 shows the details of the algorithm. The elimination of rules with ToController as action improves the effectiveness of the optimization, because otherwise, the optimization will often fail to find agreement among overlapping rules.

Theorem 3 RouteAggregation *does not alter the forwarding behavior of the network, provided rules at ingress ports include all generated rules with action* ToController.

4.6 Maple Evaluations

In this section, we demonstrate that (1) Maple generates high quality rules, (2) Maple can achieve high throughputs on augmentation and invalidation, and (3) Maple can effectively scale controller computation over large multicore processors.

4.6.1 Quality of Maple Generated Flow Rules

We first evaluate if Maple generates compact switch flow rules.

Algorithmic policies: We use two types of policies. First, we use a simple data center routing policy named `mt-route`. Specifically, the network is divided into subnets, with each subnet assigned a /24 IPv4 prefix. The subnets are partitioned among multiple tenants, and each tenant is assigned its own weights to network links to build a virtual topology when computing shortest paths. Upon receiving a packet, the `mt-route` policy reads the /24 prefixes of both the source and the destination IPv4 addresses of the packet, looks up the tenants of the source and the destination using the IP prefixes, and then computes intra-tenant routing (same tenant) or inter-tenant routing (*e.g.*, deny or through middleboxes).

Second, we derive policies from filter sets generated by Classbench [64]. Specifically, we use parameter files provided with Classbench to generate filter sets implementing Access Control Lists (ACL), Firewalls (FW), and IP Chains (IPC). For each parameter file, we generate two filter sets with roughly 1000 and 2000 rules, respectively. The first column of Table 4.1 names the generated filter sets, and the second indicates the number of filters in each Classbench-generated filter set (except for the `mt-route` policy, which does not use a filter set). For example, `acl1a` and `acl1b` are two filter sets generated from a parameter file implementing ACL, with 973 and 1883 filters respectively. We program an `f` that acts as a *filter set interpreter*, which does the following for a given input filter set: upon receiving a packet, the policy tests the packet against each filter, in sequence, until it finds the first matching filter, and then returns an action based on the matched rule. Since TCP port ranges are not directly supported by OpenFlow, our interpreter checks most TCP port ranges by reading the port value and then performing the test using program logic. However, if the range consists of all ports the interpreter omits the check, and if it consists of a single port the interpreter performs an equality assertion. Furthermore, the interpreter takes advantage of a Maple extension which allows a user-defined `f` to perform a single assertion on multiple conditions. The interpreter makes one or more assertions per filter, and therefore makes heavy use of **T** nodes, unlike `mt-route`.

Packet-in: For each Classbench filter set, we use the trace file (*i.e.*, a sequence of packets) generated by Classbench to exercise it. Since not all filters of a filter set are triggered by its given Classbench trace, we use the third column of Table 4.1 to show the number of distinct filters triggered for each filter set. For the `mt-route` policy, we generate traffic according to [9], which provides a characterization of network traffic in data centers.

In our evaluations, each packet generates a packet-in message at a variant of Cbench [13] — an OpenFlow switch emulator used to benchmark OpenFlow controllers — which we modified to generate packets from trace files. For experiments requiring accurate measurements of switch behaviors such as flow table misses, we further modified Cbench to maintain a flow table and process packets according to the flow table. This additional code was taken from the OpenFlow reference switch implementation.

Results: Table 4.1 shows the results. We make the following observations. First, Maple generates compact switch flow tables. For example, the policy `acl1a` has 973 filters, and Maple generates a total of only 1006 OpenFlow rules (see column 4) to handle packets generated by Classbench to test `acl1a`. The number of flow rules generated by Maple is typically higher than the number of filters in a filter set, due to the need to turn port ranges into exact matches and to add barriers to handle packets from ports that have not been exactly matched yet. Define Maple compactness for each filter set as the ratio of the number of rules generated by Maple over the number of rules in the filter set. One can see (column 5) that the largest compactness ratio is for `acl3b`, which is still only 1.31. One can also evaluate the compactness by using the triggered filters in a filter set. The largest is still for `acl3b`, but at 2.09.

Second, we observe that Maple is effective at implementing complex filter sets

Alg. policy	#Filts	#Trg	#Rules	Cmpkt	#Pr	Mods/Rule
mt-route			73563		1	1.00
acl1a	973	604	1006	1.03	9	2.25
acl2a	949	595	926	0.98	85	10.47
acl3a	989	622	1119	1.13	33	2.87
fw1a	856	539	821	0.96	79	17.65
fw2a	812	516	731	0.90	56	10.66
ipc1a	977	597	1052	1.08	81	4.20
ipc2a	689	442	466	0.68	26	6.73
acl1b	1883	1187	1874	1.00	18	5.35
acl2b	1834	1154	1816	0.99	119	5.02
acl3b	1966	1234	2575	1.31	119	6.13
fw1b	1700	1099	1775	1.04	113	18.32
fw2b	1747	1126	1762	1.01	60	7.69
ipc1b	1935	1227	2097	1.08	112	9.49
ipc2b	1663	1044	1169	0.70	31	10.02

Table 4.1: Numbers of flow rules, priorities and modifications generated by Maple for evaluated policies.

with a small number of flow table priorities (column 6). For example, it uses only 9 priorities for `acl1a`, which has 973 filters. The `mt-route` policy uses only one priority level.

Third, operating in an online mode, Maple does need to issue more flow table modification commands (column 7) than the final number of rules. For example, for `acl1a`, on average, 2.25 switch modification commands are issued for each final flow rule.

4.6.2 Effects of Optimizing Flow Rules

Maple generates wildcard flow rules when possible to reduce flow table "cache" misses. To demonstrate the benefits, we use the `mt-route` policy and compare the performance of Maple with that of a simple controller that uses only exact matches.

Flow table miss rates: We measure the switch flow table miss rate, defined as the fraction of packets that are diverted from a switch to the controller, at a single switch. We generate network traffic for a number of sessions between 10 hosts and 20 servers, each in distinct tenants, with an average of 4 TCP sessions per pair of hosts. Figure 4.4(a) shows the flow table miss rates of Maple compared with those of the exact-match controller, as a function of the number of TCP packets per flow, denoted F. We vary F from 4 to 80 packets per flow, as the size of most data center flows fall in this range [9].

As expected, the exact match controller incurs a miss rate of approximately $1/F$, for example incurring 25.5% and 12.8% miss rates for $F = 4$ and $F = 8$, respectively. In contrast, Maple incurs a miss rate 3 to 4 times lower, for example 6.7% and 4.1% at $F = 4$ and $F = 8$. Maple achieves this improvement by generating rules for this

policy that match only on source and destination addresses, which therefore decreases the expected miss rate by a factor of 4, the average number of flows per host pair.

Real HP switch load: We further measure the effects using 3 real HP 5406 OpenFlow switches (s1, s2, and s3). We build a simple topology in which s1 connects to s2 and s2 connects to s3. A client running httperf [46] at subnet 1 connected at s1 makes HTTP requests to an HTTP server at subnet 3 connected at s3.

We use three controllers. The first two are the same as above: exact match and `mt-route`, which modified to match only on IP and transport fields to accommodate the switches' restrictions on which flows can be placed in hardware tables. We interpret an exact match as being exact on IP and transport fields. We introduce the third controller, which is the native L2 mode (*i.e.*, no OpenFlow) at any of the switches.

Figure 4.4(b) shows the mean end-to-end HTTP connection time, which is measured by httperf as the time between a TCP connection is initiated to the time that the connection is closed, and hence includes the time to set up flow tables at all 3 switches. The x-axis of the figure is the request rate and number of requests from the client to the HTTP server. For example, 100 means that the client issues one HTTP connection per 10 ms (=1/100 sec) for a total of 100 connections. Note that the y-axis is shown as log scale. We make the following observations. First, the HTTP connection setup time of the exact-match controller and that of Maple are the same when the connection rate is 1. Then, as the connection rate increases, since Maple incurs table misses only on the first connection, its HTTP connection time reduces to around 1 ms to slightly above 2 ms when the connection rate is between 10 to 120. In contrast, the exact-match controller incurs table misses on every connection and hence its HTTP connection time increases up to 282 ms at connection rate 120. This result reflects limitations in the switches, since the load on the controller CPU remains below 2% throughout the test, and we ensured that the switches' OpenFlow rate limiters are configured to avoid affecting the switches' performance. Second, we observe that when the connection rate increases from 80 to 100, the switch CPUs becomes busy, and the HTTP connection time starts to increase from around 1 ms to 2 ms. Third, Maple has a longer HTTP connection time compared with native L2 switch, which suggests potential benefits of proactive installation.

4.6.3 Flow Table Management Throughput

After evaluating the quality of Maple generated flow table rules, we now evaluate the throughput of Maple; that is, how fast can Maple maintain its trace tree and compile flow tables?

Types of operations: We subject Maple to two types of operations: (1) *augments*, where Maple evaluates a policy, augments the trace tree, generates flow table updates, and installs the updates at switches; (2) *lookups*, where Maple handles a packet by looking up the cached answer in the trace tree.

Workload: We use the same policies and packet traces as in Section 4.6.1, but we process each packet trace multiple times. In particular, during the first pass, as nearly every packet will cause an augment operation, we measure the throughput of

Filter set	Augments/s	Lookups/s
mt-route	58719.65	555937
acl1a	1180.74	58631
acl2a	508.40	15931
acl3a	605.34	19348
fw1a	202.17	73433
fw2a	205.52	85013
ipc1a	362.98	18053
ipc2a	621.23	50544
acl1b	666.67	40133
acl2b	245.92	9809
acl3b	223.75	9749
fw1b	68.52	32917
fw2b	51.40	25994
ipc1b	142.11	10143
ipc2b	185.07	17622

Table 4.2: Maple augments and lookup rates.

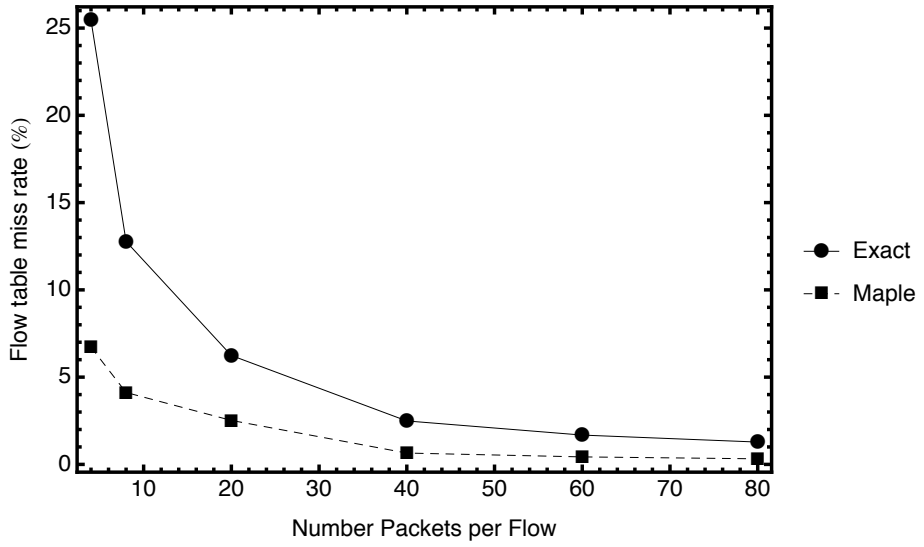
Maple to record and install new rules. During subsequent passes, we measure lookup throughput, as the trace tree has cached results for every packet in the trace.

Server: We run Maple on a Dell PowerEdge R210 II server, with 16GB DDR3 memory and Intel Xeon E31270 CPUs (with hyper-threading) running at 3.40GHz. Each CPU has 256KB L2 cache and 8MB shared L3 cache. We run CBench on a separate server and both servers are connected by a 10Gbps Ethernet network.

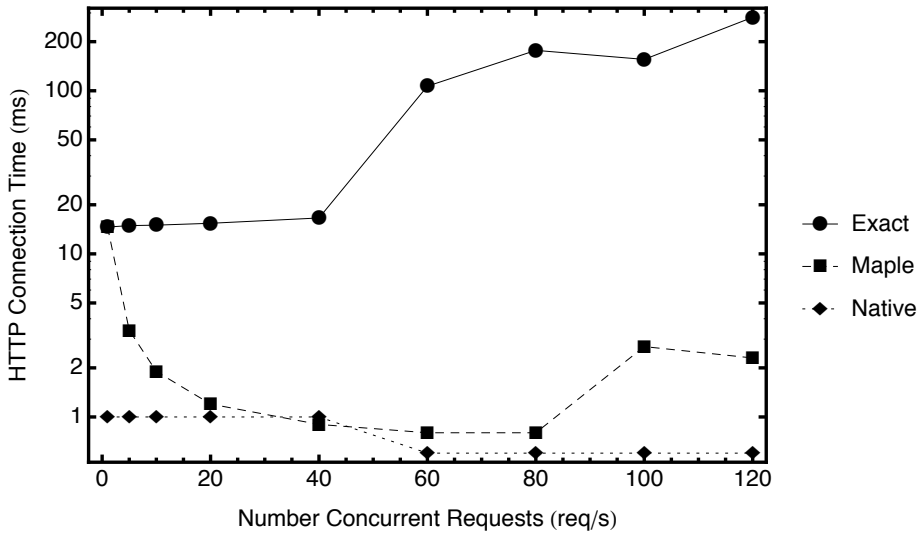
Results: Table 4.2 shows throughput for each operation type and policy, using a single 3.40 GHz core, with a single thread. For the `mt-route` policy, which uses only **V** nodes, Maple can perform all operations at high-speed, including both lookups and augments. The augmentation throughput of Classbench-based policies varies. The `fw2b` policy takes the longest time (20 ms) for Maple to handle a miss.

4.7 Summary

This chapter has introduced the trace tree data structure, the tracing of algorithmic policy computations to generate a trace tree, the compilation of trace trees to flow tables and optimizations to reduce the number of priority levels used and the number of rules required in core switches. In addition, we presented evaluations demonstrating that Maple generates effective rule sets, in particular reducing HTTP connection time of hosts in a real network by a factor of 100 at high load.



(a) Flow table miss rate



(b) HTTP connection time using real HP switches

Figure 4.4: Effects of optimizing flow rules.

Chapter 5

Automatic, Incremental Controller-Switch State Consistency

Realistic network controllers depend on state. Even the most basic network controller which forwards packets according to a shortest path algorithm relies on the current set of ports, network topology, and host locations. In addition, most network controllers provide a north-bound API that provides configuration state that drives the network's forwarding behavior. Moreover, real world north-bound APIs, such as Neutron [49] (part of the OpenStack [51] open source cloud management platform), are often complex, involving several interrelated entities with complex consistency constraints.

SDN frameworks today provide little support for coping with the challenging problem of mapping these complex controller states to distributed forwarding configurations and for maintaining consistency as the system changes state. Instead, SDN programmers today use ad-hoc methods to map each network controller's specific state space to low-level configurations.

Maple allows programmers to express stateful network controllers using algorithmic policies using a variety of data structures to model algorithmic policy state. Maple automatically maintains consistency between the controller's policy and distributed forwarding rule sets, using novel methods to precisely determine which forwarding rules are affected by a state update and to minimize the disruption caused by state changes. Evaluations with real switches and network topologies show that Maple reduces link failure recovery time by 2.5x to 6x over a widely used, open-source network controller.

We begin this chapter by providing an overview of the main techniques used to efficiently implement stateful algorithmic policies. Sections 5.2 and 5.3 present the technical details for dependency localization, speculative execution, and update cancellation and Section 5.4 presents evaluations of these techniques.

5.1 Basic Ideas

This section provides an informal overview of the most basic ideas, proceeding from basic to advanced techniques. Details and optimizations will be presented in the following sections.

Dependency tracking achieving incremental, safety update: A key challenge that is clear from the examples in Sections 3.2 and 3.3 is that incremental update of flow rules (without recomputing the flow tables completely from scratch) can cause correctness issues. Hence, the first basic idea is simple: Maple tracks the dependency of flow rules on states to prevent errors.

Specifically, consider the following typical state-dependent packet-in handler procedure when a controller generates a flow rule for a switch:

```
def onPacket(p):
 read attributes of packet p.
 read configuration state variables {X_i}.
 compute flow rule R using value of {X_i}.
 install R into switch to handle packets similar to p.
```

Now, if the controller keeps a record of all values of all state variables $\{X_i = x_i\}$ read during the generation of a flow rule R, then when the value of X_i changes, the controller knows that the rule R may no longer be valid. Hence the controller should update the flow rule R to re-send packets to the controller, in order to revisit decisions for this class of packets. At the same time, if the generation of a flow rule R' is not dependent on X_i (*i.e.*, the program does not access the state variable X_i), then the controller should not change R' when the value of X_i changes, to guarantee correctness.

Proactive repair of affected rules: An issue of the preceding basic idea is that although it guarantees correctness, it may be inefficient. In particular, after a rule is invalidated by a state change, the modification of the rule to send packets to the controller will lead to some packets of the affected flows being diverted to the controller. Moreover, this may occur even when the controller would install the same rule R despite the state change. This situation, which we call a *false positive* may occur frequently. For example, most rules will depend on network topology. On the other hand, a particular topology change may not in fact lead to any changes in chosen routes, and hence may not require any rule changes. However, most rules may become invalidated by a topology change, since rules typically require some knowledge of topology.

To reduce the packet diversions due to rules modified to send packets to the controller and to optimize in the case where dependency tracking leads to false positives, Maple proactively reevaluates the user's packet handler on representative packets of the traffic classes corresponding to the invalidated rules. This is called *speculative execution*. Speculative execution allows Maple to directly repair rules whose action has changed without diverting packets to the controller and to avoid updating rules whose actions are unchanged. In addition, Maple calculates the overall update to

apply after invalidating rules and proactively reevaluating the user's function. This is called *update minimization*.

Packet-Dependent State Packet processing functions often maintain state components based on packet arrival events. For example, many controllers maintain mappings between MAC addresses and the port on which that MAC is found (*i.e.* host location), based on packet arrival events. A second example can be found in ontrollers implementing IP forwarding, which maintain ARP caches that map IP addresses to MAC addresses, by observing ARP reply frames from hosts. Other examples include controllers that implement security between trusted and untrusted ports by opening untrusted ports only after an application on a trusted port contacts the untrusted port.

Implementing packet processing functions that maintain state based on packet arrivals can be challenging, because the implementation must ensure that generated flow rules do not prevent the observation of packet arrival events that would otherwise cause transitions to states with forwarding behavior different from the current state. For example, if a packet from a host on a trusted port is forwarded directly to an untrusted host without notifying the controller of the event, then the untrusted host will be unable to reply to send packets back to the trusted host, even though it should now be trusted. In addition, a rule handling packets whose observation would not lead to state changes in one state may later obscure state changes after the system state changes.

Therefore, to effectively generate flow rules for algorithmic policies with packet arrival-driven state, the Maple tracing runtime includes an *execution idempotence detection* algorithm, which detects when an execution is such that any other packets which could lead to an execution with the same trace do not change system state further. When an idempotent execution is detected, Maple may safely installs rules matching the execution's packets. In addition, the tracing runtime tracks which state components are written to by an execution, in order to ensure that rules installed for that execution are removed after later changes to those components.

5.2 Automatic Change Management

Our goal is to faithfully implement the algorithmic policy model. In particular, the Maple system should behave as if the algorithmic policy operates on every packet and event. Logically, we can consider an execution of an algorithmic policy `f` as a sequence of state transitions:

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$$

where each transition is caused by either a packet arrival or some other (*e.g.* topology change) event. Of course, to provide a scalable system, our implementation must distribute rules to switches so that most of the packet processing is performed locally, without involving the controller.

At a high level, our strategy is to distribute rules to switches so that (1) the

forwarding behavior of the switches is exactly the same as the forwarding behavior of `f` in the current state s, and (2) any packet p forwarded locally by the rule set would not lead to a distinct state s' if `f` were executed on packet p in the current state s. If these two conditions hold, then the system will clearly induce the same state transitions in the Maple program as if it had executed every packet at the controller. As a consequence of this conclusion and condition (1), the overall forwarding behavior of the Maple system executing `f` will be equivalent to the naive execution of `f` on every packet and system event.

The Maple trace tree is initially empty and therefore all packets will be directed to the controller. Therefore, the correctness condition is trivially satisfied in the initial system state.

We now consider how to maintain the correctness condition in every state change. Note that although a packet p' may have safely matched a switch rule in state s, the transition to state s' may cause two distinct violations of the correctness condition. One, the forwarding behavior in state s' may no longer be consistent with the distributed rules. Two, some packet p' may cause s' to take a transition to a new state s'', and therefore should not be handled locally by a switch, even though it may have previously been handled locally. Therefore, previously installed rules may need to be revised or removed to maintain the above-stated correctness condition.

Forwarding State To help formulate our system's operation and correctness, we first refine our notion of the system state. We let $D_1, \ldots, D_n$ be the collection of forwarding data structures declared in f. We view each D_i as a state-dependent function from some domain X_i to some range Y_i. We let $D_i^s : X_i \to Y_i$ denote the function represented by D_i in state s. We use the notation $dom(D_i)$ to refer to the domain X_i of D_i.

Cacheable Executions An execution of `f` in state s on packet p is an invocation of the user's Maple program in the tracing runtime. We define notation to refer to several attributes associated with an execution ex. We write *ex.packet*, *ex.result*, *ex.start*, *ex.end* to denote, respectively, the packet that `f` was applied to for ex, the final result (*i.e.* the return value) of the execution, the state of the system when the execution was started, and the state of the system after the execution.

We let *ex.trace* denote the sequence $o_1, \ldots, o_m$ of packet and data access API calls made during the execution, where each operation o_i is either a packet attribute access trace item or one of the following two items:

1. $R(D, x)$, a read of a state component D at argument x.
2. $W(D, x, y)$, an update of state component D at argument x to value y.

In addition, we let *ex.pmatch* denote the set of packets which have the same values as *ex.packet* for all the packet attributes accessed and assertions made during the execution.

We define the notion of cacheable executions to specify when it is safe to install an execution into our trace tree:

Definition 2 (Cacheable execution in state s) *We say that an execution ex is cacheable in state s if for all packets $p \in ex.pmatch$*

$$(ex.result, s) = f^s(p)$$

In other words, an execution ex made in s is no longer cacheable in s' if there is even a single packet p' that matches the class of packets identified by the execution such that $f^s(p')$ would either return a result that differs from *ex.result* or would change the system state.

Given this definition, we can now restate our correctness property: in each state s the distributed rules are the result of compiling a trace tree containing only cacheable executions. To ensure correctness our implementation will ensure that (1) all executions added to the trace tree are cacheable in the current state, and (2) all prior executions that become non-cacheable are removed from the trace tree.

Idempotent State Changes Clearly, an execution ex which does not change the state is cacheable in the final state. However, many executions change state; *e.g.* a controller that learns host locations from packets. On the other hand, many state-changing executions are cacheable, since executing them more than once will not change the state further; *i.e.* they are idempotent. We now formulate a criterion that identifies many idempotent state-changing executions of `f`.

Consider the set W^{ex} of components changed during the execution ex:

$$W^{ex} = \{(D, x) : D^{ex.start}(x) \neq D^{ex.end}(x)\}.$$

In addition, consider the set R^{ex} of components read during the execution before any writes:

$$\begin{aligned} R^{ex} = \{(D, x) :& \exists i \text{ with } o_i = R(D, x) \text{ and} \\ & \neg\exists j < i \text{ with } o_j = W(D, x, y)\}. \end{aligned}$$

Then ex is cacheable in *ex.end* whenever $R^{ex} \cap W^{ex} = \emptyset$. We consider two examples to illustrate these definitions and this criteria. Consider a first example where the packet-processing function updates the host location for the source address and then looks up the locations of the sender and receiver addresses. First, notice that executing this function consecutively on a single packet will lead to no state change on the second execution, since the update to the location of the sender is the same in both cases. For this controller, we may obtain an execution trace ex such as:

$$\begin{aligned} ex = & \mathit{Read}(\mathit{inport}), \mathit{Read}(\mathit{ethSrc}), W(D_{\text{loc}}, \mathit{ethSrc}, \mathit{inport}), \\ & R(D_{\text{loc}}, \mathit{ethSrc}), \mathit{Read}(\mathit{ethDst}), R(D_{\text{loc}}, \mathit{ethDst}), \mathit{Return}(\mathit{action}_1) \end{aligned}$$

Supposing that this execution updated the sender's location, we have $W^{ex} = \{(D_{\text{loc}}, \mathit{ethSrc})\}$ and $R^{ex} = \{(D_{\text{loc}}, \mathit{ethDst})\}$. The intersection of these sets is empty, correctly identifying this computation as idempotent.

On the other hand, consider a packet processing function that increments a counter

counter on every packet. Executions of this function are not cacheable, since a single packet would change states in consecutive executions. An execution trace of this function may be

$$ex' = R(D_{\text{counter}}, ()), W(D_{\text{counter}}, (), x)$$

for some value x which is one more than the value of the *counter* previously read. In this case $W^{ex} = \{(D_{\text{counter}}, ())\}$ and $R^{ex} = \{(D_{\text{counter}}, ())\}$. The intersection of these sets is non-empty, indicating that this execution is not idempotent.

More formally, consider executing $f^{ex.end}(p)$ where $p \in ex.pmatch$ and obtaining execution ex'. In particular, we prove that $ex.trace = ex'.trace$ by showing by induction on the prefixes of $ex'.trace$ that the successive operations in ex and ex' are identical, with each action returning the same value to the program. Clearly the proposition holds for the empty prefix of operations in ex'. Suppose the proposition holds for prefix $o_1, \ldots o_{i-1}$ and consider o_i. If o_i is a packet access attribute, then since $p \in ex.pmatch$ the operation returns the same value as in ex. Alternatively, suppose o_i is $R(D, x)$. If $(D, x) \in R^{ex}$, then by disjointness of R^{ex} and W^{ex} we have that $D^s(x) = D^{s'}(x)$ (since ex left (D, x) unchanged) and by induction the read therefore returns the same value as in ex. Otherwise, $(D, x) \notin R^{ex}$ and (D, x) is read after a closest preceding $W(D, x, y)$ operation. By induction, the $W(D, x, y)$ operation is identical as in ex and hence (D, x) has the same value at this operation as in the corresponding operation in ex. Similarly, if $o_i = W(D, x, y)$, then clearly the operation has the same result. By induction, the traces of ex and ex' are identical, and hence they execute the same write operations, leaving the state unchanged.

We therefore enhance the tracing runtime system to maintain a read set R and a write set W, where each set consists of a collection of pairs (D, x) where $x \in dom(D)$ and is initialized to $\emptyset$. Whenever a read of (D, x) is performed during the execution, we add (D, x) to R if $(D, x) \notin W$. Whenever a write is made to (D, x) we add (D, x) to W. As shown above, the execution is then cacheable whenever $R \cap \{(D, x) \in W : D^s(x) \neq D^{s'}(x)\} = \emptyset$ at the end of the execution.

Dependency Localization To provide precise information on when executions started in state s become uncacheable, Maple maintains a *dependency table*, which we denote with *depTable*. The *depTable* precisely pinpoints the executions which depend on a particular state component. Each entry in the dependency table $depTable(D, x)$ consists of a set of execution traces, which will be maintained to satisfy the following Prefix Property:

Definition 3 (Prefix Property) *The set of traces τ has the Prefix Property for data structure D and argument $x \in dom(D)$ if, for every cached execution ex such that $(D, x) \in R^{ex} \cup \{(D, x) : W(D, x, y) \in ex.trace\}$, there is some $t' \in \tau$ such that t' is a prefix of $ex.trace$.*

The Prefix Property ensures that the dependency table provides a straightforward algorithm to determine which recorded executions are no longer valid when a particular data structure value is changed. Concretely, suppose that an event causes the system to perform transition $s \to s'$. Suppose further that this transition consists of

Algorithm 7: depChange($\{(D_1, x_1), \ldots, (D_m, x_m)\}$)

```
for trace ∈ ∪_{j∈{1...n}} depTable(D_j, x_j) do
    trace' = removeTrace(traceTree, trace)
    for (D, x) ∈ keys(depTable) do
        removeDepTrace(depTable(D, x), trace');
depTable(D_1, x_1) = ··· = depTable(D_m, x_m) = ∅
```

a collection of changes to forwarding data structures:

$$D_1^s(x_1) \neq D_1^{s'}(x_1), \ldots, D_m^s(x_m) \neq D_m^{s'}(x_m)$$

The only traces that may no longer be cacheable in s' are:

$$T = \bigcup_{j \in \{1,\ldots,m\}} depTable(D_j, x_j)$$

To see this, consider some execution ex that was cacheable in s and whose trace $ex.trace \notin T$. Since $ex.trace \notin T$, ex only evaluated state components (D, x) which are unchanged in s'. Therefore, evaluating $f^{s'}(p)$ on any packet $p \in ex.pmatch$ would execute the same set of reads and writes and would return the same final value. Moreover, the writes are to components (D, x) which are unchanged in s' and hence the writes would also not change s'. Therefore, ex remains cacheable in s'.

Algorithm 7 exploits the above property to handle updates to a set of state components $\{(D_1, x_1), \ldots, (D_m, x_m)\}$. In particular, for each *trace* in T, the algorithm removes *trace* from the trace tree using removeTrace. The removeTrace algorithm, which we omit here, descends into the trace tree to find the subtree of least depth which contains only trace *trace*. We denote the trace up to this subtree as *trace*′. The removeTrace algorithm removes all flow table entries stored at the leaves of the subtree identified by *trace*′ and returns *trace*′. Following this, the algorithm removes any extensions of *trace*′ from all other *depTable* entries using the removeDepTrace algorithm, which we discuss later. This ensures that each entry of *depTable* includes only traces with a non-empty set of extensions that are in the trace tree.

Building the Dependency Table We have seen how we can use the dependency table to restore the correctness condition after any transition $s \to s'$. A simple procedure ensures that we construct *depTable* to satisfy the prefix property.

In particular, the dependency table is maintained by intercepting reads or writes to (D, x) during execution of the forwarding function `f`. Algorithm 8 and 9 show the algorithms for reads and writes to (D, x). These algorithms both update *depTable* and maintain the R and W sets for the execution. To avoid recording dependencies for non-cacheable executions, we use a persistent finite map data structure for *depTable*, in which updates to the *depTable* result in a new copy *depTable*′. If the execution is cacheable, we update *depTable* to be *depTable*′. Otherwise, we simply discard

Algorithm 8: READ(D, x)

```
1 if (D, x) ∉ W ∧ (D, x) ∉ R then
2     R = R ∪ {(D, x)};
3     trace = currentTrace();
4     insertDepTrace(depTable, D, x, trace);
```

Algorithm 9: WRITE(D, x)

```
1 if (D, x) ∉ W then
2     W = W ∪ {(D, x)};
3     trace = currentTrace();
4     insertDepTrace(depTable, D, x, trace);
```

depTable′.

It is clear that for each pair (D, x), *depTable*(D, x) built using the above process will satisfy the Prefix Property: consider any execution *ex* with trace *ex.trace* that evaluated or updated (D, x). This access occurred at some point during the execution with current trace t, which was inserted into *depTable*. The execution continues and results in final trace *ex.trace*, which is clearly an extension of t. Hence t' is a prefix of *ex.trace* contained in *depTable*(D, x).

Efficient Dependency Sets The dependency localization algorithms utilize two operations on sets of traces stored in *depTable*: insertDepTrace and removeDepTrace. removeDepTrace(*traceSet*, t) removes from *traceSet* all traces that are extensions of t. To implement this operation efficiently, we organize *traceSet* as a trie. To illustrate the idea, Figure 5.1 depicts an example prefix set, containing three traces.

With this representation, removeDepTrace is simple: descend into the trie until the next trace action is not in the tree, in which case the algorithm simply returns, since the given trace is not the set. For example, trace *ae* is not in the trie of Figure 5.1(a); descending into the trie leads to the node reached after traversing branch labelled a, and then reaches a dead end, since no branch is labelled e. Otherwise, the trie is traversed until the input trace is exhausted. Remove the node at this location. Figure 5.1(b) illustrates the result of executing removeDepTrace on trace *ac*.

The trie structure also allows us to avoid recording a dependency twice for a single execution which evaluates accesses the same data multiple times. In particular, insertDepTrace simply descends into the tree and avoids adding an element if it would extend the trie at a leaf node (*i.e.* a trace in the set).

Overall Algorithm The overall algorithm is therefore as follows. When receiving a packet not handled by the cached rules, we evaluate f to obtain new state s' and route r with execution *ex*. We apply depChange to the set

$$W^{ex} = \{(D_1, x_1), \ldots, (D_m, x_m)\}$$

to remove all executions that are no longer cacheable in s'. In addition, if *ex* is

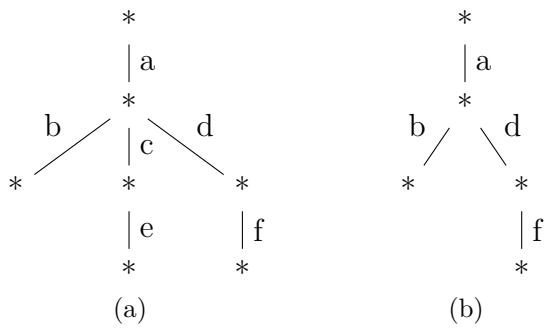


Figure 5.1: (a) Trace prefix tree containing traces ab, ace, and adf, where a, b, c, d, e, f are trace items, such as $a = read(ethDst, 1)$, $b = assert(tcpDst = 80, false)$, etc. (b) The result of executing REMOVEDEPTRACE on trace ac.

cacheable in s', we augment the trace tree with ex. We apply the rule modifications provided by the trace tree compiler to modify the switch flow tables. Otherwise the execution is not cacheable, we install no new rules, and we forward the packet through the network using packet-send commands.

If we receive a non-packet event, we evaluate g on the event in state s to obtain s'. As above, we apply DEPCHANGE to the set $\{(D_1, x_1), \ldots, (D_m, x_m)\}$ to remove all executions that are no longer cacheable in s'.

The algorithm therefore maintains correctness by ensuring that only cacheable executions are installed into the trace tree after every state transition.

5.3 Proactive Repair

Although the basic update algorithm described in the previous section effectively pinpoints cached executions that may be affected by a state change, it simply removes flow table rules that handle packets for affected cached executions. Thus, the basic algorithm relies on packet arrivals and packet-in events to generate new executions and flow table rules to handle packets locally. This may lead to many packets diverted to the controller. Furthermore, as explained in Section 5.1 above, state variable dependency analysis may produce false positives, *i.e.* executions that depend on a changed state variable, but whose results and traces are unaffected by the particular change of values.

In this section, we introduce two techniques, *speculative evaluation* and *update cancellation* which together eliminate these unnecessary changes.

5.3.1 Speculative Evaluation

Suppose some event causes the Maple program to transition from $s \to s'$ leading to a collection of executions $ex_1, \ldots, ex_n$ being identified as no longer cacheable in s'. The algorithm in Section 5.2 simply removes all of these executions from the trace tree and the corresponding rules from the switch flow tables. Some of these invalidated

executions may correspond to active flows in the network. Each of these active flows will incur a miss in the flow table, possibly causing substantial disruption to the flow and a large number of packet-in and flow mod messages exchanged between switches and controller.

Instead of simply removing rules and waiting for packet-in events, Maple could proactively evaluate f on packets that are likely to occur in the network. Given a sufficiently large and diverse sample of packets, Maple will fully expand the trace tree. If speculation can be performed quickly enough, the controller can proactively push the resulting flow rules to the switches before most packets arrive at the switch. We call this technique *speculative evaluation*.

Speculative evaluation creates two potential problems. On the one hand, speculative evaluation could lead to executions and rules being cached for packets which will never appear in the network. For example, packets could be generated with a particular pair of sender MAC address and sender IP address which will never be seen in the network. We call such packets *impossible packets*. Caching rules for impossible packets may not lead to errors, but leads to wasted resources (*e.g.* flow table rules).

A second problem is that packet arrivals may lead to state changes. For example, a particular packet could lead to a new entry added to the host location table. In a NAT controller, a packet could lead the network to open a port for communication with external hosts temporarily. If the system speculatively evaluates f with such packets, the system may make a state change which would not have occurred otherwise.

To avoid these problems, Maple only speculates using packets actually seen in the network that led to cached executions and which do not change program state. To accomplish this, whenever Maple stores a cacheable execution of f, it also stores the packet header that f was applied to. The trace tree leaves are therefore annotated with the packet header used to generate the trace.

When removing a subtree from the trace tree, the packet headers stored in the leaves of the subtree being deleted are collected into a list, and f is speculatively evaluated on these packets. Note that by storing the packets in the leaves of the tree, we naturally re-evaluate f on only packets for which we no longer have cached executions. In order to avoid changing state, we modify the tracing runtime to abort the execution if any changing write operations occur.

The effectiveness of speculation depends critically on reevaluating `f` with a sample of packets that leads to the installation of rules covering packets that will occur frequently and in the near future. Fortunately, there are important instances of state changes for which the aforementioned packet sample (one representative, but arbitrary packet per execution trace) is guaranteed to evaluate enough executions of `f` to cover the same set of traffic that was covered prior to the state change. We now describe two such cases.

For many controllers, topology data is used to calculate routes, but is not used in classifying packets into different cases. For example, access control policies are typically specified over packet attributes, rather than performing different access control checks depending on current topology. In these controllers, the equivalence classes formed by the possible traces through the program are unchanged after a topology change. Therefore, the cached packets will lead to the exploration of exactly the same

algorithm traces as prior to the change. Hence the generated rules will cover exactly the same set of traffic as the rules prior to the change.

As a second example, consider a controller using an access control list as a forwarding data structure. If an access control list entry is removed, the program will make strictly fewer distinctions among classes of traffic. Hence, speculative evaluation with cached packets will cover exactly the same set of packets as before the access control list change.

Speculative evaluation transforms Maple's reactive approach into a mostly proactive approach: Maple is only reactive during a warmup period immediately after initialization. After running for some time, virtually all configuration is proactive, since the trace tree will typically include cached executions and packets for flows occurring in the network.

5.3.2 Update Minimization

Suppose a Maple system is in state s with flow tables F. When a state change occurs, the Maple algorithm with speculative evaluation will produce a sequence of updates,

$$U_1, \ldots, U_n, U'_1, \ldots, U'_m$$

where U_i are changes due to removal of non-cacheable traces from the trace tree and U'_j are changes due to the addition of newly cacheable traces resulting from speculative evaluations. The final result of applying these state changes is a new collection of flow tables F'. In the case where most flows are unaffected, F' will be very similar to F, even though there may be many changes contained in
$U_1, \ldots, U_n, U'_1, \ldots, U'_m$. Naively executing each of these flow table changes to move the system to flow tables F' may become a substantial bottleneck, since currently available hardware OpenFlow switches process fewer than 1000 flow table updates per second [57].

Rather than actually apply all the changes specified by the updates, Maple applies a technique called *update minimization* to execute the minimum number of updates required to move update flow tables from F to F'. In particular, let F consist of rules

$$r_i = (p_i, m_i, a_i) \text{ , for } i \in I$$

where p_i is rule priority, m_i is the rule match condition, a_i is the rule action. Similarly, let F' be

$$r'_j = (p'_j, m'_j, a'_j) \text{ , for } j \in J.$$

Then the minimal collection of updates is as follows:

$$\text{delete } r_i \text{ , when } \neg\exists j : p_i = p'_j \wedge m_i = m'_j$$

$$\text{insert } r'_j \text{ , when } \neg\exists i : p_i = p'_j \wedge m_i = m'_j$$

$$\text{modify } r_i \text{ to action } a'_j \text{ , when } \exists j : p_i = p'_j \wedge m_i = m'_j \wedge a_i \neq a_j$$

This can be seen to be a minimal update since any updates that result in a rule

with a priority and match in that were not used in F' must accomplish this through at least one insert. Likewise, a rule in F such that no rule in F' has the same priority and match must have been removed through at least one deletion. Rules with same priority and match but different action require at least one modification or at least one deletion and one insertion.

Maple calculates the above minimum update using a linear $\mathcal{O}(\sum_i |U_i| + \sum_j |U'_j|)$ algorithm. To specify the algorithm, we first define some notation. Each update U_i and U'_j consists of a sequence of changes c where each change c has two components $c.type$, which is one of the update type constants *Insert*, *Delete*, *Modify*, and $c.rule$ which is an OpenFlow rule. Note that the updates produced by the trace tree operations (remove trace and augment) generates a sequence of inserts and deletions (*i.e.* no modifications) with no consecutive inserts (or deletes) of rules with the same priority and match. We let c_k be the sequence that results from concatenating the changes of $U_1, \ldots, U_n, U'_1, \ldots, U'_m$ in order. The CANCELUPDATES algorithm is shown in Figure 10. It computes a dictionary, z, mapping pairs (p, m) of priority p and match m to a triple (t, a, a_0) where t is the update type, a which is the action to use in the final modify or update command and a_0 is the action that the rule had in the initial rule set F in the case that there was a rule with priority p and match m in F. The only rules that need to be updated are the ones with entries in z. The algorithm makes a single pass over c_k in order. For each (p, m) it processes alternating insertions and deletions (if no rule with (p, m) is in F) or alternating deletions and insertions (if a rule with (p, m) is in F). If it encounters a change with (p, m) not in z, then it adds the change to z. In addition, if the type is *Delete*, then we must be deleting a rule existing in F (since, given the meaning of z, no change has been made to a (p, m) rule yet), so the action is the action of the rule in F and we save that in the triple added to z. Otherwise, we have already saved a change to the rule with (p, m) and we update the change as appropriate. For example, if c inserts a rule that was previously deleted and the inserted rule has an action that differs from the initial action in F, then we use a modification. On the other hand, if the inserted rule has the same action as the initial action in F, we delete the change, since we need to make no changes to F in this case. The other cases follow from similar reasoning.

5.4 Evaluations

5.4.1 Evaluation Methodology

Network topology: We evaluate controllers using four topologies, named "Linear", "Square", "Internet2" and "FatTree". The Linear topology consists of three real HP 5406 switches. The Square topology is a small cyclic topology with 4 switches. The Internet2 topology is based on the topology of Internet2 [27] exchanges and includes 14 routers. For Linear, Square and Internet2 we locate one host at each switch. The FatTree topology is a Fat Tree [3] topology with $k = 4$, consisting of 20 switches and two hosts per edge switch.

Controllers: We compare the performance of Maple with Pyretic [45] and Flood-

Algorithm 10: cancelUpdates($\{c_k\}$)

```
1  z = empty dictionary keyed on priority-match pair (p, m)
2  for c ∈ {c_k} do
3      let t = c.type
4      let p = c.rule.priority
5      let m = c.rule.match
6      let a = c.rule.action
7      if (p, m) ∉ z then
8          if t = Insert then
9              z[(p, m)] = (t, a, ⊥)
10         else
11             z[(p, m)] = (t, a, a)
12     else
13         let (t', a', a_0) = z[(p, m)]
14         if t' = Delete ∧ t = Insert ∧ a = a_0 then
15             delete (p, m) from z
16         else if t' = Delete ∧ t = Insert ∧ a ≠ a_0 then
17             z[(p, m)] = (Modify, a, a_0)
18         else if t' = Insert ∧ t = Delete then
19             delete (p, m) from z
20         else if t' = Modify ∧ t = Delete then
21             delete (p, m) from z
22         else
23             error: impossible
24 return z
```

light [20]. In the case of Pyretic, we evaluate a so-called learning switch controller included in the Pyretic distribution, which forwards packets over a spanning tree. The Floodlight controller is evaluated in its default configuration, which uses the "Forwarding" application module to forward along shortest paths between any sender and receiver. In addition, we compare Maple with a version of Maple that uses a naive strategy of completely removing all cached decisions (and corresponding flow entries) on every state change. In the following evaluations, we use the label "Naive" to denote controllers using this naive strategy.

Experimental Setup In the first two experiments, we initialize the system by executing a ping from every host to every other host, thereby forcing the controller to learn about every host in the network and to configure paths for all pairs of hosts. We then execute a particular state change, followed by pinging between all pairs of hosts a second time. Since some controllers (notably Naive and Floodlight) rely mostly on a reactive approach, the successful all-to-all pings ensures that the new network

configuration is in place. We run all controllers on a BSD system with 2.9 GHz Intel Core i7 processors and 8GB DDR3 memory. Pyretic is run with Python 2.7.5 and Floodlight is run with Java 1.6 using the Java HotSpot 64-bit Server VM. Maple is compiled with GHC 7.6.3. All switches run OpenFlow 1.0. Simulated switches and hosts execute in a Linux virtual machine running Mininet 2.0.

5.4.2 Access Control Policy Changes

In this experiment, we evaluate the effectiveness of dependency tables, speculation, and update cancellation to support user-defined, northbound state changes. In particular, we implement and evaluate the bad host control program presented in Chapter 3 using Maple, Maple without speculation, and with the Naive strategy. We simulate the Square, Internet2, and FatTree topologies in Mininet [30]. After initialization, we add a single, particular host to the bad host set.

Figure 5.2 shows the number of flow modifications made by three controllers in order to repair the forwarding state to support all-to-all traffic after the access control policy change. In particular, we evaluate Maple, Maple without speculative execution (labelled "Maple-"), and Maple with the Naive strategy. We see that dependency localization, with or without speculation, dramatically reduces the number of modifications required. In particular, Maple without speculation achieves improvements ranging from 3x to 12x over the Naive controller. Speculation provides a further small improvement ranging from factors of 1.2x to 1.4x. Without speculation, rules for the new bad host are removed with flow deletion commands; later, the second ping leads to flow insert commands at the edge switches to drop the host's packets. With speculation, the delete and insert are combined into a single modification and hence speculation leads to a reduction of $(H-1)*2$ insertions, where H is the number of hosts in the network.

5.4.3 Link Failure Recovery

We now evaluate the effectiveness of speculative evaluation to recognize results that are invariant under a particular state change despite having a dependence on a changed component. In particular, we evaluate Maple, Naive, and Floodlight controllers that forward along shortest paths and a Pyretic controller that forwards along a spanning tree. After initializing the system with all-to-all ping traffic, we remove one link from the network. We then measure both the number of flow modification commands and the time needed to complete pings from every host to every other host. For the case of Floodlight, we take two measurements. We do this because Floodlight updates rules only when they time out due to being idle for at least 5 seconds. With this strategy, an incorrect rule (i.e. one forwarding to a failed port) may remain in the switch flow table indefinitely, if packets continue to arrive to trigger the rule. In fact, the evaluation in this experiment triggers this behavior and measurements triggering this behavior are labelled "Floodlight-I". We also include measurements of Floodlight when this behavior is not triggered; these are labelled "Floodlight".

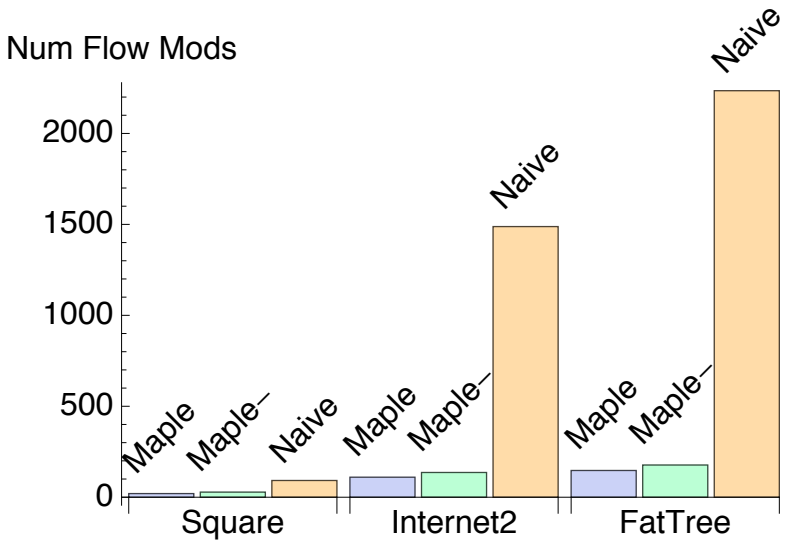


Figure 5.2: Number of modifications needed by Maple with and without speculation (labelled "Maple-") and Naive after a bad host is identified in three topologies.

Figure 5.3 shows that in all topologies, Maple's speculative execution provides a substantial improvement over all other controllers. In particular, the mean time to complete an all-to-all ping after a link failure for Maple is 189 msec, 1.02 sec, and 1.35 sec, for the three topologies respectively. Naive, in contrast requires from 6x to 8x longer to restore connectivity, Floodlight requires up to 87x longer when rule idle timeouts fail to trigger and from 2.5x to 6x longer when rule idle timeouts remove rules before the link down event. Pyretic requires 87x longer to restore connectivity and illustrates the disadvantage of applying a complex compilation process on each state change. Pyretic results for larger topologies are omitted as the system failed to restore connectivity within a reasonable (lengthy) time frame.

Figure 5.4 shows the number of flow modifications performed by the Maple, Naive and Pyretic controllers to restore connectivity. In particular, Maple requires 54, 146, and 559 flow modifications in the three topologies, respectively. In contrast, Pyretic performs over 7500 modifications for the same event. Figure 5.5 shows the factor improvement of Maple over Naive for the three topologies. In particular, it shows that Maple uses between 2x to 12x fewer rules when compared with the same controller using the Naive strategy.

5.4.4 Idempotence Detection

In this section, we evaluate the impact of idempotence detection for state-changing forwarding functions in a real network consisting of HP 5406 switches connected in the Linear topology. We use a Maple controller which updates a host location table on every received packet. In this case, the forwarding function `f` performs potentially state-changing write operations on every execution. We evaluate the Maple controller using two implementations: in RW, we use the read-write analysis described in Sec-

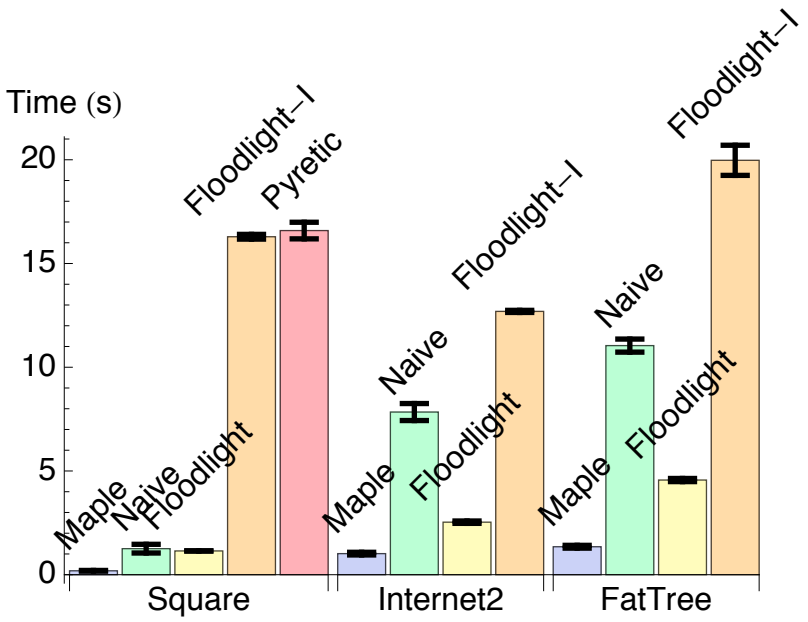


Figure 5.3: Amount of time needed by Maple, Naive, Floodlight, and Pyretic to restore all-to-all connectivity after a single link failure in three topologies.

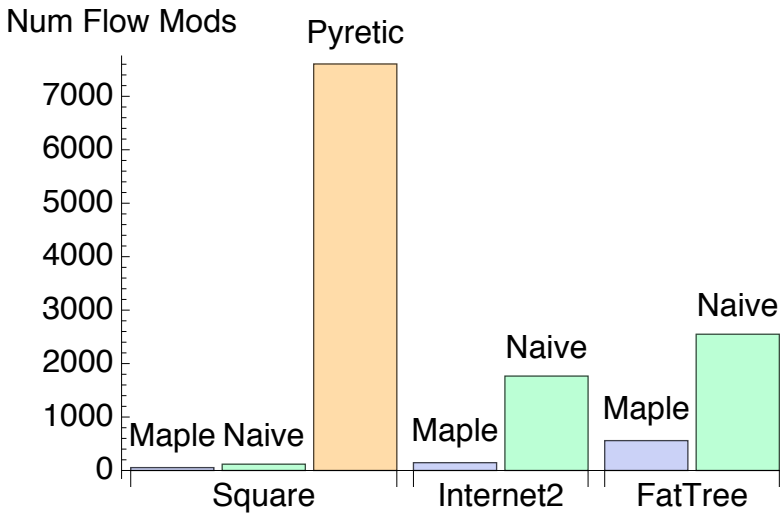


Figure 5.4: Number of flow modifications needed by Maple, Naive, and Pyretic to restore all-to-all connectivity after a single link failure in three topologies.

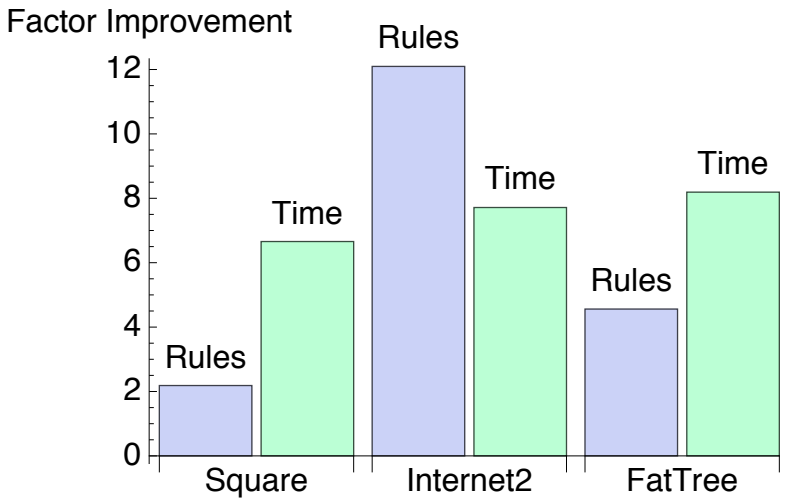


Figure 5.5: Number of modifications and time needed by the Naive controller to restore all-to-all connectivity after a single link failure in three topologies expressed as a factor of the number of modifications and time needed by Maple to handle the same event.

tion 5.2; in RW- we only cache executions which leave the state unchanged after execution. We measure the time for a client on one side of the linear network to retrieve a web page from a server at the other side of the network. Figure 5.6 shows the distribution of the request times in both systems. We see that detecting cacheability of state-changing executions can have a substantial impact: the median time to request a web page is reduced by 2x in this network. The difference is due to the number of packets which are diverted to the controller: the mean number of packets diverted to the controller per client request for RW and RW- is 2.5 and 6 packets, respectively.

5.5 Summary

This chapter has presented the methods used by Maple to efficiently update distributed flow tables under network and policy dynamics, including state updates executed as a result of the execution of an algorithmic policy on packets arriving in the network. Evaluations demonstrate that dependency localization, speculative evaluation, and update minimization allow Maple to adapt to changes quickly. In particular, Maple reduces the link failure recovery time in realistic network topologies by a factor of 2.5x to 6x when compared with a popular open-source OpenFlow network controller.

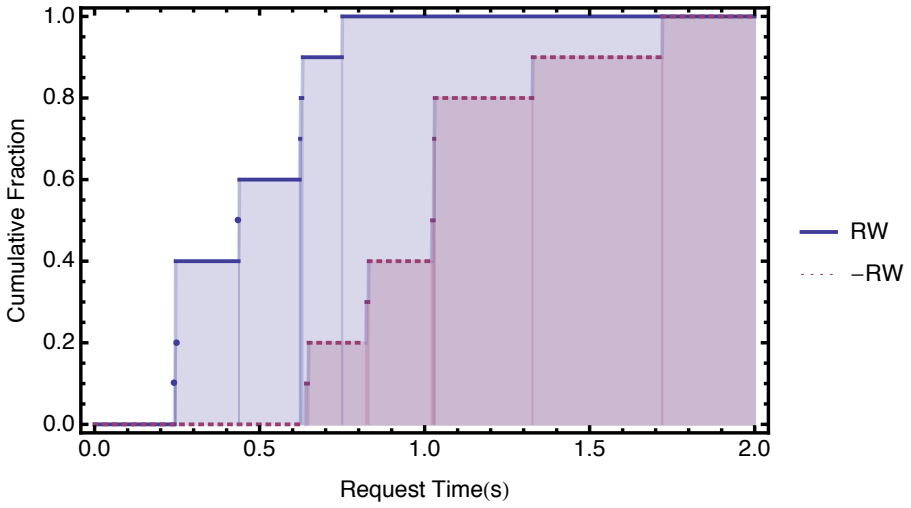


Figure 5.6: Comparison of the cumulative distributions of the times to retrieve a web page from an HTTP server with and without idempotence detection.

Chapter 6

Applications of Maple

Maple has been implemented in Haskell and Java, supporting algorithmic policies expressed in either language. In this chapter, we demonstrate several example programs written using the Java version of Maple. For each program, we describe the source code and show the rules generated by Maple. These examples serve to illustrate the breadth of network functions that can be implemented using Maple.

6.1 Spanning Tree Broadcast

We first consider a very simple controller that broadcasts every packet from the sender's ingress location (switch and port) to all edge ports. We consider a switch and port to be an edge port if it is an active port and the port is not participating in linking two OpenFlow switch ports together. Listing 6.1 shows the algorithmic policy. On line 1, we declare the class `Broadcast` to extend `MapleFunction`, which all algorithmic policies in Maple for Java must do. On line 2, the `Broadcast` class overrides the abstract `Route onPacket(Packet)` function to define the forwarding function of the network. On line 3, the forwarding function obtains the sender's location, by applying the built-in `hostLocation()` state function to the builtin `ethSrc()` packet attribute access function. On line 4, the function constructs a graph based on the network topology, obtained through the calls to the built-in `switches()` and `links()` state functions. The resulting graph `gr` is an instance of the user-defined (*i.e.* not built-in) `Digraph` data structure, a plain-old-Java object, which provides several convenient graph and routing algorithms that we will use throughout this chapter. On line 6, the function returns a multicast route from the given source location, with the computed minimum spanning tree and with the packets delivered to all ports in the network. Note that Maple automatically removes the sender port from the collection of egress ports, so it is safe to include all edge ports in the list of egress ports to the multicast route return value.

We use Mininet [30] to simulate a network with the topology shown in Figure 6.1. This network consists of four switches, labelled s_1 through s_4 and five hosts, labelled h_1 through h_5. Links between switches and between switches and hosts are shown. Labels on a link indicate the port numbers at which the link attaches to the given

Listing 6.1: Broadcast along spanning tree to all edge ports.

```
1 public class Broadcast extends MapleFunction {
2   public Route onPacket(Packet p) {
3     SwitchPort srcLoc = hostLocation(p.ethSrc());
4     Digraph gr = Digraph.unitDigraph(switches(), links());
5     return Route.multicast(srcLoc, gr.minSpanningTree(), edgePorts());
6   }
7 }
```

switches. If a link connects two switch ports both having the same number, the link is labelled with a single port number at its midpoint. For other links, each end of the link that attaches to a switch is labelled with a port number.

After pinging from h_1 to h_4, Maple generates and installs the following rules in the network[1]:

Switch	Inport	Prio	Match	Outports
1	1	0	`srcEth=4`	[2,3]
1	3	0	`srcEth=1`	[1,2]
2	1	0	`srcEth=1`	[3,4]
2	3	0	`srcEth=4`	[1,4]
3	1	0	`srcEth=1`	[4]
3	1	0	`srcEth=4`	[4]
4	1	0	`srcEth=1`	[3,4]
4	3	0	`srcEth=4`	[1,4]

Since the ping causes both hosts (h_1 and h_4) to send traffic to each other, the rules include spanning trees from h_1 to all edge ports and from h_4 to all edge ports. We can understand the rules by following the processing path starting from h_1 or h_4. From the topology figure, we see that h_1 is located at port 3 of switch s_1. Hence, a packet sent by h_1 will first be processed by the second rule in the above table and will be sent to ports 1 and 2 of s_1, reaching both switches s_2 and s_3 at ports 1 and 1 respectively. At s_2 the packet is processed by the third rule, which sends the packet to ports 3 and 4, reaching host h_2, which ignores the packet, and switch s_4 at port 1. At s_4, the packet is processed by the penultimate rule in the table above and forwarded to ports 3 and 4, reaching h_4, the intended recipient, and h_5 which ignores the packet. Meanwhile, the packet reaches s_3 and is forwarded with the fifth rule, sending to port 4, where h_3 is located. We can similarly trace the path for packets sent by h_4.

Note also that the rules only match on the source Ethernet address. Since the rules are all disjoint, they all share the same priority value.

1. In all flow tables in this chapter, the Ethernet addresses occurring in match conditions are written in decimal notation, rather than the traditional format consisting of 6 pairs of hexadecimal characters. Since the examples in this chapter use sequentially numbered mac addresses starting at 1, this format allows for tables to be formatted more compactly.

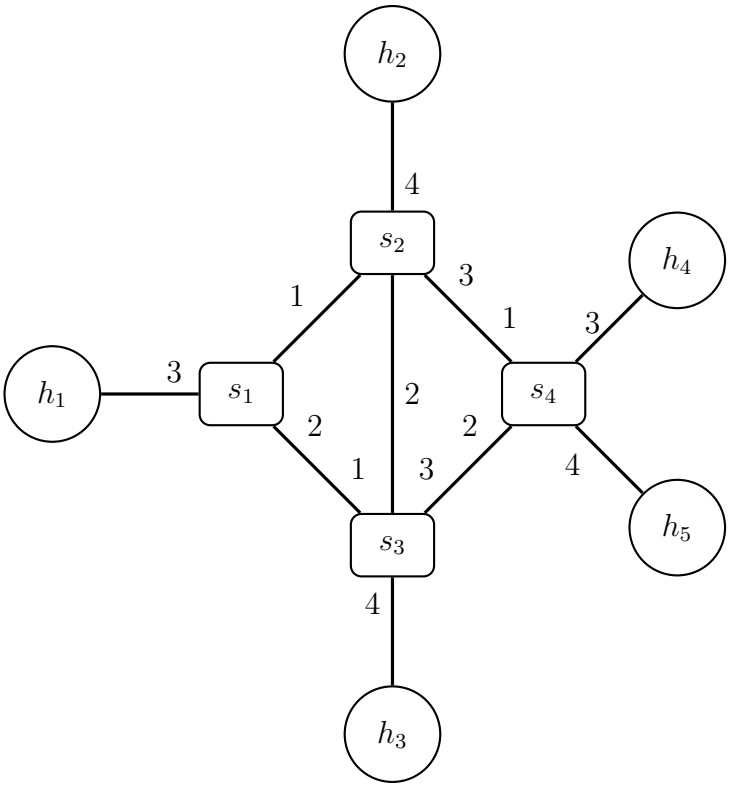


Figure 6.1: Example network used throughout Chapter 6.

6.2 Shortest Paths

The previous `Broadcast` policy uses link bandwidth inefficiently, since it uses bandwidth on links which are not used to deliver traffic from the sender to the destination. For example, packets from h_1 to h_4 were forwarded along the link between s_1 to s_3, even though s_3 was not used in forwarding the packet to the recipient h_4. Moreover, the path through a minimum spanning tree from one host to another host may not be the shortest path, leading to further unnecessary link usage.

We can improve on the `Broadcast` policy by forwarding packets along the shortest path between sender and receiver. In particular, the `SP` policy in Listing 6.2 forwards packets along shortest paths between sender and receiver. In line 13, the policy calculates the shortest path between two switches the `Digraph.symmetricShortestPath`, a user-defined method in `Digraph` class. In line 14, the policy returns the overall unicast path starting at the sender, ending at the destination and traversing the calculated shortest path.

One complication in this program is that the packet may not be addressed to a host with a known location. For example, the packet may be sent to the Ethernet broadcast address, which does not correspond to a host at a known location. Alternatively, the packet may be addressed to a particular host, but that destination host may not have sent any packets yet, causing the network controller to have no knowledge of the destination's location. To handle this case, the policy checks (line 8) whether the

Listing 6.2: Forward along shortest path between source and destination when destination location is known.

```
1  public class SP extends MapleFunction {
2    public Route onPacket(Packet p) {
3
4      SwitchPort srcLoc = hostLocation(p.ethSrc());
5      SwitchPort dstLoc = hostLocation(p.ethDst());
6      Digraph gr = Digraph.unitDigraph(switches(), links());
7
8      if (null == dstLoc) {
9        // Destination location unknown, so broadcast everywhere.
10       return Route.multicast(srcLoc, gr.minSpanningTree(), edgePorts());
11     } else {
12       // Destination location known, so use shortest path.
13       List<Link> path = gr.symmetricShortestPath(srcLoc.getSwitch(), dstLoc.getSwitch());
14       return Route.unicast(srcLoc, dstLoc, path);
15     }
16   }
17 }
```

destination location is null, indicating that the location is not known. In this case, it broadcasts the packet along a minimum spanning tree rooted at the sender, as in the `Broadcast` policy.

After pinging from h_1 to h_4, Maple has generated the following flow rules:

Switch	Inport	Prio	Match	Outports
1	1	0	`srcEth=4,dstEth=1`	[3]
1	3	0	`srcEth=1,dstEth=4`	[1]
2	1	0	`srcEth=1,dstEth=4`	[3]
2	3	0	`srcEth=4,dstEth=1`	[1]
4	1	0	`srcEth=1,dstEth=4`	[3]
4	3	0	`srcEth=4,dstEth=1`	[1]

We can observe that the match conditions now match on both sender and receiver addresses. We also observe that shortest paths are used: for example, a packet sent by h_1 to h_4 will enter s_1, be forwarded over port 1 to s_2, where it is forwarded on port 3 to s_4, which forwards the packet to the receiver at port 3.

We note that Maple automatically handles network dynamics to keep the forwarding table in sync with the `SP` policy. For example, if, in the current state, we remove the link between s_2 and s_4, as depicted in Figure 6.2, the forwarding tables are automatically updated to send the packets from h_1 to h_4 over the path s_1, s_3, s_4, which is now the shortest path between this sender and receiver. The updated forwarding tables generated by Maple are:

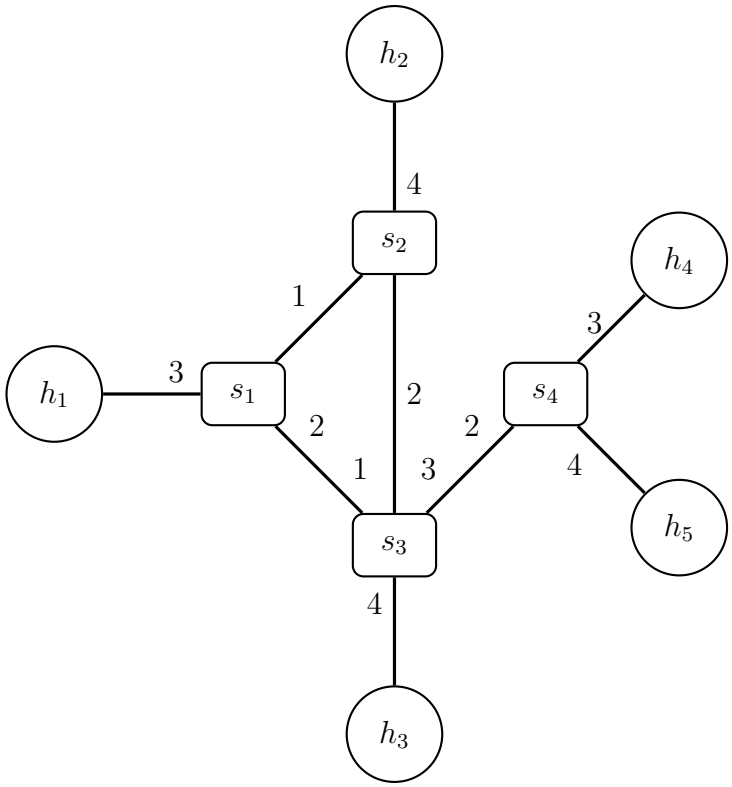


Figure 6.2: Example network after failure of the link between s_2 and s_4.

Switch	Inport	Prio	Match	Outports
1	2	0	`srcEth=4,dstEth=1`	[3]
1	3	0	`srcEth=1,dstEth=4`	[2]
3	1	0	`srcEth=1,dstEth=4`	[3]
3	3	0	`srcEth=4,dstEth=1`	[1]
4	2	0	`srcEth=1,dstEth=4`	[3]
4	3	0	`srcEth=4,dstEth=1`	[2]

6.3 Bad Hosts

Many network controllers — unlike the `Broadcast` and `SP` policies — provide parameters that can be adjusted at runtime by a network administrator. In this section, we illustrate the use of forwarding data structures to provide a very simple access control policy that administrators can change at runtime. In particular, we implement the bad hosts policy described in Chapter 3. This policy performs shortest path routing between all hosts, but denies any traffic sent from or to so-called "bad hosts".

Listing 6.3 specifies the bad host controller, named `BadHostSP`. Line 3 declares the `badHosts` set as a set of `Long` values. Lines 5-7 specify the constructor of the class, which will be called upon system initialization. In line 6, the constructor initializes the `badHosts` set, which is initially empty. In line 7, the command `externallyVisible(badHosts, ‘‘BadHosts’’, Long.class);` is executed, which

causes the built-in REST HTTP server to allow external processes to issue read and write commands, expressed as JSON objects, to the server. In these commands the string identifier "BadHosts" will be used to refer to the set and the `Long.class` class is used to convert to and from JSON objects representing elements of the set. Finally, the forwarding function checks, in lines 11-13, whether either the sender or receiver is in the `badHosts` set; if so, the packet is dropped, using the `Route.null` return value and otherwise, the packet is forwarded normally.

Maple includes an http server that provides a resource with URI "DB" that accepts state-changing commands encoded in JSON objects submitted in the body of PUT requests. In this example, the declaration that `badHosts` is externally editable with identifier "BadHosts", allows external processes to insert host `4` by executing a PUT to the Maple server's "DB" resource with a body containing the following JSON object:

```
1 { component: 'BadHosts', method:'insert', value: 4 }
```

Similarly, host `4` can be deleted from the `badHosts` set by executing a PUT to "DB" with the following body:

```
1 { component: 'BadHosts', method:'delete', value: 4 }
```

This API also allows further programs to be constructed above Maple. In particular, we use this API to implement both CLI and GUIs that allow administrators to easily edit the runtime parameters of the system.

Using `BadHostSP` produces the same table as `SP` after h_1 pings h_4. However, If we then add host 4 to the set, Maple automatically updates the flow tables to be:

Switch	Inport	Prio	Match	Outports
1	3	0	`srcEth=1,dstEth=4`	[]
4	3	0	`srcEth=4`	[]

The old forwarding actions have been deleted and new rules have been installed that drop traffic (due to the outgoing ports list being empty) from or to h_4 at the egress point. Note that many rules that were previously installed at various switches in the network have been removed, since they are no longer needed by any network flows.

6.4 Access Control Lists

Maple can also support complex, declarative policies, such as access control lists (ACLs), which are frequently used by network administrators today to implement network security policies. An access control list is a sequence (i.e. ordered list) of access control rules. Typically each access control rule specifies a subset of IP traffic which it applies to, by specifying ranges for the source and destination IP addresses and the source and destination TCP or UDP ports. In addition, typically each rule also specifies whether packets matching that criterion should be permitted or denied. The semantics of the entire ACL is very simple to express as an algorithm: check each entry of the ACL, in order, until the first matching rule is found, returning the action

Listing 6.3: Bad host access control combined with shortest path routing.

```
1  public class BadHostSP extends MapleFunction {
2
3    MapleSet<Long> badHosts;
4
5    public BadHostSP() {
6      badHosts = newSet();
7      externallyVisible(badHosts, ''BadHosts'', Long.class);
8    }
9
10   public Route onPacket(Packet p) {
11     if (badHosts.contains(p.ethSrc()) || badHosts.contains(p.ethDst())) {
12       return Route.null;
13     }
14     SwitchPort srcLoc = hostLocation(p.ethSrc());
15     SwitchPort dstLoc = hostLocation(p.ethDst());
16     Digraph gr = Digraph.unitDigraph(switches(), links());
17
18     if (null == dstLoc) {
19       // Destination location unknown, so broadcast everywhere.
20       return Route.multicast(srcLoc, gr.minSpanningTree(), edgePorts());
21     } else {
22       // Destination location known, so use shortest path.
23       List<Link> path = gr.symmetricShortestPath(srcLoc.getSwitch(), dstLoc.getSwitch());
24       return Route.unicast(srcLoc, dstLoc, path);
25     }
26   }
27 }
```

```
103.14.202.2/31   40.207.141.55/32 UDP    0 : 0       23 : 23    deny
103.14.202.2/32   40.207.141.55/32 UDP    0 : 65535   23 : 23    deny
103.14.200.221/32 0.0.0.0/1        TCP  123 : 123     22 : 22    deny
103.14.202.2/32  40.207.141.48/31  TCP 1024 : 65535 1024 : 65535 deny
```

Figure 6.3: Example access control list.

in that rule. Typically there is also a default action, in case no rules apply. Since the semantics of an ACL can be naturally expressed as a functional computation of a packet, we can easily implement ACLs in Maple as well, by executing the algorithm, using Maple's packet access API to test the rules against the packet.

We now demonstrate how to implement a simple form of ACLs in Maple. In particular, we assume that each ACL rule consists of the following data items: two IPv4 prefixes, one each for source IP address and for destination IP address, a transport type which is either UDP or TCP, and two transport port ranges, one range each for source and destination port, where each range is expressed by a starting port and ending port, and finally an action that specifies whether the packet should be allowed or denied. We will store our ACL in a file. Figure 6.3 shows a concrete example of a file containing an ACL.

We begin the implementation by specifying a class of objects, called `ACRule`, such that each object represents a single ACL rule. Listing 6.4 shows the instance fields of the `ACRule` class, which provide a straightforward representation the data required to represent a single ACL rule. The class should provide an instance method that evaluates the rule against the packet and returns true if the packet matches the rule, false otherwise. Concretely, this method should have the following signature:

```
1 public boolean matches(Packet p);
```

This method would be easy to implement, if transport port ranges were not included in ACL rules. For example, setting aside port ranges for the moment, we could implement the function as follows:

```
1  public boolean matches(Packet p) {
2      // Decide which condition to assert for protocol type.
3      Assertion protAssertion;
4      protAssertion = protocol == Protocol.TCP ? Assertion.isTCP() : Assertion.isUDP();
5
6      // Form the overall condition to assert.
7      Assertion ruleAssertion;
8      ruleAssertion = Assertion.all( Assertion.ipSrcIn(srcIp,srcIpLen),
9                                     Assertion.ipDstIn(dstIp,dstIpLen)
10                                    protAssertion);
11
12     return p.satisfies(ruleAssertion);
13 }
```

Listing 6.4: ACRule instance fields

```
1  public class ACRule {
2
3    public enum Protocol { TCP, UDP }
4    public enum Action { DENY, PERMIT };
5
6    int srcIp;
7    int srcIpLen;
8    int dstIp;
9    int dstIpLen;
10   Protocol protocol;
11   int srcPortBegin;
12   int srcPortEnd;
13   int dstPortBegin;
14   int dstPortEnd;
15
16   Action action;
17   ...
18 }
```

However, the port ranges pose a complication. OpenFlow does not support port ranges, and Maple does not provides assertions on port ranges either.[2] Hence, for a general port range, our program must read the transport port and check whether it is in the given range. However, two special cases exist: when the port range contains all ports and when it contains a single port. In the former case, we can simply omit the assertion, and in the latter case we can use a Maple assertion on a single port value. Therefore, our strategy will be to use assertions on port ranges in these two special cases and otherwise simply read the appropriate transport field and test against the range in the rule.

Listing 6.5 implements this strategy. In line 9, it invokes the `sourcePortAssertion` method to calculate an appropriate assertion on the transport source port, if one exists. If no assertion can be made for the rule (i.e. it does not fall into one of the two special cases mentioned above), then `null` is returned. In line 10, the `canAssertSrcPort` boolean is set if the source port assertion is non-null. Lines 14 and 15 perform the analogous computation for the destination port. In lines 18-23, the overall assertion is constructed. Note that if the transport fields can be asserted, then the above-calculated assertion is used, otherwise a constantly true assertion is used (`Assertion.anyPacket()`). In line 24, the assertion is evaluated. If true, lines 25-28 continue to check any remaining conditions on transport ports that could not be asserted. Otherwise, the packet failed the assertion and false is returned. Listing 6.6 shows the definitions of the helper functions used in `matches`.

Given the above, we can now easily write a function to execute an ACL against

2. Assertions on port ranges may be added in future versions of Maple.

Listing 6.5: Match ACL rule against a packet.

```
1  public boolean matches(Packet p) {
2
3    // Decide which condition to assert for protocol type.
4    Assertion protAssertion;
5    protAssertion = protocol == Protocol.TCP ? Assertion.isTCP() : Assertion.isUDP();
6
7    // decide whether we can use an assertion for the dest port
8    // range and if so what the assertion is.
9    Assertion srcPortAssertion = sourcePortAssertion();
10   boolean canAssertSrcPort = (null != srcPortAssertion);
11
12   // As above, decide whether we can use an assertion for the dest port
13   // range and if so what the assertion is.
14   Assertion dstPortAssertion = destinationPortAssertion();
15   boolean canAssertDstPort = (null != dstPortAssertion);
16
17   // Form the overall condition that can be asserted.
18   Assertion ruleAssertion =
19       Assertion.all(Assertion.ipSrcIn(srcIp,srcIpLen),
20                     Assertion.ipDstIn(dstIp, dstIpLen),
21                     protAssertion,
22                     canAssertSrcPort ? srcPortAssertion : Assertion.anyPacket(),
23                     canAssertDstPort ? dstPortAssertion : Assertion.anyPacket());
24   if (p.satisfies(ruleAssertion)) {
25     if (canAssertSrcPort && canAssertDstPort) return true;
26     if (canAssertSrcPort && !canAssertDstPort) return checkDstPort(p);
27     if (canAssertDstPort) return checkSrcPort(p);
28     return (checkSrcPort(p) && checkDstPort(p));
29   } else {
30     return false;
31   }
32 }
```

Listing 6.6: Helper functions for match method in ACLRule.

```
1  private Assertion sourcePortAssertion() {
2    if (srcPortBegin == srcPortEnd) {
3      if (protocol == Protocol.TCP)
4        return Assertion.tcpSrcEquals(srcPortBegin);
5      else
6        return Assertion.udpSrcEquals(srcPortBegin);
7    } else if (srcPortBegin == 0 && srcPortEnd == 0xffff) {
8      return Assertion.anyPacket();
9    } else
10     return null;
11 }
12 private Assertion destinationPortAssertion() {
13   if (dstPortBegin == dstPortEnd) {
14     if (protocol == Protocol.TCP)
15       return Assertion.tcpDstEquals(dstPortBegin);
16     else
17       return Assertion.udpDstEquals(dstPortBegin);
18   } else if (dstPortBegin == 0 && dstPortEnd == 0xffff) {
19     return Assertion.anyPacket();
20   } else
21     return null;
22 }
23 private boolean checkDstPort(Packet p) {
24   int dstPort = (protocol == Protocol.TCP) ? p.tcpDst() : p.udpDst();
25   return (dstPortBegin <= dstPort && dstPort <= dstPortEnd);
26 }
27 private boolean checkSrcPort(Packet p) {
28   int srcPort = (protocol == Protocol.TCP) ? p.tcpSrc() : p.udpSrc();
29   return (srcPortBegin <= srcPort && srcPort <= srcPortEnd);
30 }
```

Listing 6.7: Match an ACL against a packet.

```
1  ACRule.Action matches(LinkedList<ACRule> acl, Packet p) {
2    for (ACRule rule : acl)
3      if (rule.matches(p)) return rule.action;
4    return ACRule.DENY;
5  }
```

a packet, as shown in Listing 6.7. Finally, assuming that we also have a function `parseACLFromFile` to parse an ACL from a file (which we omit here), we can now implement an algorithmic policy that uses an ACL to implement access control. The policy, named `ACLSP` is shown in Listing 6.8. The constructor (lines 5-12) loads the ACL from a particular file and catches any error that occur. The forwarding function tests the ACL in line 16. If the ACL returns action `DENY` against the particular packet, the packet is dropped; otherwise the packet is processed normally.

If we run Maple with the `ACLSP` policy loading the ACL shown in Figure 6.3 and then ping from h_1 to h_4, as before, we now find that Maple generates the following flow tables:

Sw	Inp	Prio	Match	Out
1	1	0	`srcEth=4,dstEth=1`	[3]
1	3	2	`srcEth=1,ethtyp=2048,prot=17,sip=103.14.202.2/31,` `dip=40.207.141.55/32,stp=0,dtp=23`	?
1	3	1	`srcEth=1,ethtyp=2048,prot=6,sip=103.14.200.221/32,` `dip=0.0.0.0/1,stp=123,dtp=22`	?
1	3	1	`srcEth=1,ethtyp=2048,prot=6,sip=103.14.202.2/32,` `dip=40.207.141.48/31`	?
1	3	1	`srcEth=1,ethtyp=2048,prot=17,sip=103.14.202.2/32,` `dip=40.207.141.55/32,dtp=23`	?
1	3	0	`srcEth=1,dstEth=4`	[1]
2	1	0	`srcEth=1,dstEth=4`	[3]
2	3	0	`srcEth=4,dstEth=1`	[1]
4	1	0	`srcEth=1,dstEth=4`	[3]
4	3	2	`srcEth=4,ethtyp=2048,prot=17,sip=103.14.202.2/31,` `dip=40.207.141.55/32,stp=0,dtp=23`	?
4	3	1	`srcEth=4,ethtyp=2048,prot=6,sip=103.14.200.221/32,` `dip=0.0.0.0/1,stp=123,dtp=22`	?
4	3	1	`srcEth=4,ethtyp=2048,prot=6,sip=103.14.202.2/32,` `dip=40.207.141.48/31`	?
4	3	1	`srcEth=4,ethtyp=2048,prot=17,sip=103.14.202.2/32,` `dip=40.207.141.55/32,dtp=23`	?
4	3	0	`srcEth=4,dstEth=1`	[1]

We see that more complex rules matching on IP prefixes have been added to the flow tables. These access control rules all have action `?`, which means that Maple has not determined the action which should be taken for packets of satisfying the given condition. In this case, the flow rule will forward packets back to the controller for

Listing 6.8: Shortest path routing with ACL loaded from file.

```
1  public class ACLSP extends MapleFunction {
2
3    LinkedList<ACRule> acl;
4
5    public ACLSP() {
6      try {
7        acl =parseACLFromFile(new File("acl1"));
8        System.out.println("OK");
9      } catch (FileNotFoundException e) {
10       System.out.println("Failed: " + e.toString());
11     }
12   }
13
14   public Route onPacket(Packet p) {
15
16     if (ACRule.DENY == matches(acl,p)) return Route.null;
17
18     SwitchPort srcLoc = hostLocation(p.ethSrc());
19     SwitchPort dstLoc = hostLocation(p.ethDst());
20     Digraph gr = Digraph.unitDigraph(switches(), links());
21
22     if (null == dstLoc) {
23       // Destination location unknown, so broadcast everywhere.
24       return Route.multicast(srcLoc, gr.minSpanningTree(), edgePorts());
25     } else {
26       // Destination location known, so use shortest path.
27       List<Link> path = gr.symmetricShortestPath(srcLoc.getSwitch(), dstLoc.getSwitch());
28       return Route.unicast(srcLoc, dstLoc, path);
29     }
30   }
31 }
```

further evaluation of `ACLSP`. Note that such rules are only inserted at the ingress port, since they are not needed at internal nodes.

In addition, the generated flow tables demonstrate that Maple correctly assigns priority levels to flow table rules and uses relatively few priority levels. For example, consider the rule at switch 1, port 3. The rule at priority 0 overlaps with the rules at levels 1 and 2, and hence must have lower priority than those rules. The three rules at priority 1 are all disjoint, since they all match distinct source IP addresses. The rule at priority 2 overlaps the third (from top) rule at priority 1, hence must be at a higher priority. We see that Maple satisfies the required priority constraints, but avoids using unnecessary priority levels when possible.

6.5 Traffic Monitoring

Network administrators in both enterprise and data centers often need to gain visibility into the actual traffic passing through their network. For example, administrators may need to view traffic in order to debug application performance problems or to perform security tasks. The basic requirement of a traffic monitoring system is that an administrator should be able to specify one or more monitoring rules. Each monitoring rule specifies a traffic type and a collection of destination locations (e.g. network ports) to send the given traffic to. The network system must ensure that any traffic matching a given monitoring rule is copied to all ports mentioned in the rule and if multiple rules match a packet and direct the packet to a given port, exactly one copy of the packet is delivered to the specified port (i.e. no duplicates). Typically, monitoring tools, are located at the destination locations to allow administrators to record and view traffic. There are many further requirements of real-world traffic monitoring systems, but for simplicity we consider only the above-mentioned basic requirements here.

Achieving even this basic requirement in a traditional network can be highly challenging. First, either SPAN ports must be configured to mirror traffic from one port to another port, or tap devices must be inserted on to links to duplicate traffic onto the tap device output. Second, one or more tunnels will likely need to be configured in order to move traffic from the tapping point to the destination ports, since this traffic would not normally be sent to the tapping device. Finally, filters should be configured to remove traffic that is not the target of the monitoring. For example, the administrator may want to monitor only HTTP traffic to a particular server, not all traffic traversing a particular link. Filtering the traffic may prevent the monitoring tools from being overloaded with excess traffic.

In contrast, implementing the basic traffic monitoring requirement with SDN, and in particular with Maple is simple, as we now show. In particular, we allow administrators to specify a collection of monitoring rules in a file. The file is very similar to the ACL file. Each line lists a packet predicate with the same format as the ACL file. However, instead of permit or deny actions, the final component of the rule is a list of locations, each expressed as a pair of a switch identifier and port number. Analogous to the ACL controller in the previous section, we model each rule with a

`MonitorRule` object and we modify our ACL interpreter to implement the following function:

```
1 public List<Location> evaluate(List<MonitorRule>,Packet p);
```

This function simply evaluates the packet against each monitoring rule in the input list, collecting the set of all monitoring ports that this packet must be delivered to. We omit the straightforward implementation of this function here.

With this function in place, a simple implementation of monitoring can be accomplished by first evaluating if the packet should be monitored and if so, to which destinations. If the packet's destination is unknown, we simply broadcast the packet on a spanning tree to all edge ports in the network. Otherwise, we evaluate monitoring policy to obtain the set of the monitoring destinations to which the packet should be sent. If there are no monitoring destinations, we send the packet along the shortest path between the sender and the intended recipient. If there are monitoring destinations, we send the packet along a tree which reaches the intended recipient and all the required monitoring destiations. Listing 6.9 shows the Maple algorithmic policy that implements this traffic monitoring logic.

6.6 Summary

In this chapter, we have applied Maple to several problems, including broadcasting, shortest path routing, runtime-variable device access control, declarative access control with ACLs, and traffic monitoring. We have demonstrated that Maple automatically generates correct flow tables that use wildcard rules to encode the algorithmic policy in compact flow tables. We have also illustrated that Maple automatically updates flow tables when state changes occur, either in Maple-maintained state, such as topology, or user-defined state, such as access control state.

Listing 6.9: Traffic monitoring policy.

```
1  public class MonitorSP extends MapleFunction {
2    LinkedList<MonitorRule> monitorPolicy;
3    public MonitorSP() {
4      try {
5        monitorPolicy = parseMonitorPolicyFromFile(new File("monitor1"));
6      } catch (FileNotFoundException e) {
7        System.out.println("Failed: " + e.toString());
8      }
9    }
10
11   Route onPacket(Packet p) {
12     SwitchPort srcLoc = hostLocation(p.ethSrc());
13     SwitchPort dstLoc = hostLocation(p.ethDst());
14     Digraph gr = Digraph.unitDigraph(switches(), links());
15     if (null == dstLoc) {
16       return Route.multicast(srcLoc, gr.minSpanningTree(), edgePorts());
17     } else {
18       List<Location> tapDests = evaluate(monitorPolicy,p);
19       if (tapDests.size() > 0) {
20         return Route.multicast(srcLoc, gr.minSpanningTree(), tapDests.add(dstLoc));
21       } else {
22         List<Link> path = gr.symmetricShortestPath(srcLoc.getSwitch(), dstLoc.getSwitch());
23         return Route.unicast(srcLoc, dstLoc, path);
24       }
25     }
26   }
27 }
```

Chapter 7

Scalable OpenFlow Processing on Multicore Servers

Even with efficient distributed flow table management, some fraction of the packets will "miss" the cached policy decisions at switches and hence require interaction with the central controller. For example, a load balancing network controller may perform a routing decision for each new TCP flow, incurring one packet miss per TCP flow. Tavakoli et al. [63] estimate that this control requirement will generate as many as 20 million flows per second in large data centers consisting of 2 million virtual machines. On the other hand, current controllers, such as NOX [23] or early versions of Nettle [68], are able to process on the order of only 10^5 flows per second [63]. Hence, controller-side processing of misses may become a bottleneck and may prevent SDN controllers from scaling to large networks.

We therefore design and implement an extensible OpenFlow messaging service, McNettle, which can scale performance on multicore CPUs to handle networks with thousands of switches and over 20 million events per second. In particular, McNettle executes user-defined OpenFlow event handlers written in Haskell, a high-level functional programming language. McNettle automatically and efficiently executes event handlers on multiple cores, handling issues of scheduling event handlers on cores, managing message buffers, and parsing and serializing control messages in order to reduce memory traffic, core synchronizations, and synchronizations in the Glasgow Haskell Compiler (GHC) runtime system.

A key design principle instrumental in achieving controller scalability is *switch-level parallelism*: designing the controller's thread model, memory management, and event processing loops to localize controller state relevant to a particular "client" switch. This effectively reduces the amount and frequency of accesses to state shared across the processing paths for multiple client switches.

While many of our design techniques represent "common sense" and are well-known in the broader context of parallel/multicore software design, all of the SDN controllers that we had access to showed major scalability shortcomings, as we explore later in Section 7.3. We were able to address most of these shortcomings through a judicious application of switch-level parallelism principles, such as buffering and batching input and output message streams, and appropriate scheduling and load

balancing across cores.

The chapter is organized as follows. In Section 7.1 we present the overall organization of McNettle controllers and present several simple examples. In Section 7.2 we present several key aspects of McNettle's implementation on multicore CPUs. In Section 7.3 presents evaluations of McNettle.

7.1 Programming with McNettle

A McNettle network controller consists of a collection of *switch event handlers*, each one responsible for communicating with and controlling a single OpenFlow switch. A switch handler is a function that handles each message generated from a particular switch, and is created for a switch by a switch handler factory, which is invoked whenever a switch successfully connects to the control server. The following simple example shows how these pieces fit together in a complete McNettle controller:

```
1  import McNettle
2  import Text.Printf
3  main :: IO ()
4  main = runMcNettle defaultParams factory
5  factory :: SwitchFeatures → Reaction Handler
6  factory features = do
7     liftIO $ printf "Hello, switch %d!\n" (switchID features)
8     return handler
9  handler :: Handler
10 handler ev = return ()
```

The first and second lines import the McNettle module and the *Text.Printf* module which provides the *printf* command. Line 3 defines the type of the main function as *IO* (). (In general, in Haskell we write $e :: t$ to express the assertion that expression e has type t.) The third line defines the *main* function, where execution of the program begins (as is the case with all Haskell programs). In this case, *main* consists of a single command, *runMcNettle* to run the McNettle network controller. The first argument to this command specifies various OpenFlow controller parameters, such as the TCP port on which to listen for switch clients; here we simply use the default settings. The second argument is the switch handler factory to be used for this controller, which is the user-defined *factory* function in this example. The *factory* function takes an argument of type *SwitchFeatures* that specifies various information about a switch, such as its switch identifier and a list of switch ports, with each port providing details such as bandwidth, configuration, and status. Given a *SwitchFeatures* value, the factory performs an imperative computation of type *Reaction Handler* that can perform various side-effects and various commands on the switch and then finally returns a switch *handler* to handle further messages from the given switch. The *Reaction* type is defined by McNettle and allows a computation to send OpenFlow messages to switches, as we will see later in this chapter. In this example, however, *factory* does

not send any initial commands to the switch, but instead simply prints the switch ID and then returns a trivial handler which simply returns without performing any action whenever a message is received from its switch. A *Reaction* computation allows side-effecting computations from the *IO* monad to be executed as part of a *Reaction* through the primitive $liftIO :: IO\ a \rightarrow Reaction\ a$ function. This is used in line 7 above to allow the factory to print the switch ID when a switch connects with the control server.

7.1.1 Learning Switch

To illustrate how switch handlers respond to switch events and program switch flow tables, we now present the standard "learning switch" example in McNettle. The learning switch controller adapts the "learning" algorithm used in transparent LAN Bridges [53] to OpenFlow networks. This algorithm works for networks whose connectivity graph is acyclic (i.e. there is only one path between any pair of nodes). The algorithm "learns" the location of hosts with respect to a switch by observing the source address and incoming port of packets. In particular, when a packet arrives at a switch, the switch infers that since the sender of the packet reached the switch via the given incoming port, the unique path to the sender from the current switch must use the link identified by the incoming port number. Therefore, the switch maintains a table associating hosts to the port on which it last observed the arrival of a packet from the host. Then, when sending packets to a destination host, the switch forwards the packet directly to the learned port, if it has learned a port for this host. If the destination host is not associated with any port, then the switch simply broadcasts the packet out of all but the incoming port.

To implement this algorithm, we introduce the notion of *switch-local state*, which is provided in McNettle using Haskell's mutable variables. The controller in Figure 7.1 uses this approach to implement the transparent bridge learning algorithm. In particular, the factory method creates and initializes the switch-local mutable variable, named *locationMapVar*, holding a dictionary mapping host addresses to port identifiers. Concretely, we use the *Data.Map* datatype to implement the key-value dictionary and use mutable variables of type $IORef\ a$ (a polymorphic datatype of basic mutable variables in the *IO* monad that hold values of type a). In line 6, the factory creates and initializes a new variable for each connecting switch, using the functions $newIORef :: a \rightarrow IO\ (IORef\ a)$ and $liftIO :: IO\ a \rightarrow Reaction\ a$. The factory then partially applies the handler function to the newly created variable, thus providing the handler with its own local mutable state which it uses to implement the learning process. Then, in lines 10-12, the handler updates the location of the sender of the packet by reading the value in the mutable variable using the function $readIORef :: IORef\ a \rightarrow IO\ a$, calculates a new map resulting from inserting an entry for the given packet, and writes this new map back to the mutable variable using the function $writeIORef :: IORef\ a \rightarrow a \rightarrow IO\ ()$. In lines 13-15, the handler looks up the next hop port ID of the intended recipient of the packet and if found, forwards the packet to the appropriate next hop; otherwise the controller floods the packet.

```
1  import McNettle
2  import qualified Data.Map as Map
3  import Data.IORef
4  main = runMcNettle defaultParams factory
5  factory features = do
6    locationMapVar ← liftIO (newIORef Map.empty)
7    return (handler locationMapVar)
8  handler locationMapVar (PacketIn pkt) = do
9    let hdr = etherHeader pkt
10   locMap ← liftIO (readIORef locationMapVar)
11   let locMap' = Map.insert (etherSrc hdr) (inPort pkt) locMap
12   liftIO $ writeIORef locationMapVar locMap'
13   case Map.lookup (etherDst hdr) locMap' of
14     Nothing  → forward pkt [flood]
15     Just port → forward pkt [phyPort port]
16 handler locationMapVar msg = return ()
```

Figure 7.1: Code for the simple learning controller that handles all packets at the controller.

7.1.2 Configuring Flow Tables

The previous learning switch controller (Figure 7.1) leaves flow tables empty and simply forwards each packet at the controller. While this makes the controller simple, the resulting system will perform poorly, since every packet must make an extra round-trip from a switch to the controller for each switch along the path from the sender to the receiver, adding substantial latency to the communication. In addition, this design is not scalable, since both the communication network to the controller and the computational capabilities of the controller itself will become bottlenecks as the flow of packets through the network increases. We now demonstrate how to use McNettle to install flow table entries and thereby offload packet processing to the switches, leading to much better network performance and scalability.

Figure 7.2 shows the type definitions for *Rule* and *Priority* and lists several commands provided by McNettle to modify flow tables. In particular, the *Rule* type is simply a triple of a priority value (defined to be an *Int*), a *Match* value that denotes a subset of packets, and a list of *Action*s. While the OpenFlow protocol provides a concrete representation of a *Match* as a C struct, we provide a small embedded language in Haskell to specify matches. This match language provides a more convenient syntax and prevents the user from making common errors with match conditions. For example, to specify all packets arriving on physical port with ID 3, we would write *in_port* 3. To match all packets arriving on physical port 3 and with destination

```
1 type Rule = (Priority, Match, [Action])
2 type Priority = Int
3 insertRule :: SwitchID → Rule → RuleOptions → Reaction ()
4 deleteRules :: SwitchID → Match → Maybe PseudoPort → Reaction ()
5 forwardWithExactRule :: PacketInfo → [Action] → RuleOptions → Reaction ()
6 forwardWithRule :: PacketInfo → Rule → RuleOptions → Reaction ()
```

Figure 7.2: Key McNettle commands that alter a switch's flow table.

Ethernet address 987 we would write *in_port* 3 ⊓ *eth_dst* 987[1]. To match only IP packets from source prefix `11.2.3.0/24` one would write: *ip_src* (*ipAddress* 11 2 3 0 // 24). To match only TCP packets from source prefix `11.2.3.0/24` one would write *ip_src* (*ipAddress* 11 2 3 0 // 24) ⊓ *tcp*.

insertRule s r opts inserts the rule *r* into switch *s* with options *opts*. The *deleteRules s m mport* deletes all rules at switch *s* whose match conditions represents a subset of the packets represented by the match condition *m*. If *mport* equals *Just p* for some pseudo-port (a pseudo-port can be either a physical port or a special port such as "send to controller") then the command only affects those rules that also mention pseud-port *p*. The *RuleOptions* datatype, whose definition we omit here, can be used to specify options such as soft and hard timeouts and whether a notification should be sent to the controller on removal of a given rule.

The final two commands of Figure 7.2 provide a combined effect of installing a rule *and* processing a referenced packet with the given rule's actions. In particular, *forwardWithExactRule pkt action opts* installs a rule that matches only packets that have the same values for all fields matchable via OpenFlow matches (i.e. an *exact* match). The installed rule will use the specified options and will forward packets with *action*. In addition, the referenced packet will be sent using *action*. *forwardWithRule* allows the user to specify an arbitrary rule to install and with which to process the given packet.

Figure 7.3 shows the learning controller augmented to use exact matches when forwarding packets between hosts whose location is known. This controller is identical to that of Figure 7.1 except line 15 which uses the *forwardWithExactRule* command.

As discussed in Chapter 3, using exact matches can lead to poor performance. Fortunately, for this simple learning controller, we can easily use non-exact rules to improve usage of the flow table and reduce cache misses. Figure 7.4 shows a modification to our previous controller that replaces the invocation of *forwardWithExactRule* with a call to *forwardWithRule* using a rule that matches only on incoming port, source MAC and destination MAC addresses (as discussed in Chapter 3).

1. In Haskell code files, we write `.&&.`, but in this document we typeset it as ⊓

```
1  import McNettle
2  import qualified Data.Map as Map
3  import Data.IORef
4  main = runMcNettle defaultParams factory
5  factory features = do
6    locationMapVar ← liftIO (newIORef Map.empty)
7    return (handler locationMapVar)
8  handler locationMapVar (PacketIn pkt) = do
9    let hdr = etherHeader pkt
10   locMap ← liftIO (readIORef locationMapVar)
11   let locMap' = Map.insert (etherSrc hdr) (inPort pkt) locMap
12   liftIO $ writeIORef locationMapVar locMap'
13   case Map.lookup (etherDst hdr) locMap' of
14     Nothing  → forward pkt [flood]
15     Just port → forwardWithExactRule pkt [phyPort port] myOpts
16 handler locationMapVar msg = return ()
```

Figure 7.3: Learning controller code that populates flow tables with exact matches.

7.1.3 Global State

The switch handlers for the previous controllers all relied only on switch-local state. McNettle also supports global, shared state, which can be used to implement more efficient control. For example, since the host locations are not global variables in the previous controllers, some switches may continue to flood packets for a given host even though the location of the host is known to other switches in the network. This may lead to reduced network performance as more packets are switched at the controller.

In addition, global state can be used to conveniently implement features requiring global coordination. Consider for example a "bandwidth-on-demand" controller. This controller assumes that the links in our network have some maximum known bandwidth capacity and that some communication sessions should have some capacity reserved for them along the route between source and destination. At a high-level, the controller will work as follows: it maintains a global table of host locations, a global shared map of the available bandwidth on each network link, and route tables. When a packet-in arrives from a switch, the receiving switch looks up the destination location and retrieves a route to this location. It then attempts to reserve bandwidth on each link along the route. The reservation may fail if granting the reservation would cause oversubscription on any link in the path. If the reservation succeeds, flow rules are installed in switches and the packet is sent. Otherwise, the packet is dropped. Note that to ensure that bandwidth reservations are honored, the rules installed at switches must enforce bandwidth limits, for example by enqueuing each packet into an appropriately configured queue. We omit this detail in this example.

Figure 7.5 demonstrates the use of Haskell's software transactional memory (STM)

```
1  import McNettle
2  import qualified Data.Map as Map
3  import Data.IORef
4  main = runMcNettle defaultParams factory
5  factory features = do
6    locationMapVar <- liftIO (newIORef Map.empty)
7    return (handler locationMapVar)
8  handler locationMapVar (PacketIn pkt) = do
9    let hdr = etherHeader pkt
10   locMap <- liftIO (readIORef locationMapVar)
11   let locMap' = Map.insert (etherSrc hdr) (inPort pkt) locMap
12   liftIO $ writeIORef locationMapVar locMap'
13   case Map.lookup (etherDst hdr) locMap' of
14     Nothing  -> forward pkt [flood]
15     Just port -> do let src = etherSrc hdr
16                     let dst = etherDst hdr
17                     let match = in_port inPort ⊓ eth_src src ⊓ eth_dst dst
18                     let rule = (0, match, [phyPort port])
19                     forwardWithRule pkt rule myOpts
20 handler locationMapVar msg = return ()
```

Figure 7.4: Learning controller code that populates flow tables with rules with non-exact match conditions.

```
1  reservePath linkVars path amt
2    = res path `orElse` return False
3    where res []           = return True
4          res (u : [])     = return True
5          res (u : v : rest) = do {reserveLink linkVars amt u v; res (v : rest)}
6  reserveLink linkVars amt u v =
7    do let var = linkVariable linkVars (u, v)
8       current ← readTVar var
9       when (current < amt) retry
10      writeTVar var (current − amt)
```

Figure 7.5: Bandwidth reservation using STM.

system to structure reservations as transactions. In particular, the *reservePath* function attempts to reserve bandwidth on each link of a given path; if any link reservation fails, the entire *res* transaction is aborted and the alternative, which simply returns *False*, is executed instead. The *reserveLink* function reads the amount of unreserved capacity from the specified link (based on the *linkVars* table) and updates the remaining capacity if sufficient capacity is available. Given these functions, we can atomically reserve *amount* bandwidth along path *path* with the following command:

```
1  atomically (reservePath linkVars path amount)
```

7.2 McNettle Implementation

McNettle's implementation is organized around switch-level parallelism: each switch handler executes concurrently and possibly in parallel on different CPUs. This design allows for efficient work scheduling, message processing, and memory management.

Throughout this section, we present measurements taken on a DELL Poweredge R815 server with 48 cores and 64 GB memory with Broadcom NetXtreme II network adapters with 8 1Gbps Ethernet ports, and one two port Intel 10Gb NIC with an 82529 Ethernet controller. The 48 cores are provided by four AMD Opteron 6164 processors [14]. Each processor holds two dies, each die containing six cores sharing a 6M L3 cache. The cores, dies and RAM banks form a hierarchy with significant differences in available memory bandwidth and latency due to the location of cores and memory. The workloads used to evaluate McNettle in this section are identical to those used (and described) in Section 7.3.

7.2.1 Scheduling Switch Event Processing

We build McNettle in Haskell using the Glasgow Haskell Compiler (GHC) and threaded runtime system (RTS), leveraging its extensive support for multicore execution. In

particular, McNettle uses a single, lightweight Haskell thread to service work for each OpenFlow switch. Each *switch-handler thread* repeatedly performs the following actions on behalf of its switch: (1) read data from the socket used to communicate with the switch, (2) parse the stream into messages, (3) service messages such as echo requests automatically, (4) execute user-defined event handlers, and (5) write any outgoing messages back to the switch, serializing to the OpenFlow wire format.

By mapping each switch handler to a single Haskell thread, we rely on GHC's threaded RTS to perform work scheduling. GHC's work-conserving scheduler executes many lightweight threads on a small number of OS threads and CPU cores [37, 40] (this is described in more detail in Chapter 8). The scheduler balances load among CPUs by shifting Haskell threads to lightly loaded CPUs. In the context of McNettle, this ensures that all available CPUs are used as needed, and that the processing of a single switch's incoming and outgoing message stream is processed entirely on one CPU, except for infrequent shifts due to load balancing. This affinity reduces the latency introduced by transferring messages between cores, and enables threads serving "busy" switches to retain cached state relevant to that switch.

This design provides abundant fine-grained parallelism, based on two assumptions. First, a typical network includes many more switches than the controller has cores. Second, each individual switch generates request traffic at a rate that can easily be processed by a single processor core. We expect these assumptions to hold in realistic SDN systems [42].

Bottlenecks can arise due to synchronization in the handling of I/O operations. To achieve I/O scalability, McNettle further leverages its thread-per-switch design by ensuring that each OS-level thread invokes I/O system calls (read, write, and epoll in this case) on sockets for switches currently assigned to that particular core. Since the scheduler assigns each switch to one core at a time, this affinity avoids I/O-related contention on those sockets both at application level and within the OS kernel code servicing system calls on those sockets. The implementation of this technique required modifications to the GHC runtime system, which we describe in detail in Chapter 8.

7.2.2 Message Processing

Switches send variable-length, short — typically less than 100 bytes — control messages over their TCP connection with the controller. In addition, the message length is known only by reading an application-layer header field of an OpenFlow message. A naive implementation (and in fact used in both NOX [23] and Nettle [68]) reads a single message at a time from the TCP socket using 2 system calls: one to read the short fixed length header and another to read the remaining body of the message, whose length is now known. As observed by both Tavakoli [63] and Voellmy [67], performing two `read` system calls per OpenFlow message becomes a bottleneck at high throughput rates. To reduce the overhead, McNettle reads data from the socket into a buffer in batches and then parses OpenFlow messages from the incoming buffer. This reduces the number of system calls and dramatically improves performance. Figure 7.6 demonstrates the throughput and proportion of execution time in system calls for several batch sizes for the learning switch controller when each message is received

separately and when batching is used.

Large batched `read` operations often leave the head of a switch's message stream in between message boundaries. This implies that using multiple user-level threads to process a switch's message stream would cause frequent thread stalls while one thread waits for the parsing state of a previous parsing thread to become available. Large non-atomic `write` operations cause similar problems on the output path. However, since McNettle dedicates a thread to each switch, its switch-level parallelism avoids unnecessary synchronization and facilitates efficient request batching.

McNettle applies batching to output as well: the messages generated by user-specified control logic are serialized into a buffer, which is then written to the switch socket after processing a single batch of incoming messages.

7.2.3 Memory Management

The standard network libraries for Haskell use a common functional programming idiom for reading from a socket: they allocate a new immutable byte array of the desired size and fill it with data from the socket buffer. This immutable byte array is then passed to parsing functions.

McNettle's strategy of reading large chunks of data from a socket uncovered a bottleneck in GHC's memory allocator. In particular, GHC allocates "large" byte arrays from a global block pool, rather than the core-local nurseries that are used for other objects, and this global block pool is protected by a lock. Haskell's standard network libraries allocate and fill a new, immutable byte array for each read operation. Since McNettle reads from sockets in large chunks, using the standard library causes frequent allocations of large byte arrays, leading to heavy contention on the lock on the global large object pool.

To avoid this bottleneck, McNettle allocates fixed size buffers for each switch, reads data from sockets into these buffers, and writes data from these buffers to sockets. McNettle uses modified versions of Haskell's standard network functions to read and write using these buffers. This technique dramatically improves performance: the maximum throughput for the learning controller running with 40 cores and 500 switches when using static pre-allocated buffers was a factor of 5 higher than when allocating a new byte array for every socket read.

GHC's RTS uses a parallel, generational, stop-the-world garbage collector [34]. While it performs well and is highly parallelized, the stop-the-world collection requires all of the RTS's OS threads to synchronize with barriers to start garbage collection and to re-start mutators. To reduce the impact of the this all-core synchronization, McNettle uses a large allocation area (generation 0 area), which reduces the frequency of garbage collections. Figure 7.7 demonstrates the effect of different allocation area sizes and numbers of cores on the maximum throughput and average pause times of the learning controller.

Since all cores must stop executing mutators for the duration of the garbage collection, GC pause times may reduce controller responsiveness. Unfortunately, pause times increase as the allocation area increases, as measured in Figure 7.8. Recent work [35] on a thread-local collector for GHC would allow cores to perform core-local

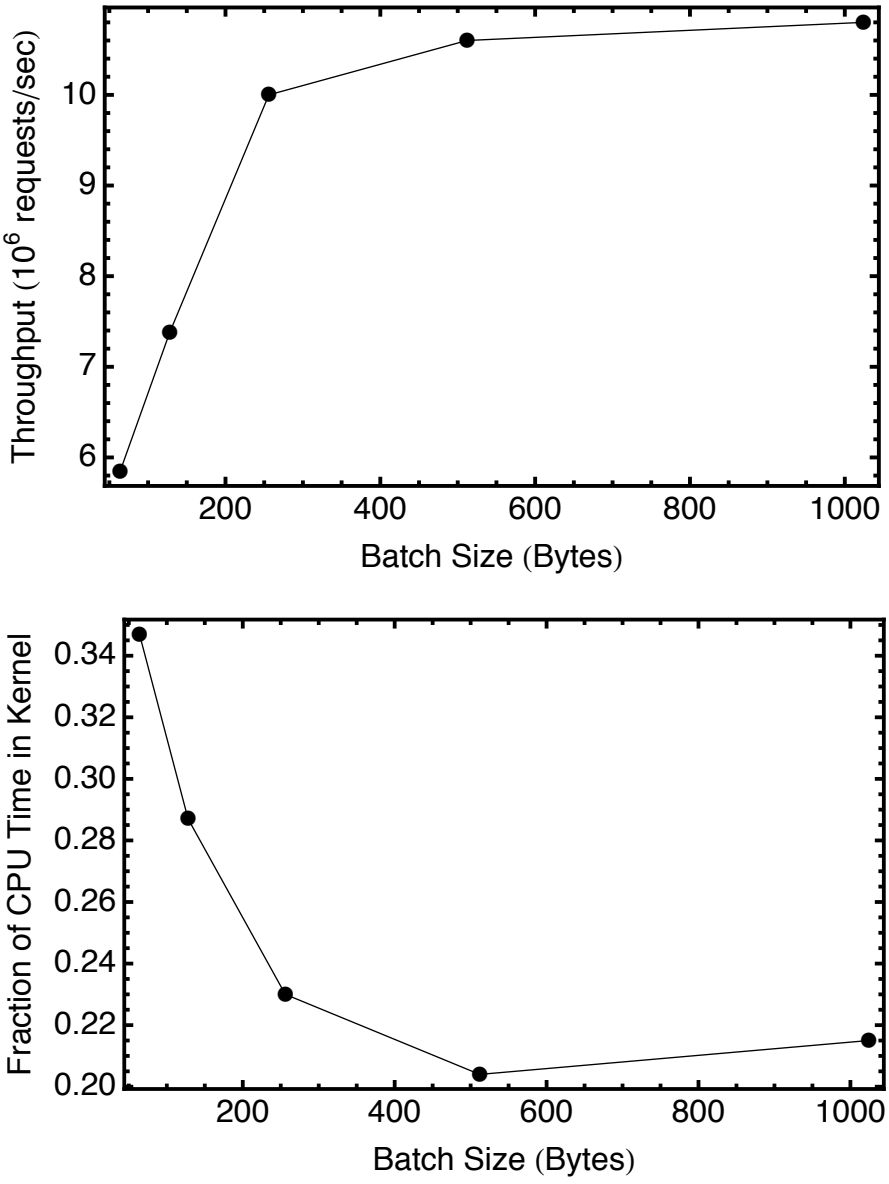


Figure 7.6: Maximum throughput (in millions of messages per second) and percent system time (of total CPU time) as a function of batch size when running the learning controller with 40 cores and 500 switches.

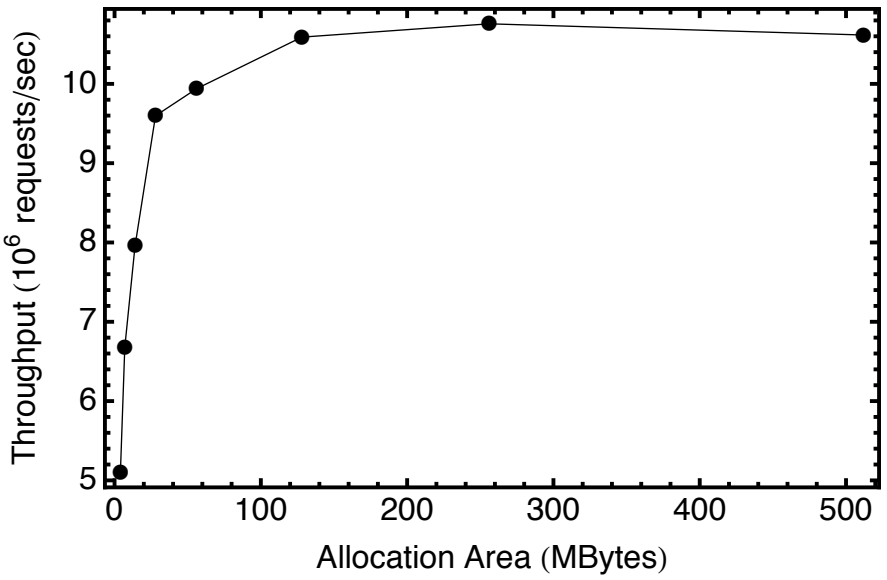


Figure 7.7: Maximum throughput (in millions of messages per second) for different allocation areas (generation 0) when running the learning controller with 40 cores and 500 switches.

garbage collection without stopping all other cores, which may reduce the impact of GC pause times.

7.3 Evaluation

We evaluate the performance of McNettle using the learning switch controller with exact match, since this controller is also available in other available frameworks. We measure both throughput (*i.e.*, the number of requests that our controller can process each second) and latency. We compare McNettle with Beacon [18] and NOX-MT [66], two well-known OpenFlow control frameworks that aim to provide high performance.

Server: We run our OpenFlow controllers on an 80 core SuperMicro server, with 8 Intel Xeon E7-8850 2.00GHz processors, each having 10 cores with a 24MB smart cache and 32MB L3 cache. We use four 10 Gbps Intel NICs. Our server software includes Linux kernel version 3.7.1 and Intel ixgbe driver (version 3.9.17).

Workload: We simulate 100 switches with a version of Cbench [57] modified to run on several servers, in order to generate sufficient workload. We use 8 Cbench workload servers connected over 10Gbps links to a single L2 switch, which connects to four 10Gbps interfaces of our control server. We limit the packet-in messages generated by CBench, so that the number of requests outstanding from a single CBench instance does not exceed a configurable limit. This allows us to control the response time while evaluating throughput.

Results: Figure 7.9(a) shows the throughput as a function of the number of cores used for all three systems. We observe that McNettle serves over 20 million requests per second using 40 cores and scales substantially better than Beacon or NOX-MT. In

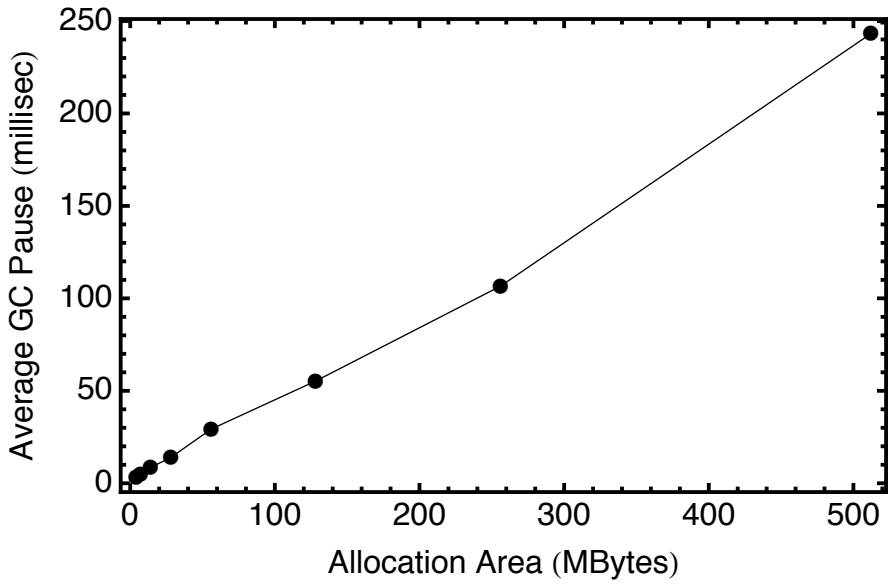
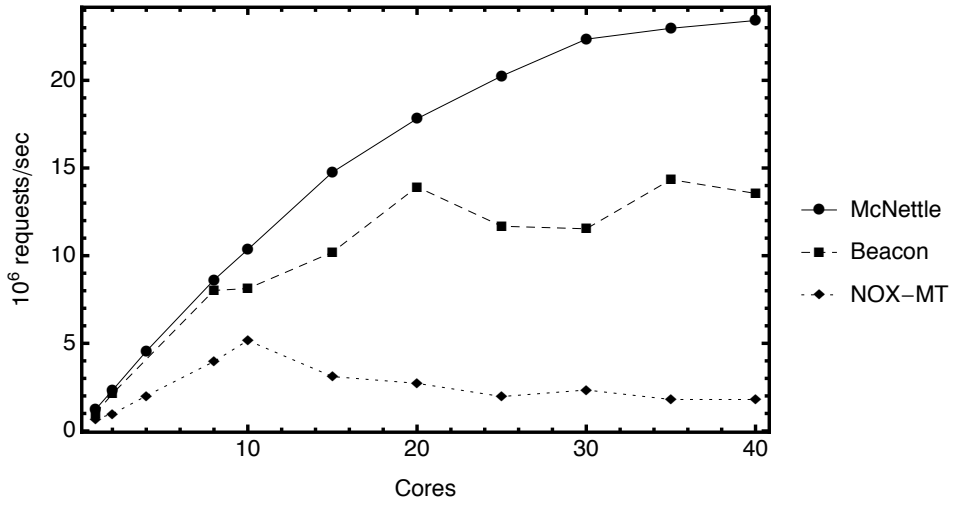


Figure 7.8: Average GC pause time for different allocation areas (generation 0) when running the learning controller with 40 cores and 500 switches.

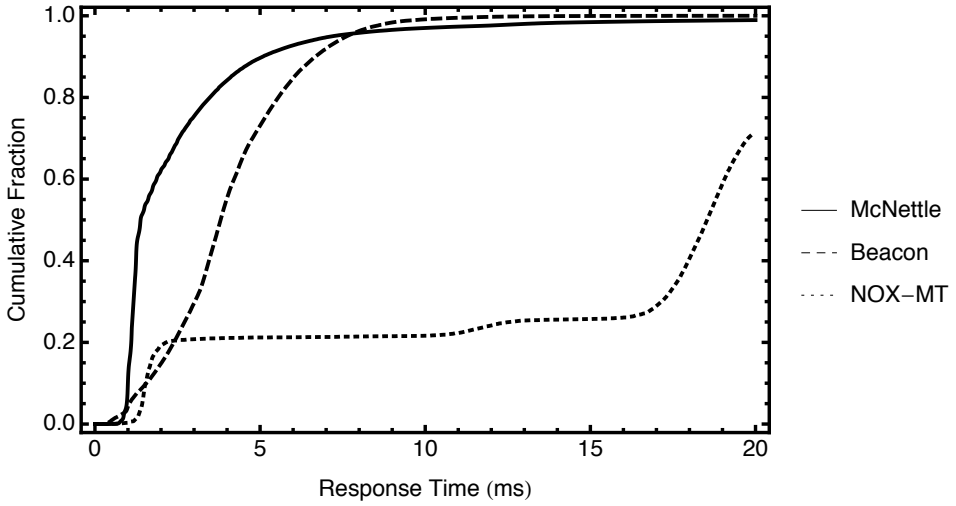
particular Beacon scales to less than 15 million/second, and NOX-MT is only around 2 million/second. Figure 7.9(b) shows the corresponding latency CDF for all three systems. The median latency of McNettle is 1 ms, Beacon is almost 4 ms, and NOX-MT reaches as high as 17 ms. The 95-percentile latency of McNettle is still under 10 ms.

7.4 Summary

This chapter introduced McNettle, an OpenFlow message service (or OpenFlow network controller) that allows users to program OpenFlow message handlers in Haskell, a high-level, statically-typed functional programming language. McNettle efficiently implements OpenFlow message processing and user-defined logic on Haskell's lightweight threads and GHC's multicore runtime system by applying switch-level parallelism to avoid unnecessary synchronization between CPU cores and by resolving bottlenecks in the GHC runtime system. As a result, McNettle is able to effectively utilize 40 CPU cores to handle over 20 million flow setups per second generated from 100 switches.



(a) Throughput comparison



(b) Latency comparison

Figure 7.9: Throughput and latency of SDN controllers.

Chapter 8

Mio: High-performance IO Notification in GHC

Although most of McNettle's implementation is entirely in Haskell, one component of GHC in particular, the *GHC IO Manager*, presented a significant bottleneck. To overcome this bottleneck and reach our performance targets, we designed and implemented a new IO manager for GHC, called the Multicore IO Manager (Mio). This chapter focuses on the design, implementation and evaluation of Mio. Since Mio solves performance problems for many network servers written in Haskell, we perform evaluations with various web servers.

8.1 Introduction

Haskell threads (also known as green threads or user threads) provide a key abstraction for writing high-performance, concurrent programs [54] in Haskell [33]. For example, consider a network server such as a web server. Serving a web request may involve disk IO operations to fetch the requested web page, which can be slow. To achieve high performance, web servers process other requests, if any are available, while waiting for slow disk operations to complete. During high load, a large number of requests may arrive during the IO operations, which gives rise to many requests in progress concurrently. A naive implementation, using one *native thread* (i.e. OS thread) per request would lead to the use of a large number of native threads, which would substantially degrade performance due to the relatively high cost of OS context switches [74]. In contrast, Haskell threads are *lightweight* threads, which can be context switched without incurring an OS context switch and with much lower overhead. Hence, lightweight Haskell threads are particularly well-suited for implementing high performance servers.

Haskell's lightweight threads reduce the incentive for programmers to abandon the simple, threaded model in favor of more complex, event-driven programming to achieve acceptable performance, as is often done in practice [74]. Event-driven programs require that a programmer reorganize a sequential program into a harder-to-understand, more complex structure: pieces of *non-blocking* code segments and

a state machine that determines the operations that will be performed upon each event. In contrast, the processing logic at a network server to handle a client request is typically much easier to understand when programmed as a single thread of execution rather than as collection of event handlers [70].

Given the importance of Haskell threads, the Glasgow Haskell Compiler (GHC) [36], the flagship Haskell compiler and runtime system (RTS), provides substantial support to implement Haskell threads. For example, realizing that CPU may become a bottleneck, GHC RTS introduces a load-balancing multicore scheduler to try to leverage the current and future trend of multicore processors. To avoid memory bottlenecks, GHC RTS introduces a parallel garbage collector and efficient multicore memory allocation methods. To avoid using one native thread for each blocking I/O operation, GHC RTS introduces an *IO manager* [39, 52] to support a large number of Haskell threads over a small number of native threads. The objective of these components is to provide a highly scalable Haskell thread implementation.

Unfortunately, despite introducing many important components, GHC RTS did not scale on multicores, leading to poor performance of many network applications that try to use lightweight Haskell threads. In particular, our experiments demonstrate that even embarrassingly concurrent network servers are typically unable to make effective use of more than one or two cores using current GHC RTS.

In this chapter we first diagnose the causes for the poor multicore performance of Haskell network servers compiled with GHC. We identify that the GHC IO manager (also known as the "new IO manager" [52]) as the scaling bottleneck. Through a series of experiments, we identify key data structure, scheduling, and dispatching bottlenecks of the GHC IO manager.

Our next contribution is that we redesign the GHC IO manager to overcome multicore bottlenecks. Our new design, called *Mio*, introduces several new techniques: (1) *concurrent callback tables* to allow concurrent use of the IO manager facilities, (2) *per-core dispatchers* to parallelize the work of dispatching work to waiting threads, improve locality, and reduce cross-core interactions and (3) *scalable OS event registration* to reduce the use of non-scalable and expensive system calls. The new techniques are all "under-the-hood" and will apply transparently to all Haskell programs without modification. Mio is simple to implement, with only 874 new lines of code added to the GHC code base and 359 old lines of code deleted. Mio has been incorporated into GHC and will be released in GHC 7.8.1.

In Chapter 7, we demonstrated that with Mio, McNettle scales effectively to 40+ cores, reaches a throughput of over 20 million new requests per second on a single machine, and hence becomes the fastest of all existing SDN controllers. In this chapter, we further evaluate Mio using several web servers. Specifically, with Mio, realistic HTTP servers in Haskell scale to 20 CPU cores, achieving peak performance up to factor of 6.5x compared to the same servers using previous versions of GHC. The latency of Haskell servers is also improved: using the threaded RTS with Mio manager, when compared with *non-threaded* RTS, adds just 6 microseconds to the expected latency of the server under light load, and under a moderate load, reduces expected response time by 5.7x when compared with previous versions of GHC.

The rest of this chapter is organized as follows. Section 8.2 presents background

on the design and API of the GHC IO manager. Section 8.3 shows problems of the GHC IO manager and our approaches to overcome them. Section 8.4 and 8.5 discuss implementation details and OS bottlenecks and bugs, respectively. Section 8.6 presents an evaluation of our techniques. Finally, Section 8.8 summarizes this chapter and suggests possible directions for future work on this topic.

8.2 Background: GHC Threaded RTS

A GHC user can choose to link a Haskell program with either the threaded RTS or the non-threaded RTS. In this section, we briefly review the operation of the threaded RTS and the GHC IO manager, which are presented in more detail in [38, 39, 52], respectively.

8.2.1 Threaded RTS Organization

A concurrent Haskell program is written using Haskell threads typically created as a result of invocations of `forkIO` by the user program. These Haskell threads are multiplexed over a much smaller number of native threads. Figure 8.1 shows the structure of GHC's threaded RTS. A Haskell program running with the threaded RTS makes use of N cores (or CPUs), where N is typically specified as a command-line argument to the program. The RTS maintains an array of *capability* (also called a *Haskell Execution Context (HEC)*) data structures with each capability maintaining the state needed to run the RTS on one core. This state includes a run queue of runnable Haskell threads and a message queue for interactions between capabilities. At any given time, a single native thread is executing on behalf of the capability and holds a lock on many of its data structures, such as the run queue. The native thread running the capability repeatedly runs the thread scheduler, and each iteration of the scheduler loop processes the message queue, balances its run queue with other idle capabilities (i.e. performs load balancing) and executes the next Haskell thread on its run queue (among other things).

In order to support blocking foreign calls, the RTS may in fact make use of more than N native threads. When a Haskell thread performs a foreign call that may block, the native thread running the capability releases the capability and creates a new native thread to begin running the capability, and then performs the blocking OS call. This allows the RTS to continue running other Haskell threads and to participate in garbage collections, which requires synchronization across all capabilities. In fact, a pool of native threads is maintained per capability so as to avoid creating a new native thread for each blocking call.

8.2.2 GHC IO Manager

Although the GHC IO manager is an ordinary Haskell library, distributed as part of the Haskell base libraries, it is in fact tightly coupled with the threaded RTS, which is primarily written in C. During the RTS initialization sequence, the RTS

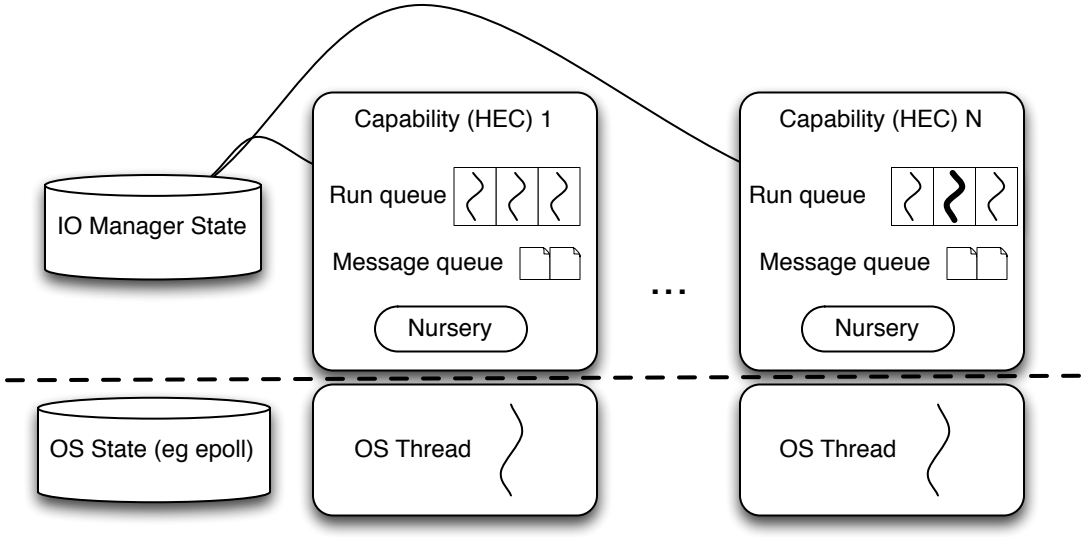


Figure 8.1: Components of the threaded RTS, consisting of N capabilities (caps), each running a scheduler which manages its capability's run queue and services messages to it from other capabilities. At any given time, a single native thread is executing a capability. The system uses a single GHC IO manager component, shared among all capabilities.

calls a function implemented in the GHC IO manager library (i.e. in Haskell) to initialize the state of the GHC IO manager. This function initializes a value of type `EventManager` and stores this value to a global variable. This value is indicated as the component labelled "IO Manager State" in Figure 8.1. The `EventManager` structure contains several pieces of state, including a *callback table*, which keeps track of the events that are currently being waited on by other Haskell threads. The callback table is a search tree, keyed on file descriptor, and has type `IntMap [FdData]`. Each value stored in the table is a list of `FdData` values, where an `FdData` value includes the file descriptor, a unique key, the type of operation (i.e. read or write) being waited on, and a callback function to invoke when the operation is ready. Specifically, the GHC IO manager provides the following API to retrieve the GHC IO manager and to register or unregister interest in file-related events:

$$\begin{aligned}
&\mathit{getSystemEventManager} :: \mathit{IO}\ (\mathit{Maybe}\ \mathit{EventManager})\\
&\mathit{registerFd} :: \mathit{EventManager} \to \mathit{IOCallback} \to \mathit{Fd} \to \mathit{Event} \to \mathit{IO}\ \mathit{FdKey}\\
&\mathit{unregisterFd} :: \mathit{EventManager} \to \mathit{FdKey} \to \mathit{IO}\ ()
\end{aligned}$$

The `Fd` parameter is the file descriptor of an open file and the `Event` parameter is either `Read` or `Write`. The `FdKey` value returned by a call to `registerFd` includes both the file descriptor and the unique key, uniquely identifying the subscription. The unique key is automatically generated by the GHC IO manager in `registerFd` and is used to allow multiple subscriptions to exist on the same file descriptor, where each subscription can be independently cancelled.

The callback table is stored in an `MVar` [54] that is stored in the `EventManager`

value. An `MVar` is essentially a mutable variable protected by a mutex; any modification obtains the *MVar* lock and returns the lock when finished. If a thread executes an operation on *MVar* while the lock is taken, then the thread is enqueued on a FIFO queue of threads waiting to gain access to the *MVar*. This ensures that Haskell threads are granted fair access to an *MVar*. The `registerFd` function takes the callback table *MVar* lock, and then inserts (or modifies) an entry in the callback table and registers the event subscription using the underlying OS event notification mechanism, for example `epoll` [32] on Linux, `kqueue` [31] on BSD variants, and `poll` on other operating systems, and finally restores the callback table lock. Similarly, `unregisterFd` takes the callback table lock, removes the appropriate entry, removes the event subscription from the underlying OS event queue, and then restores the lock. Note that the callback table is stored in an `MVar` in order to ensure that the callback table and OS event queue are operated on atomically.

The GHC IO manager initialization function also forks a Haskell thread, called the *dispatcher thread*. The dispatcher thread executes a poll loop, repeating the following steps. It performs a blocking call to the appropriate OS function to wait on all registered events (e.g. `epoll_wait` on Linux). When it returns from the call, it retrieves the ready events, and for each one, it takes the callback table lock, retrieves the relevant callback, restores the lock and then invokes the provided callback. Having dispatched all the callbacks, the dispatcher thread repeats the process indefinitely (i.e. calls `epoll_wait` to wait for new events, etc.). The dispatcher thread is shown in Figure 8.1 as the bold Haskell thread on capability N's run queue. With this approach, the RTS uses a single extra native thread to perform a blocking system call, instead of using one native thread per Haskell thread performing a blocking system call.

8.2.3 OS Event Notification

As alluded to above, each OS provides one or more event notification facilities. Some mechanisms, such as `select` and `poll`, offer a single system call which accepts an argument listing all the events of interest to the caller and blocks until at least one of those events is ready; upon returning the system call indicates which of the input events have occurred. Other mechanisms, such as `epoll` on Linux and `kqueue` on BSD allow the program to register or unregister interest in events, and then separately wait on all registered events. This second method has been shown (in [31]) to be more efficient when a large number of files is being monitored, since it avoids passing the full list of events of interest to the OS each time the program checks for event readiness. For concreteness, Figure 8.2 shows the API provided by `epoll`.

Specifically, `epoll_create` creates a new `epoll` instance and returns a file descriptor for the instance. `epoll_ctl` registers or unregisters (depending on the value of `op`) interest in an event on file descriptor `fd` for events indicated by the `event` pointer with the `epoll` instance `epfd`. In particular, `struct epoll_event` includes a bitset (also known as an *interest mask*) indicating which event types, such as `EPOLLIN` and `EPOLLOUT` (readable and writable, respectively) the caller is interested in. Finally, `epoll_wait` waits for events registered with `epoll` instance `epfd` and receives up to

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

Figure 8.2: `Epoll` API.

`maxevents` events into the `events` array, or returns with no events if `timeout` time has elapsed.

8.2.4 Thread API

The file registration API presented in the previous section is used to implement more convenient functions used by Haskell threads. In particular, the GHC IO manager provides the following two Haskell functions:

$$threadWaitRead, threadWaitWrite :: Fd \to IO\ ()$$

These functions allow a Haskell thread to wait until the OS indicates that a file can be read from or written to without blocking. They are logically blocking, i.e. the calling Haskell thread will appear to block until the requested operation can be performed by the OS.

These two functions are typically used in the following way. First, Haskell programs place any files (e.g. sockets) into non-blocking mode. Then, when performing a read or write on a file, the OS will either perform the operation without blocking, or will return an error code indicating that the operation could not be performed without blocking. The Haskell thread then typically handles this latter condition by executing either `threadWaitRead` or `threadWaitWrite`, as appropriate. This call will only return when the file is readable or writable, at which point the thread will attempt the preceding sequence again. More concretely, a thread may perform the following command to send a number of bytes on a given socket:

```
send :: Fd → Ptr Word8 → Int → IO Int
send sock ptr bytes = do
  result ← c_send sock ptr (fromIntegral bytes) 0
  if result ≡ −1
    then do err ← getErrno
            if err ≡ eAGAIN
              then do threadWaitWrite sock
                      send sock ptr bytes
              else error "unexpected error"
    else return $ fromIntegral result
```

In this code, `sock` is the file descriptor of an open socket, `ptr` is a pointer to a byte array, and `bytes` is the number of bytes to send from the beginning of `ptr`.

We assume that we have a function `c_send` that calls the `send()` system call of the underlying OS. We also assume that we have a function `getErrno` that provides the error number returned and a constant `eAGAIN` that is used by the OS to indicate when the operation would block. Such code is typically implemented in a library that provides more convenient functions. For example, the `Network.Socket.Bytestring` library provides a function `sendAll` whose implementation is similar to that given for `send` above.

The `threadWaitRead` and `threadWaitWrite` functions are both implemented using `threadWait`, shown in Figure 8.3. `threadWait` first creates a new, empty `MVar` (distinct from the `MVar` holding the callback table), which we call the invoking *thread's wait variable*, and uses `getSystemEventManager` to get the manager value. It then registers a callback using `registerFd` for the current file descriptor and appropriate `Event` (i.e. read or write event), and then proceeds to wait on the initially empty `MVar`. Since it encounters an empty `MVar` (typically), the scheduler for the capability running this thread will remove the current thread from its run queue. Later, when the OS has indicated that the event is ready, the dispatcher thread will invoke the registered callback, which first unregisters the event (using `unregisterFd_`) with the IO manager and then writes to the thread's wait variable, causing the scheduler to make the original thread runnable again. As a result, the original thread will be placed on the run queue of the capability on which it was previously running. At this point, the `threadWait` function returns and the original thread continues.

```
threadWait :: Event → Fd → IO ()
threadWait evt fd = mask_ $ do
  m ← newEmptyMVar
  Just mgr ← getSystemEventManager
  let callback reg e = unregisterFd_ mgr reg ≫ putMVar m e
  reg ← registerFd mgr callback fd evt
  evt' ← takeMVar m `onException` unregisterFd_ mgr reg
  when (evt' `eventIs` evtClose) $ ioError $
    errnoToIOError "threadWait" eBADF Nothing Nothing
```

Figure 8.3: Method for a thread to wait on an event.

8.3 Analysis & Multicore IO Manager Design

In this section, we demonstrate the bottlenecks in the GHC IO manager and the techniques we use to overcome these. We first introduce a simple HTTP server written in Haskell, which we will use throughout this section. We then diagnose and solve bottlenecks one at a time, with each solution revealing a subsequent bottleneck at a higher number of cores. In particular, we explain our three main techniques to improve performance: concurrent callback tables, per-core dispatchers and scalable

OS event registration.

8.3.1 The Simple Server

We illustrate the problems with the GHC IO manager by using a concurrent HTTP network server written in Haskell, called `SimpleServer`[1], that *should* be perfectly scalable: it serves each HTTP client independently, using no shared state and no synchronization between clients (e.g. locks) anywhere in the Haskell program. With a sufficient number of concurrent clients, this is a plentiful, perfectly parallelizable workload. As a result, we would expect that as we utilize more cores, `SimpleServer` should be able to server more requests per second, provided there are a sufficient number of clients. Unfortunately, as we will show, with the GHC IO manager, this is not the case.

`SimpleServer` is a drastically simplified HTTP server. Its key parts are listed in Figure 8.4. The server consists of a main thread that starts a listening socket and then repeatedly accepts incoming connections on this socket, making use of the `network` package for basic datatypes and functions like *accept*. The main thread then forks a new worker (Haskell) thread for each newly accepted connection. A worker thread then repeatedly serves requests. To handle a single request, it receives *ByteString* values (using the *Network.Socket.ByteString.recv* function) until a single request of length *requestLen* has been received, and then sends a single response (using the *Network.Socket.ByteString.sendAll* function). The connection is closed when the worker thread receives a zero length *ByteString*. The *sendAll* and *recv* functions internally call *threadWaitWrite* and *threadWaitRead* whenever the respective operation would block, much as the *send* function shown earlier does.

```
main :: IO ()
main = do listenSock ← startListenSock
          forever $ do (sock, _) ← accept listenSock
                       forkIO $ worker sock

worker :: Socket → IO ()
worker sock = loop requestLen
  where loop left | left ≡ 0    = do sendAll sock reply
                                     loop requestLen
                  | otherwise = do bs ← recv sock left
                                   let len = B.length bs
                                   when (len ≢ 0) $ loop (left − len)
```

Figure 8.4: Main parts of `SimpleServer`.

This highly simplified server performs no disk access, performs no HTTP header parsing, and allocates only one *ByteString* for responses. Using this simplified server

1. `https://github.com/AndreasVoellmy/SimpleServer`

allows us to avoid application-level bottlenecks that may arise from implementations of features such as HTTP parsing or protection against denial of service attacks, both of which are required in real HTTP servers. In Section 8.6, we evaluate more realistic web servers.

We evaluate `SimpleServer` using a closed system, connecting multiple clients to the server where each client repeatedly sends a request, waits for the response, and then repeats. The clients generate requests with the fixed length expected by `SimpleServer`. Throughout this section, we evaluate the server's throughput, in requests/second, using 400 concurrent clients. We run `SimpleServer` and the clients on different Linux servers using a setup described in detail in Section 8.6.1.

8.3.2 Concurrent Callback Tables

Unfortunately, as the curve labelled "Current" in Figure 8.5 shows, the GHC IO manager scales poorly, reaching a maximum performance at 2 cores serving 37,000 requests per second and then declining, with per-core performance rapidly deteriorating. Despite our application being perfectly parallelizable, the RTS essentially forces a completely sequential execution.

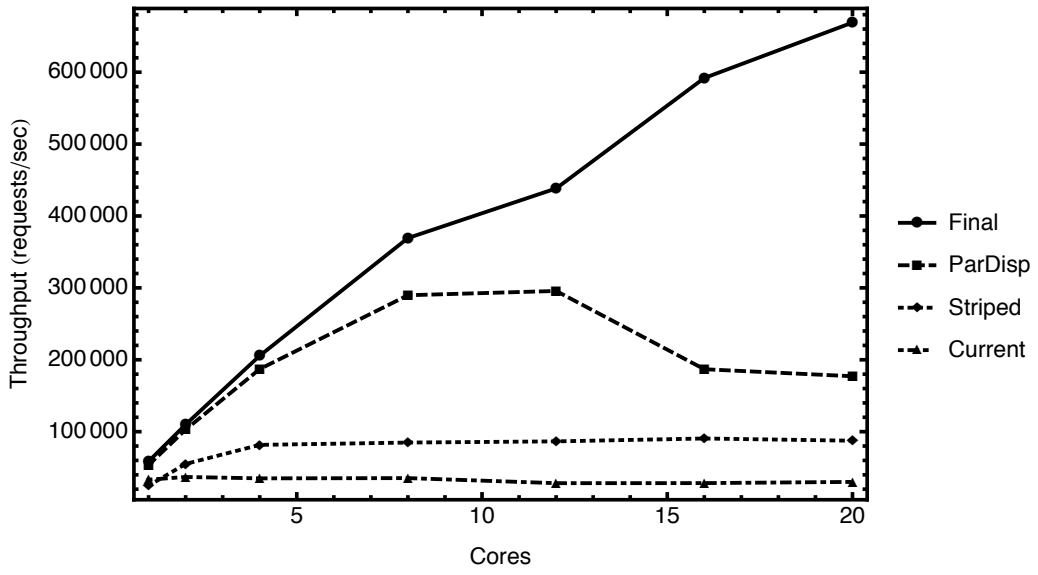


Figure 8.5: Throughput of `SimpleServer` shown as number of requests served per second. "Current" is the GHC 7.6.3 IO manager. "Striped", "ParDisp" and "Final" are described in Sections 8.3.2, 8.3.3 and 8.3.4, respectively.

The most severe problem in the GHC IO manager is that a single *MVar* is used to manage the callback table. The problem becomes evident by carefully examining an *event log* [28] recorded during an execution of `SimpleServer`. By compiling and running `SimpleServer` with event logging turned on, an execution of `SimpleServer` generates a log file that records the timing of various events, such as thread start and stop times, thread wakeup messages, thread migrations, and user-defined events.

We then visualize the event logs offline, using the Threadscope program [65], which produces graphical timelines for an event log. Threadscope produces one timeline for each capability (labelled "HEC") and also produces an overall utilization timeline (labelled "Activity"). The capability timelines are white when no Haskell thread is being executed and are light gray when executing a Haskell thread. If there is sufficient space, the thread ID is indicated in the light gray region indicating a running Haskell thread. In the event logs shown in this chapter, we use solid, black bars to indicate when one thread sends a thread wakeup message for another Haskell thread, and we use dashed, gray bars to indicate when a `SimpleServer` worker thread calls *threadWaitRead* on the thread's socket.

Figure 8.6 shows a detailed fragment of an event log recorded when running `SimpleServer` using the GHC IO manager and using 4 capabilities (i.e. $N = 4$) and which demonstrates contention on the callback table variable. In particular, we see that thread 18 on HEC 1 attempts to wait on its socket (the dashed, gray bar), but fails due to the callback table variable already being taken, in this case by thread 2, which is the dispatcher thread of the GHC IO manager. Therefore, thread 18 enqueues itself on the callback table variable wait queue and is removed from the run queue, after which HEC 1 becomes idle. Eventually, when the dispatcher thread (thread 2) runs again on HEC 0 (after migrating from HEC 3 just moments earlier), it releases the callback table variable and HEC 1 is messaged to indicate that thread 18 is next in line to take the callback table variable (the solid line on HEC 0 following thread 18's wait). The dispatcher thread then immediately attempts to take the callback table variable again, causing it to be queued and descheduled. Just after this, thread 50 on HEC 2 also attempts to wait on its socket (dashed gray line), but also finds the callback table variable taken, and therefore also enqueues itself on the variable's waiter queue. HEC 1 then runs thread 18, which finishes its registration, returns the lock, and notifies HEC 0 to run thread 2 again (second solid bar for thread 18). When thread 2 runs again, it finishes its work, in this case dispatching a new event to thread 13, and notifies HEC 2 for thread 50.

This sequence shows in detail that not only are workers interfering with each other in order to register interest in events, but workers may interfere with the dispatcher, preventing the dispatcher from placing runnable threads on the run queues of idle cores. As the wait queues on the callback table variable build up, the dispatcher thread may ultimately have to wait its turn in a long line to get access to the callback table. This results in the episodic behavior seen in Figure 8.7. When the dispatcher gains access to the callback table, most activity has died down, and it is able to dispatch a large number of threads. Subsequently, these threads serve their connections and then as they attempt to wait, large queues form, and the dispatcher thread is unable to run long enough to dispatch any more work. Thus, system activity begins to decline, until the dispatcher gains access and starts the cycle again.

We can resolve this severe contention by using a data structure that allows for more concurrency. We apply a simple solution, which we call *lock striping*: we use an array of callback tables and hash file descriptors to array locations. Specifically, we change the type of the callback table field of the *EventManager* data type from $\mathit{MVar}\ (\mathit{IntMap}\ [\mathit{FdData}])$ to $\mathit{Array}\ \mathit{Int}\ (\mathit{MVar}\ (\mathit{IntMap}\ [\mathit{FdData}]))$ and use a fixed

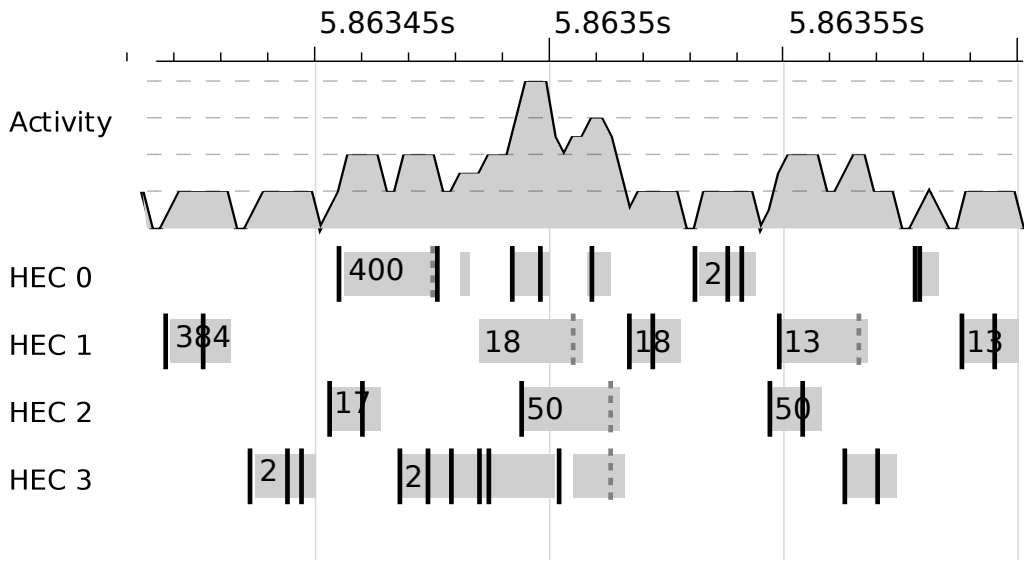


Figure 8.6: Event log fragment from an execution of `SimpleServer` with the GHC IO manager and 4 capabilities and illustrating contention for the callback table variable.

array size of 2^5. With this data structure, registration or notification on a file descriptor simply performs the original registration or notification procedure to the callback table at the location that the file descriptor hashes to. Hence, registrations or notifications on different file descriptors do not interfere. We store the callback table in each array location in an *MVar* so that we can perform both the callback table update and the operation on the OS event subsystem in one atomic step for any file descriptor. This prevents a race condition in which two updates on the same file descriptor are applied to the callback table in one order and to the OS event subsystem in the reverse order.

To illustrate the effectiveness of this technique, we measure performance applying only lock striping to the GHC IO manager. The curve labelled "Striped" in Figure 8.5 shows that the resulting server scales better through 4 cores, reaching a performance of over 80,000 requests per second, more than doubling peak performance of the program using the GHC IO manager.

Figure 8.8 shows a segment of an event log taken with 4 cores that indicates in more detail how the severe contention is relieved and how work is dispatched more smoothly. The activity time-graph is close to maximum throughout this segment, indicating that all four cores are now effectively utilized. In more detail, we can see the dispatcher thread (thread 2) on HEC 0 is able to dispatch work to threads (indicated by solid black bars on HEC 0) at the same time that the worker threads serve client requests and register callbacks to wait on again (dashed bars on HECs 1, 2 and 3).

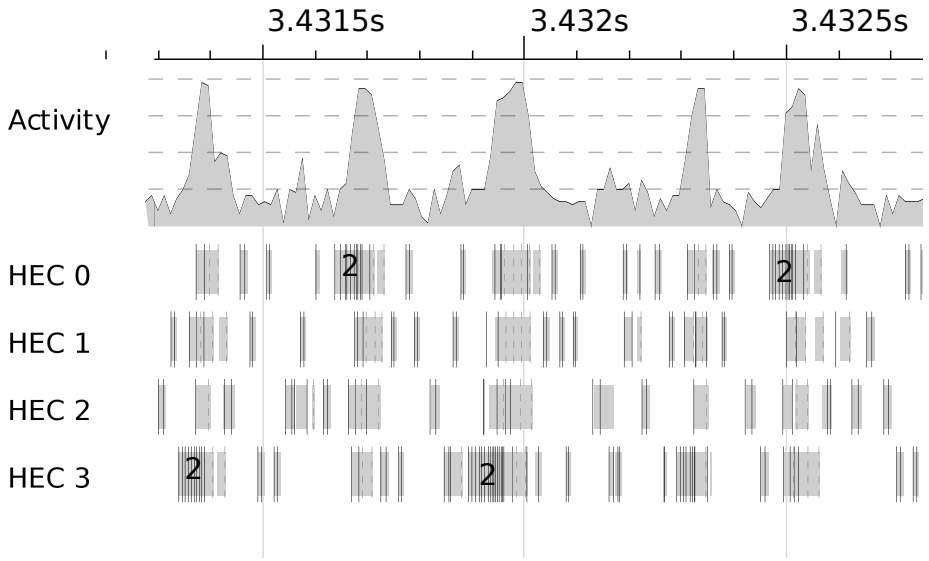


Figure 8.7: Event log fragment from an execution of `SimpleServer` with the GHC IO manager and 4 capabilities: lock contention leads to HECs mostly idling.

8.3.3 Per-Core Dispatchers

Unfortunately, concurrent callback tables are not sufficient to allow `SimpleServer` to continue scaling beyond 4 cores. The event log fragment shown in Figure 8.9 is taken from a run with 8 cores and provides insight into the problem. In particular, we see that the dispatcher thread (on HEC 1) is very busy, while other HECs are mostly idle. We see that as the dispatcher thread notifies each thread, the notified thread quickly executes and finishes. The dispatcher is simply unable to create work fast enough to keep 7 other cores busy. The underlying problem is that GHC IO manager design essentially limits the notification work to consume no more than 1 CPU core, and this workload (`SimpleServer`) results in full utilization of the dispatcher component, creating a bottleneck in the system.

We solve this problem by introducing per-core dispatcher threads, effectively eliminating the restriction of 1 CPU core for dispatching work. In particular, the initialization sequence of the RTS creates an array of N *EventManager* values, where N is the number of capabilities used by the program. We also fork N dispatcher threads, each pinned to run only on a distinct capability, and each monitoring the files registered with its respective *EventManager*. We modify the *threadWait* functions so that they register the callback with the manager for the capability that the calling thread is currently running on. This design allows each capability to perform the dispatching for Haskell threads running on its capability. Since the RTS scheduler balances threads across capabilities, this often leads to balancing the dispatching work as well. Furthermore, since a newly awoken thread is scheduled on the same capability that it previously ran on, the dispatcher's invocation of callbacks does not cause any cross-capability thread wakeup messages to be sent, improving core locality.

Unfortunately, this design introduces a problem that threatens to negate its ben-

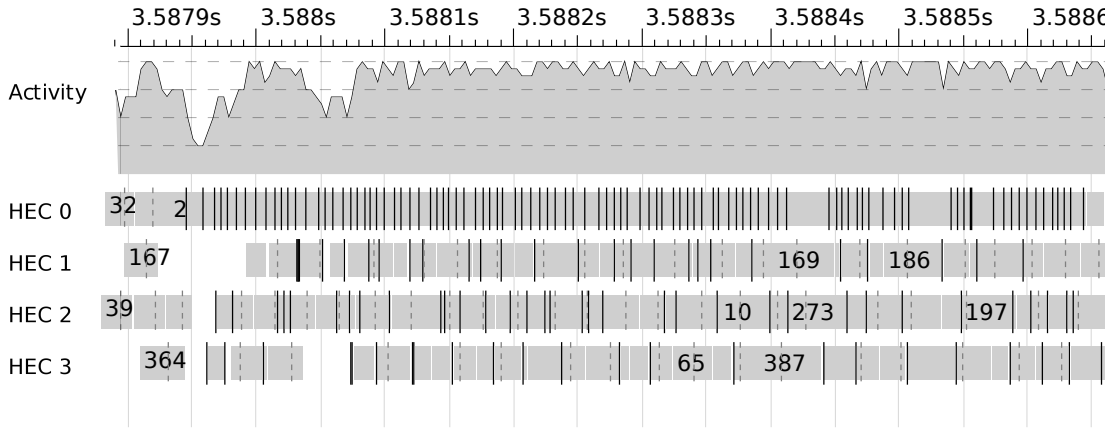


Figure 8.8: Event log fragment from an execution of `SimpleServer` using the concurrent callback table and 4 capabilities and illustrating concurrent registrations and dispatching.

efits: a dispatcher thread often makes blocking OS calls, and therefore relinquishes its HEC and causes a context switch to another native thread which begins executing the HEC's work [38]. Using one dispatcher per HEC increases the frequency with which this expensive operation occurs. For example, running `SimpleServer` using per-core dispatchers with 8 capabilities, we incur 35,000 context switches per second.

To alleviate this problem, we modify the dispatcher loop, to first perform a non-blocking poll for ready events (this can be accomplished with `epoll` by calling `epoll_wait` with a `timeout` of 0). If no events are ready, the dispatcher thread yields (by calling the *Control.Concurrent.yield* function), causing the scheduler to move it to the end of its capability's run queue. When the dispatcher thread returns to the front of the run queue, it again performs a non-blocking poll, and only if it again finds no ready events, then it performs a blocking OS call, which removes it from the capability's run queue. This optimization is effective when events become ready frequently, as the dispatcher collects events without incurring context switches. In the case that events become ready infrequently, the loop falls back to blocking calls. With this revision, the context switch rate for `SimpleServer` drops to 6,000 per second when using 8 capabilities.

The curve labelled "ParDisp" in Figure 8.5 shows the improved scaling resulting from applying per-core dispatchers with the aforementioned optimized dispatcher loop. `SimpleServer` now serves nearly 300,000 requests per second using 8 cores, achieving over triple the throughput of "Striped". The event log shown in Figure 8.10 shows how the 8 capabilities are now fully utilized and that the work of dispatching is now distributed over many capabilities, with HECs 0, 4, 5, and 7 visibly dispatching threads in this event log fragment (the closely spaced solid bars indicate dispatching of threads).

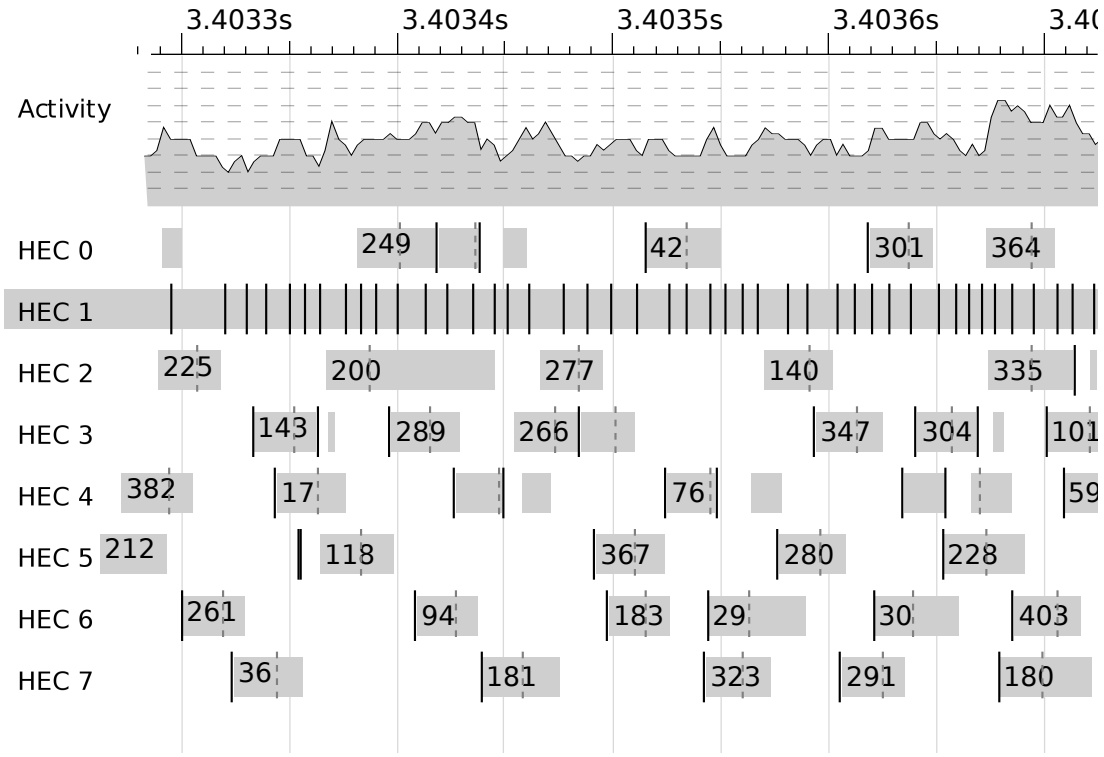


Figure 8.9: Timeline for an execution of `SimpleServer` using concurrent callback tables and 8 capabilities: The dispatcher thread on HEC 1 is fully utilized and has become a bottleneck.

8.3.4 Scalable OS Event Registration

Even after adding per-core dispatchers, the server stops scaling after about 8 cores, despite all cores being fully utilized. The event log fragment of Figure 8.10 provides insight into the problem. Although many worker threads finish quickly, returning to wait on their socket, some workers, for example thread 380 on HEC 6 and thread 126 on HEC 3 take 2-4 times as much time to service a request as other threads. Most interestingly, a large fraction of the time for these anomalous threads occurs *after* they request to wait on their socket (the dashed gray bar). For example thread 126 requires more than 100 microseconds to register its callback with the IO manager and block on its wait *MVar*, whereas thread 24 on HEC 1 finishes in around 20 microseconds. While at 8 cores, these slow-down occurrences are a small fraction of the overall thread executions, the frequency of this behavior becomes larger as more cores are added. This clearly suggests shared memory contention for some data structures shared among the capabilities.

By performing detailed profiling, we observed that the poor scaling is due to a global Linux kernel lock in the `epoll` subsystem used by the IO manager on our Linux server. This lock, named `epmutex`, is taken whenever an event is newly registered or deleted from an `epoll` object. This lock is required to allow `epoll` instances to be nested without forming cycles and is a truly global lock affecting operations on distinct `epoll` instances. Unfortunately, in the GHC IO manager, every register

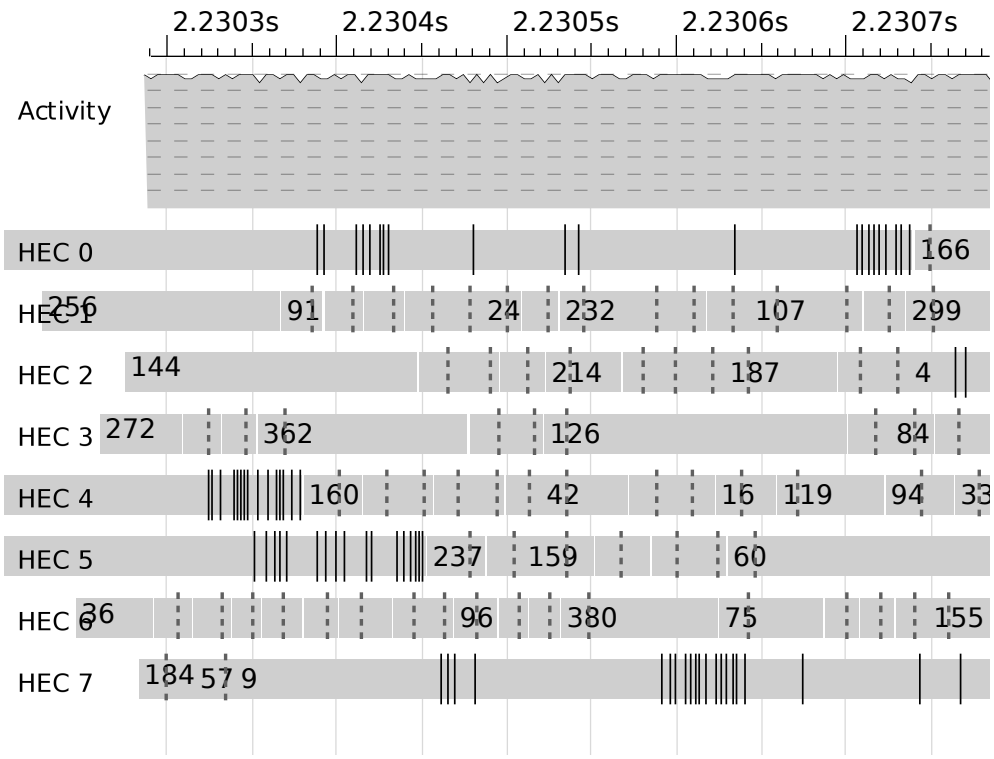


Figure 8.10: Timeline of `SimpleServer` with per-core dispatcher threads. All HECs are busy and dispatcher threads are running on several HECs.

and unregister call invokes an `epoll` operation that causes `epmutex` to be taken. Specifically, *threadWait* adds a new event to its `epoll` instance. Later, when the event is retrieved from the OS and the callback is invoked, the callback unregisters the subscription, causing the IO manager to delete the subscription from the `epoll` object. Therefore, every *threadWait* call requires a global lock to be taken twice. This results in increased contention as more cores are used. Furthermore, a contended lock may cause the native thread running the capability to be descheduled by the OS (which does not, unfortunately, show up in the event log trace).

We avoid this bottleneck by *reusing* `epoll` registrations and thereby avoiding taking `epmutex` on every operation. In particular, when we register an event, we first attempt to modify an existing registered event. Only if that fails – and indeed it will fail on the first registration on the file – we register a new event. Moreover, we register the event subscription in *one-shot mode*; event subscriptions registered in one-shot mode are automatically disabled by `epoll` after the application retrieves the first event for the subscription. Using one-shot mode allows us to safely leave the existing registration in place and has the added benefit that no extra `epoll_ctl` call is needed to delete the event registration.

The curve labelled "Final" in Figure 8.5, shows the result of applying this optimization on top of the previous improvements. We see greatly improved scaling through 20 cores, serving up to 668,000 requests per second at 20 cores, more than 18 times throughput improvement over the same program using the threaded RTS

with the GHC IO manager. Furthermore, Mio improves the single core performance of the server by 1.75x: with Mio, the server serves 58,700 requests per second while with the GHC IO manager it serves only 33,200 requests per second.

8.4 Implementation

Mio has been fully implemented, has been incorporated into the GHC code base, and will be released as part of GHC 7.8.1. Mio includes 874 new lines of code and 359 old lines of code deleted. Of these changes, 21 lines were added in the RTS code written in C. Most of these 21 lines are routine boilerplate while 7 are needed to support dynamically changing number of capabilities. In the following sections we describe our support for non-Linux operating systems, Mio's treatment of timers and several minor GHC RTS bugs and performance problems that we helped to address while implementing Mio.

8.4.1 BSD and Windows Support

On BSD systems such as FreeBSD, NetBSD and OpenBSD, Mio uses the `kqueue` subsystem, as `epoll` is unavailable on these platforms. The `kqueue` API provides two C functions as listed in Figure 8.11. The `kqueue` function creates a new `kqueue`. The `kevent` function takes a `changelist` array to specify events to be registered, unregistered, or modified and an `eventlist` array to receive events. Hence, this function may carry out both event registration and event polling at the same time. If `eventlist` is a `NULL` pointer, `kevent` performs event registration only, similar to the function performed by the `epoll_ctl`. Similarly, `kevent` can play the same role as `epoll_wait` by providing a `NULL` pointer for the `changelist` argument. Moreover, the `kqueue` subsystem also supports *one-shot* registrations. Hence, BSD variants can be supported using essentially the same approach we described in Section 8.3.

```
int kqueue(void);
int kevent(int kq,
           const struct kevent *changelist, int nchanges,
           struct kevent *eventlist, int nevents,
           const struct timespec *timeout);
```

Figure 8.11: `Kqueue` API.

Mio also uses `kqueue` on OS X, since Darwin, the foundation for Apple's OS X, is a variant of BSD. However, we encountered problems running parallel builds of the GHC compiler using Mio as described above. We have been unable to uncover the underlying source of the problem to our satisfaction. However, several Internet discussions suggest that the implementation of `kqueue` and `pipe` on OS X are unstable. We were able to resolve the observed problems on OS X by avoiding the use of one-shot mode on OS X, instead unregistering events after they are delivered,

and by sending wakeup writes to a pipe monitored by the dispatcher threads' kqueue instances on every event registration. With these changes, the behavior on OS X, including parallel builds of GHC, has been stable.

For UNIX platforms not supporting either `epoll` or `kqueue`, Mio utilizes the `poll` backend. Since `poll` does not provide a separate registration function, a Haskell thread must register interest for an event by adding it to a queue of events that should be registered and then interrupting the dispatcher if it is currently in a blocking `poll` call. The registration function performs this by sending a control message on a pipe monitored by the dispatcher thread's `poll` calls. This mechanism is unchanged from the implementation in the GHC IO manager for the `poll` backend.

GHC does not have an IO manager for Windows. For reading from and writing to files and sockets, blocking FFI calls are issued. Implementing a high-performance IO manager for Windows is challenging because Windows does not have scalable `poll`-like API. Windows I/O Completion Ports (IOCP) provides a scalable asynchronous API and substantial progress has been made in building an IO manager based on IOCP[2]. However, various complications prevented this work from being integrated into GHC. For this reason, we have not yet implemented Mio for Windows.

8.4.2 Timers

In the GHC IO manager, timers are monitored in the dispatcher loop. In Mio, the dispatcher loop has been modified to yield, which can lead to the dispatcher thread being delayed by other Haskell threads earlier on the run queue. Hence, the dispatcher loop can no longer be used for dispatching functions waiting on timers. Therefore, Mio moves timer monitoring into a separate timer dispatcher thread. This timer dispatching thread is identical to the GHC IO manager dispatcher thread, except that it no longer monitors file descriptors.

8.4.3 Support for Event-driven Programs

Although the *GHC.Event* module in the GHC IO manager is considered "private", a survey of hackage applications uncovered several programs that use this module to implement event-driven programs directly, rather than use Haskell threads and the *threadWaitRead* and *threadWaitWrite* functions. The GHC IO manager leaves a subscription registered until the client deregisters the subscription. However, as we described earlier, the typical use (via the *threadWait* function) is to deregister the subscription immediately after the callback is invoked, and Mio optimizes for this case by using the one-shot mode, automatically deregistering the subscription. To continue to support clients using the private Manager interface directly, we allow the Mio manager to be initialized with a flag indicating that it should *not* deregister subscriptions automatically after receiving an event for the subscription. Therefore, such programs can continue to use the Mio manager directly by simply adding a single flag when initializing a Mio manager.

2. `http://ghc.haskell.org/trac/ghc/ticket/7353`

8.4.4 GHC RTS Issues

Although GHC's threaded RTS provides a high-performance parallel garbage collector, we observed that garbage collection increasingly becomes an impediment to multicore scaling. One commonly used method for reducing the overhead of GC is to reduce the frequency of collections by increasing the allocation area (aka nursery) used by the program. In particular, in our experience, providing a large allocation area (for example, 32MB) improves multicore performance, confirming the observations of other researchers [76].

Each capability of GHC's threaded RTS has a private nursery that must be *cleared* at the end of a GC. This clearing consists of traversing all blocks in the nursery, resetting the free pointer of the block to the start of the block. In the parallel GC, the clearing of all capabilities' nursery blocks is done sequentially by the capability that started the GC, causing many cache lines held by other capabilities to be moved. Instead, we change the behavior such that each capability clears its own nursery in parallel.

Furthermore, many network programs use *pinned* memory regions, e.g. for *ByteString* values, in order to pass buffers to C functions, such as `send()` and `recv()`. The threaded RTS allocates such objects from the global allocator, which requires the allocating thread to acquire a global lock. Our insight led to a commit by Simon Marlow to allocate necessary blocks for small pinned objects from the capability-local nursery, which can be done without holding any global lock[3]. This change improves performance for multicore programs that allocate many small *ByteString*s.

Our dispatcher thread makes use of the *yield* function to place the dispatcher thread on the *end* of the run queue when it finds no ready events after polling once. GHC's RTS had a bug in which *yield* placed the thread back on the front of the run queue. This bug was uncovered by our use of *yield* which requires that the thread be placed at the end of the run queue.

All of these improvements have been incorporated into GHC and will be part of the GHC 7.8.1 release.

8.5 OS Bottlenecks & Bugs

While implementing Mio, we encountered hardware performance problems and a Linux kernel bug. To eliminate the possibility that Haskell (or rather GHC) was causing these problems, we implemented a C version of our `SimpleServer` program. This program, called `SimpleServerC`[4], is implemented in essentially the same way as `SimpleServer`: the main thread accepts connections and a configurable number of *worker native threads* service connections. The accepted connections are assigned to worker threads in round-robin order. Each worker thread uses a distinct epoll instance to monitor the connections it has been assigned and uses the `epoll` API in

3. `thread.gmane.org/gmane.comp.lang.haskell.parallel/218`

4. `github.com/AndreasVoellmy/epollbug`

the same way that `SimpleServer` with Mio uses `epoll`. On the other hand, it does not use a garbage collector, a thread scheduler and other subsystems of the GHC RTS that may introduce performance problems.

8.5.1 Load Balancing CPU Interrupts

In order to avoid interrupt handling from becoming a bottleneck, Linux (in its default configuration) evenly distributes interrupts (and hence workload) from network interface cards (NICs) to all of the CPU cores in the system. Unfortunately, this feature interacts poorly with power management features of our server. To save power, modern CPU cores can aggressively enter deep sleep states. Specifically, every CPU core on our multicore server can enter (1) C0 state in which the CPU core keeps busy in running code, (2) C1 state in which the CPU core turns off clocks and stops executing instructions, (3) C3 state in which the CPU core flushes the core's L1 and L2 caches into the last level cache, and (4) C6 state in which the CPU core saves all processor state in a dedicated SRAM and then completely removes voltage from the core to eliminate leakage power. When a CPU core wakes up from C6 state, it restores core state from the SRAM, activates L1 and L2 caches and core clocks. This expensive operation causes undesired application delays [1]. Experimentally, we found that, by distributing interrupts well, the workload of each CPU core is reduced to a level which triggers the CPU cores to enter deep sleep states. Later, when new packets arrive, the CPU cores have to wake up before they start processing packets, an expensive operation which leads to undesired delays and low throughput.

To verify this behavior, we compare `SimpleServerC`'s performance on our server in the default configuration with the default configuration with power-saving disabled, while varying the number of cores from 1 to 20. We disable power-saving by specifying the maximum transition latency for the CPU, which forces the CPU cores to stay in C0 state. Figure 8.12 shows the results, with the curves labelled "Default" and "NoSleep" showing the performance in the default configuration and the default configuration with power-saving disabled, respectively. Without limiting the CPU sleep states (curve "Default"), `SimpleServerC` cannot benefit from using more CPU cores and the throughput is less than 218,000 requests per second. In contrast, after preventing CPU cores entering deep sleep states (curve "NoSleep"), `SimpleServerC` scales up to 20 cores and can process 1.2 million requests per second, approximately 6 times faster than with the default configuration.

By profiling our server with `perf` in Linux and `i7z` [26] from Intel, we observed that the CPU cores of the server in the default configuration frequently entered deep sleep states. For example, when running `SimpleServerC` with 16 worker threads in the default configuration, on average 51% of the CPU time is spent on C6 state (including transition between C6 and running state), 33% on C3 state, 9% on C1 state, and only 7% of the CPU time is spent on running programs. In contrast, with power-saving disabled, the CPUs remain in C0 state throughout the program run.

Disabling sleep modes, however, is not a desirable solution for real systems due to power consumption. Rather than distribute interrupts of 10G NICs among all of the CPU cores in the system and then disable power saving, we limit the number

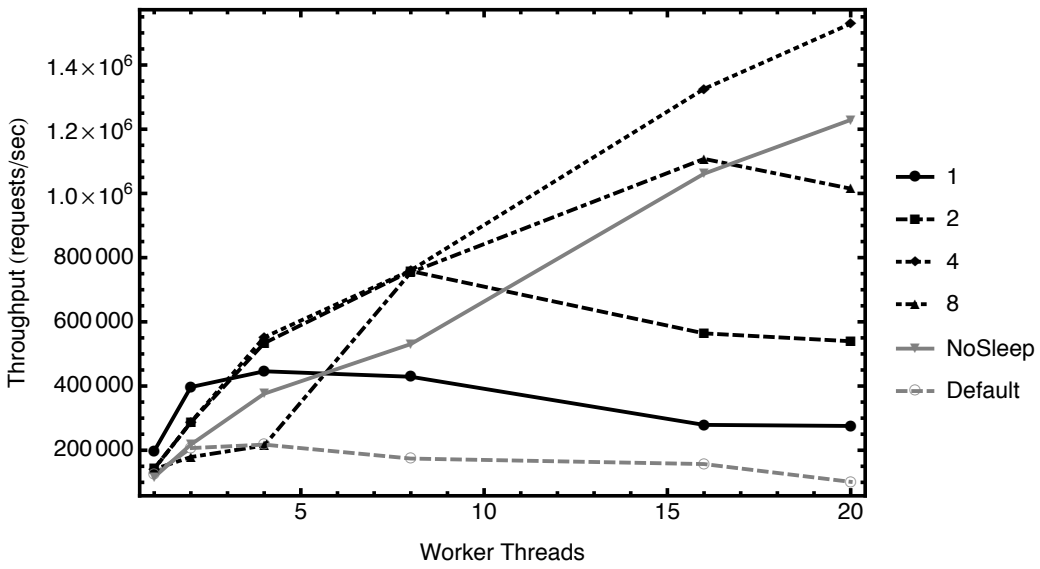


Figure 8.12: Throughput scaling of `SimpleServerC` when using different numbers of cores for interrupt handling and different power-saving settings.

of CPU cores used to handle hardware interrupts to a reasonably small number of CPU cores (In Linux, this can be achieved by adjusting `/proc/irq/N/smp_affinity` settings appropriately). In addition, we set the CPU affinity (using `taskset`) of `SimpleServerC` threads so that they run on the same CPU (NUMA) nodes as the interrupt handlers , which improves stability of the results.

Curves labelled "1", "2", "4", and "8" in Figure 8.12 show the performance of `SimpleServerC` when interrupts are handled by 1, 2, 4, and 8 cores respectively with default power-saving settings. By limiting the number of CPU cores to handle interrupts, `SimpleServerC` can scale as well or better as when power-saving is disabled. In particular, we see that using 4 cores for interrupt handling is optimal. With this configuration, `SimpleServerC` scales up to 20 workers and can process 1.5 million requests per second, with 100% of the CPU time of CPU cores handling interrupts spent on C0 state. With too few CPU cores (curves 1 and 2), the server cannot scale to more than 8 worker threads because the interrupt handling CPUs become a bottleneck. With too many CPU cores (e.g. curve 8) deep sleep is triggered frequently because the workload of each CPU core is reduced below a critical level.

We therefore prevent our server from entering deep sleep states in all evaluations in this chapter by using 4 CPU cores to handle hardware interrupts from our 10 Gbps NICs.

8.5.2 Linux Concurrency Bug

After removing various bottlenecks in our system, `SimpleServer` scaled to 20 cores and serves nearly 700,000 requests per second. This workload places an unusual burden on the Linux kernel and triggers a bug in Linux; Under such a heavy load, the `epoll` subsystem occasionally does not return read events for a socket, even though

the socket has data ready, causing worker threads in `SimpleServer` to wait for data indefinitely. To verify that this problem is due to a kernel bug, we verified that `SimpleServerC` also triggers the same problem (in fact, debugging this problem was the initial motivation for developing `SimpleServerC`).

We reported the bug to the Linux community. Kernel developers quickly identified the problem as a missing memory barrier in the `epoll` subsystem. In particular, when subscribing for an event on a socket, the interest mask for the socket is first updated and *then* the socket is checked for data to see if it is already ready. On the other hand, when processing incoming data, the Linux network stack first writes data to the socket and *then* checks the interest mask to determine whether to notify a waiter. To ensure that these operations occur in the correct order even when executing on different CPUs, memory barriers are required in both sequences. Unfortunately, a memory fence was missing in the registration code. This long-standing bug affects all Linux kernels since 2.4 and a patch fixing the issue has been accepted into the Linux kernel[5].

8.6 Evaluations

8.6.1 Methodology

We use a set of benchmark applications and workloads to characterize the performance of Mio. We use the following hardware, system software, and GHC and Haskell library versions.

Hardware

We run Haskell server programs on a SuperMicro X8OBN server, with 128 GB DDR3 memory and 8 Intel Xeon E7-8850 2 GHz CPUs, each having 10 cores with a 24 MB smart cache and 32 MB L3 cache. This server has four 10 Gbps Intel NICs. In the experiments, we turn off hyper-threading to prevent the system scheduler from scheduling multiple native threads that should run in parallel to a single physical CPU core. Client and workload generator programs run on Dell PowerEdge R210 II servers, with 16 GB DDR3 memory and 8 Intel Xeon E3-1270 CPUs (with hyper-threading) running at 3.40 GHz. Each CPU core has 256 KB L2 cache and 8 MB shared L3 cache. The servers communicate over a 10 Gbps Ethernet network and are configured so that each client server has a dedicated 10 Gbps path the main server. This level of network bandwidth was required to avoid network bottlenecks.

Software

The server software includes Ubuntu 12.04, Linux kernel version 3.7.1 on the SuperMicro server and version 3.2.0 on the Dell server, and Intel ixgbe driver (version 3.15.1). We use the latest development version of GHC at the time of this writing.

5. `https://patchwork.kernel.org/patch/1970231/`

For comparison of the threaded RTS of GHC without Mio, we use the development version of GHC with Mio patches removed.

We run `weighttp` [73] on the Dell servers to benchmark HTTP servers. `weighttp` simulates a number of clients making requests of an HTTP server, with each client making the same number of requests. Each client establishes a connection to the server and then repeatedly requests a web page and then waits for the response. In addition to recording the throughput for a given run, we extend `weighttp` to uniformly and randomly sample the latency for a fraction of the requests.

8.6.2 Web Server Performance

We evaluate two web servers written in Haskell, `acme` [2] and `mighty` [75], with the GHC IO manager and with Mio manager. `acme` is a minimal web server which does basic request and HTTP header parsing and generates a fixed response without performing any disk I/O, whereas `mighty` is a realistic, full-featured HTTP server, with realistic features such as slowloris protection. For comparison, we also measure the performance of `nginx`, arguably the world's fastest web server, written in C specifically for high performance. We evaluate all three servers with 400 concurrent clients, a total of 500,000 requests and sample the latency of 1% of the requests.

Figure 8.13 shows the result. We see that when using the GHC IO manager both `acme` and `mighty` scale only modestly to 4 cores and then do not increase performance beyond 30,000 requests per second. On the other hand, with Mio, `acme` scales well to 12 cores serving up to 340,000 requests per second and `mighty` scales up to 20 cores serving 195,000 requests per second at peak performance, resulting in a 8.4x and 6.5x increases (respectively) in peak server throughput using Mio. The graph also demonstrates that a realistic web server in Haskell, `mighty`, performs within a factor of 2.5x of `nginx` for every number of cores and performs within 2x of `nginx` for 8 cores and higher.

Figure 8.14 shows the cumulative distribution function (CDF) of the response time for the `acme` and `mighty` servers when run with 12 capabilities with and without Mio and for the `nginx` server when run with 12 worker processes. The expected response time for `acme` and `mighty` is 1.1 ms and 2.0 ms, respectively, using the Mio manager and are both 11.3 ms with the GHC IO manager. Hence, Mio improves the expected response time for `acme` and `mighty` by 10.3x and 5.7x. The 95th percentile response time of `acme` and `mighty` with Mio are 3.1 ms and 5.9 ms, whereas with the GHC IO manager they are 13.9 ms and 14.5 ms, representing a 4.4x and 2.5x reduction in 95th percentile response time. We also observe that the response time distribution of the Haskell servers closely matches that of `nginx`.

Figure 8.15 shows the throughput of `acme` and `mighty` on a FreeBSD server. For this experiment, we use different hardware than other benchmarks. In particular, two 12 core (Intel Xeon E5645, two sockets, 6 cores per 1 CPU, hyper-threading disabled) servers are connected with 1 Gbps Ethernet. One server runs Linux version 3.2.0 (Ubuntu 12.04 LTS) while the other runs FreeBSD 9.1. The web servers are running on the FreeBSD server and they are measured from the Linux server using the same benchmarking software as our earlier evaluations. `acme` and `mighty` scale

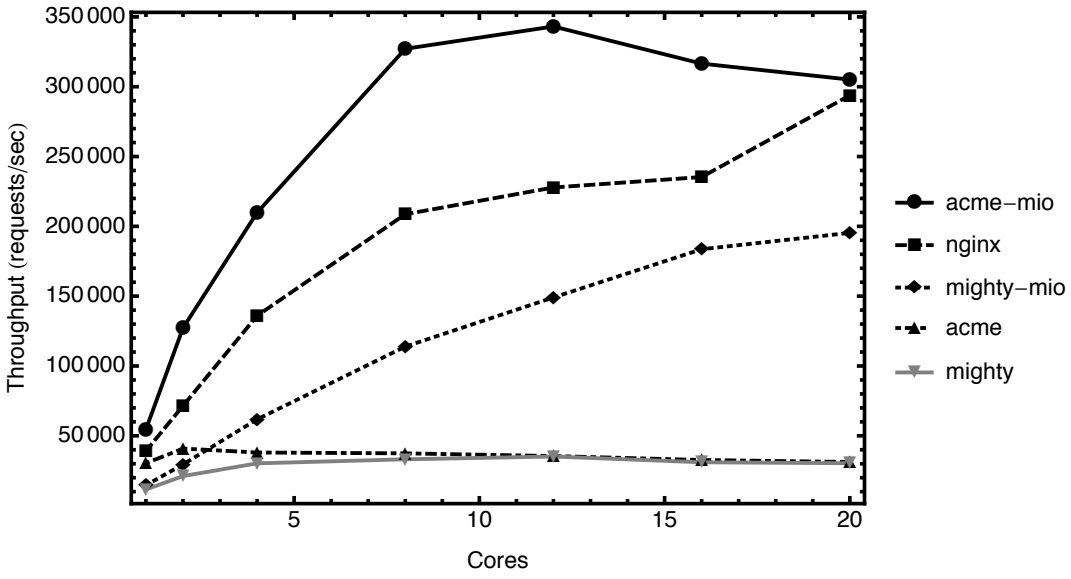


Figure 8.13: Throughput of Haskell web servers `acme` and `mighty` with GHC IO manager and Mio manager and `nginx` in HTTP requests per second as a function of number of capabilities used.

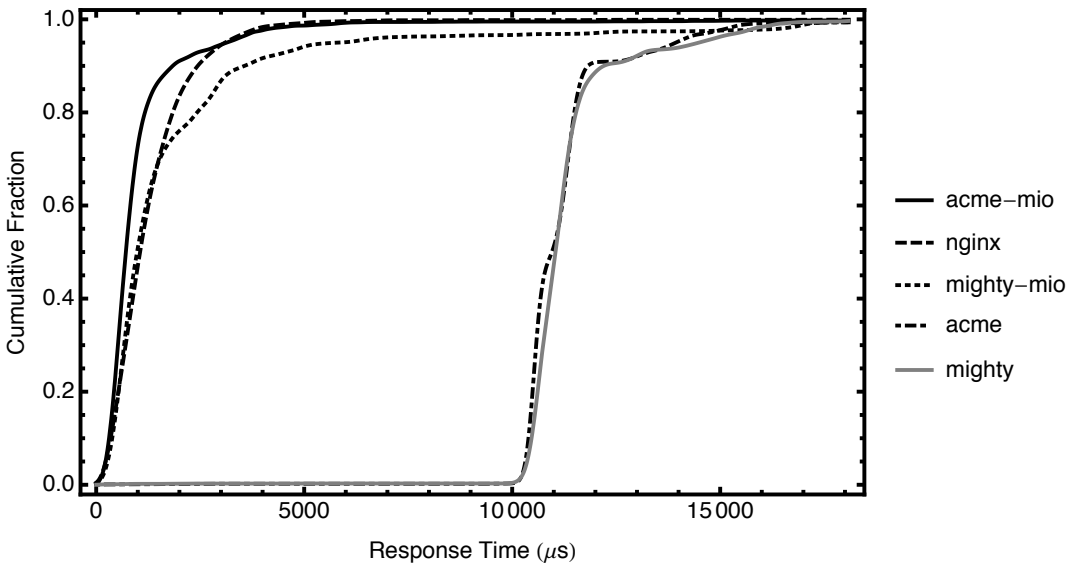


Figure 8.14: Cumulative Distribution Function (CDF) of response time of `acme` server with GHC IO manager and with Mio manager at 12 cores and 400 concurrent clients.

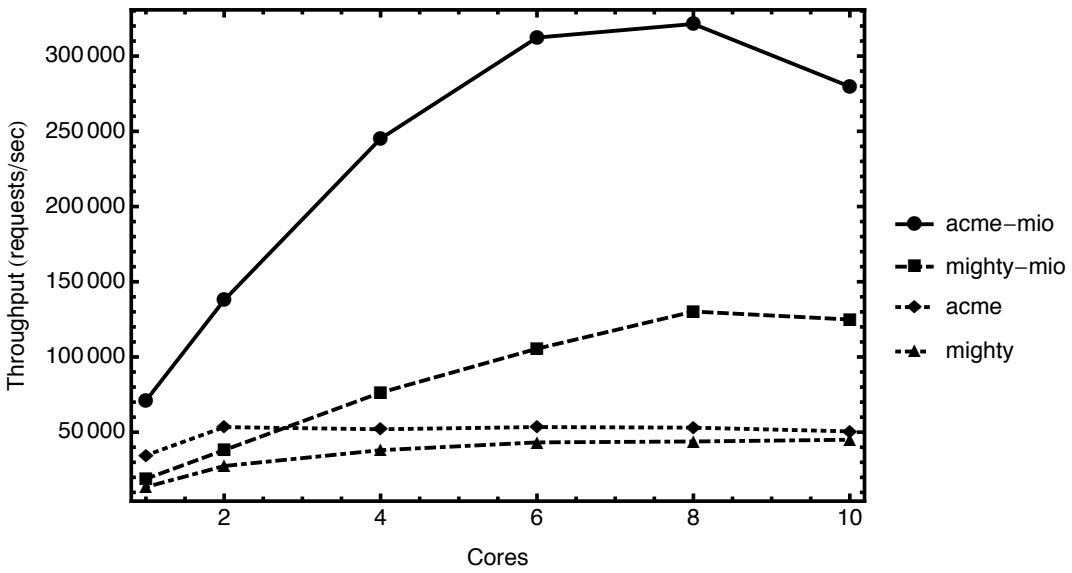


Figure 8.15: Throughput of Haskell web servers `acme` and `mighty` with GHC IO manager and Mio manager on FreeBSD.

up to 7 and 8 cores, respectively, with Mio manager, but scale poorly with the GHC IO manager. `acme` achieves a peak throughput of 330,000 requests per second, which saturates the 1 Gbps network connection between these two servers and inhibits further scaling. Using this same hardware, we evaluated `acme` and `mighty` on the Linux server with clients on the FreeBSD server. The resulting throughputs were nearly identical to those in Figure 8.15 demonstrating that performance with Mio on FreeBSD is comparable to that on Linux up to 8 cores.

8.6.3 Threaded RTS Overhead

CloudHaskell [17] developers have noted that the threaded RTS of GHC often introduces significant latency to their CloudHaskell programs[6] for light loads which do not overwhelm the single-threaded execution. Mio substantially reduces the latency overhead of the threaded RTS for these loads.

We evaluate the threaded RTS overhead with a benchmark program based on a similar program developed by the CloudHaskell community [48], which consists of a server which simply echoes a message sent by a client. We run this program using a single client and measure the round-trip time for each echo request. Figure 8.16 shows the CDF of the response times for the non-threaded RTS, the threaded RTS using the GHC IO manager, and the threaded RTS using the Mio manager, with both threaded RTSs using a single capability. The expected latencies are 63 microseconds (μs), 69 μs and 84 μs for the non-threaded RTS, threaded RTS with Mio and threaded RTS with GHC IO manager, respectively, while the 99th percentile latencies are 73 μs, 83 μs and 119 μs, respectively. The Mio manager therefore significantly reduces

6. `http://www.edsko.net/2013/02/06/performance-problems-with-threaded/`

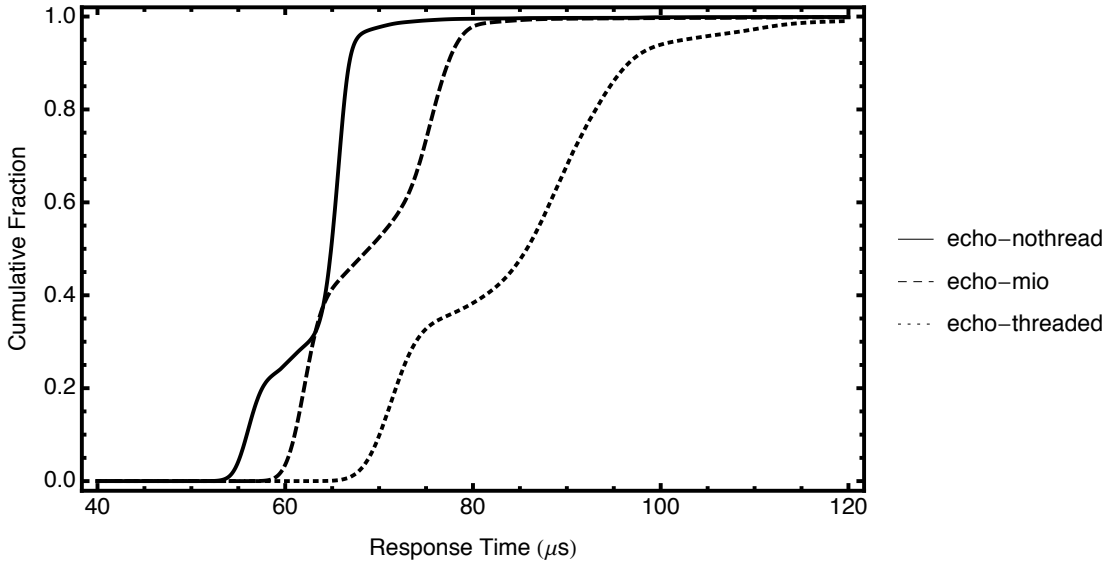


Figure 8.16: Cumulative Distribution Function (CDF) of response time of echo server with non-threaded RTS, threaded RTS with GHC IO manager and threaded RTS with Mio manager.

latency over the GHC IO manager and adds just 6 μs latency on average over the non-threaded RTS for this workload.

8.7 Related Work

8.7.1 Lightweight Components

Several prominent programming languages provide lightweight threads or processes. Erlang [6] provides lightweight processes that communicate only using message passing. The Go programming language [7] provides sophisticated coroutines, called goroutines. The Go scheduler also uses N (one per cpu) poll servers. In upcoming releases, Go's IO event notification will be integrated with the scheduler, whereas with Mio, the event notification is implemented entirely in Haskell.

8.7.2 Event Libraries

Event libraries such as `libevent`[7], `libev`[8], and `libuv`[9] also provide abstractions over platform-specific event notification APIs. These APIs require programmers to express programs as event handlers, while Mio is designed and optimized to support a

7. `http://libevent.org/`

8. `http://software.schmorp.de/pkg/libev.html`

9. `https://github.com/joyent/libuv`

lightweight thread programming model. Furthermore, these libraries do not support concurrent registration of events and notification; the program is *either* polling the OS for events *or* executing a user's event handlers, where new registrations may occur. In contrast, with Mio, a native thread may register event callbacks with a dispatcher while that dispatcher is polling the OS for notifications.

8.7.3 Prefork Technique

A typical event-driven program runs one event loop in a single process resulting in good utilization of one core but not scaling on multiple cores. For web servers, the *prefork* technique is used to utilize multiple cores. Historically, the prefork technique worked as follows. After setting up a listen socket and before starting the service, the main process forks multiple processes, called *workers*, so that the workers share the same listen socket. When a new connection arrives, one of the workers wins the race to accept the new connection and then serves the connection until completion.

Recently, systems such as `nginx` [5] and `Node.js`[10] have adapted the prefork technique by creating N worker processes (where N is the number of cores) with each process running an event-driven loop. The prefork technique can also be applied to Haskell servers by simply forking several identical processes. For instance, `mighty` can use the prefork technique to achieve good multicore performance [75] when using the GHC IO manager.

However, there are two disadvantages when adopting the prefork technique for event-driven programming: 1) *boilerplate*: code for interprocess communication is necessary to manage workers such as stopping services and reloading configuration files. 2) *thundering herd*: all workers are unnecessarily rescheduled when a new connection arrives [62]. Fortunately, the thundering herd problem for original perfork-style servers has been addressed in recent operating systems (such as Linux 2.4 or later) by waking up only one process [43] that is blocking on an `accept()` call.

Unfortunately, a thundering herd problem can still occur with the newer style of prefork servers, even on new operating systems. In this case, multiple processes watch the same listen socket using `epoll` or `kqueue` (as opposed to blocking on `accept()` calls). These event systems notify all listeners when an event occurs. High performance servers should avoid unnecessary overhead to have resistance to Slashdot effect (or flash crowd). With Mio, servers may avoid thundering herd by using a single Haskell thread for accepting new connections, as both `acme` and `mighty` do.

8.8 Summary

We presented and evaluated Mio, a new *multicore IO manager*, that improves realistic web server throughput by over 6.5x and expected response time by 5.7x. As a result, the performance of network servers in Haskell rivals that of servers written in C.

10. `http://nodejs.org/`

Chapter 9

Conclusions and Future Work

9.1 Contributions

Traditional network architecture, which emphasizes distributed routing and resource reservation, has been highly successful. Through international standardization efforts, network operators can combine network devices from several vendors into a single network. The distributed systems approach has led to highly robust network systems that are resilient to link and node failures.

On the other hand, today's networks are both complex and lacking in flexibility. In particular, network operators must master a wide variety of protocols and vendor-specific configuration interfaces. In addition, the profusion of middleboxes introduces topology-based constraints on the implementation of network services. This complexity causes network management to be time-consuming and fraught with risk. In addition, the limited configuration interfaces prevent network administrators from easily tailoring their network to their requirements.

SDN addresses these problems by separating the control and data planes of the network by way of a standard, uniform, and expressive API. This API allows a centralized control system to have sufficient visibility and control over forwarding plane resources to automate the deployment of a wide variety of network services, such as switching, routing, access control, load balancing, and traffic policing. As a result, SDN has recently generated substantial interest from both commercial and academic communities.

Although some basic aspects of SDN architecture, such as the OpenFlow protocol and OpenFlow message services, have emerged, implementing an SDN control plane remains a daunting task. In particular, SDN programmers must design programs that translate their requirements into OpenFlow flow tables and must maintain these flow tables as network and policy changes occur. Moreover, the quality of flow tables generated by their program is a critical factor in the performance of the overall network.

The main contribution of this dissertation is the development of a novel SDN programming model, called algorithmic policies, that provides SDN programmers with a convenient, high-level programming model for expressing network policies. In

particular, SDN programmers define their network policy by writing a function in a general-purpose language that will be executed on every packet entering the network. The algorithmic policy programming model thereby simplifies network programming in two ways: (1) the programmer writes a single, logically centralized function in a familiar programming language without needing to cope with distributed state or express their ideas using specialized or restrictive languages, and (2) the programmer need not generate and maintain flow tables for distributed network elements, an error-prone and performance-critical task.

This dissertation introduces Maple, an SDN programming framework consisting of a domain specific language that can be embedded into any programming language with appropriate support. We have implemented Maple in both Java and Haskell, including an optimizing compiler and a multicore runtime system that automatically and scalably executes algorithmic policies on distributed OpenFlow switches. This is accomplished by a novel tracing runtime system that dynamically models and compiles OpenFlow flow tables for all network devices. The tracing runtime system's algorithms include several optimizations, that help to reduce the size of switch flow tables, reduce the number of priority levels used by rule sets and reduce the number of update commands used. These optimizations improve network performance by increasing the proportion of packets that are processed locally by switches with up-to-date flow tables. With this approach, programmers can write high level programs, while automatically generating flow tables that match what a skilled network engineer would generate with a hand-written rule compiler.

In addition, Maple's runtime maintains consistency between algorithmic policy state and distributed flow tables using state dependency tracking and fast repair techniques. Maple is implemented using McNettle, a high-performance OpenFlow messaging service in Haskell. McNettle achieves the highest performance of any freely available OpenFlow controller on multicore servers, through improvements to the lightweight threading implementation of the Glasgow Haskell Compiler and careful optimization of IO operations and message processing.

We have demonstrated that Maple can be used to implement a number of realistic network control algorithms, including L2 shortest path routing, declarative access control policies, traffic monitoring, and policies that include run-time modifiable administrative parameters.

9.2 Limitations

This dissertation focuses on OpenFlow as the forwarding abstraction. One key question is: to what extent is Maple specific to OpenFlow? In other words, could Maple be applied to other SDN abstractions? We provide some insight into this question by describing the key requirements that Maple places on the forwarding plane.

Maple applies a fully dynamic approach; *i.e.* it does not perform any static analysis or static compilation. Instead, Maple must execute the user's packet processing function on packets in order to generate flow tables. Maple therefore relies on the ability to observe real network traffic and would not be applicable to switch inter-

faces which do not provide this data. OpenFlow provides network traffic data via `packet-in` messages, while other protocols may provide this data using other mechanisms. For forwarding abstractions that do not provide packet data, an alternative approaches based on policy compilation and static analysis must be used.

Although Maple currently generates OpenFlow flow tables to configure the forwarding plane, other models, such as ForCES could be used as well. An implementation on ForCES devices could be accomplished by emulating OpenFlow flow tables.

A second key question is: are there algorithms that are not well-suited to Maple's approach? Currently, any algorithm which changes system state on every packet will prevent Maple from caching any decisions in flow tables. For example, an algorithm which increments a counter on each packet will lead to every packet being sent to the controller. A more realistic network function would be in the implementation of layer 7 functionality. For example, to implement a classifier that matches all HTTP GET requests, the packet processing function may need to keep state across packets, since a single HTTP GET request may be carried by multiple TCP packets. Implementing such functions efficiently would require the ability to identify and implement stateful computations in the forwarding plane. OpenFlow does not currently support this capability and hence, these algorithms could not be implemented in Maple on current OpenFlow devices without switch-controller interaction.

A third key question: is it always acceptable to run the controller on a single server with sufficient CPU cores, and if not, can Maple be applied to these settings? There are several scenarios in which multiple control servers are most appropriate. In particular, in networks spanning large distances, the speed of light may impose unacceptably long response time from a centralized control server to various events, such as new types of traffic. In these cases, a distributed system of control servers is needed. Another case occurs when multiple networks, each under different administrative control, connect. In such cases, it is required to run each network with a distinct control server.

In both cases, protocols will be needed to coordinate control systems. In the case of multiple autonomous networks, existing protocols, such as BGP [56] can be used. In the case of distribution to reduce latency, it may be possible to execute multiple Maple servers using distributed database technology to implement global state.

9.3 Future Work

9.3.1 Deployment Experience

Although Maple has been deployed on real and virtual switches in lab and simulated environments, we have no experience in applying Maple in production networks. Such experiments will be crucial to understand the current limitations of Maple and to identify new research opportunities.

9.3.2 Compact Network Function Representations

Algorithmic representations of network functions often lead to compact function encodings. These representations are then expanded into much larger representations, both as rule sets for switches and in intermediate representations in the controller, such as Maple's decision tree representation. Large representations lead to several problems. Large lists of rules may exceed switch table capacity, degrading performance. Large flow tables lead to long switch table update times, which typically grow linearly with the size of the switch flow table. Finally, large intermediate representations lead to longer recompilation times (triggered by state changes, e.g. topology changes).

One technique that can be used to produce more compact representations is the multi-table pipeline, which has been standardized in recent versions of OpenFlow. Consider a network function which uses a classifier based on source addresses to determine the QoS treatment of the packet and uses a destination address to determine the output port of the packet. Suppose n_1 rules are required to implement the QoS classifier for n_1 source addresses and n_2 rules are required to implement the output port classifier for n_2 destination addresses. To implement a single combined classifier that incorporates both QoS settings and output port actions will require $n_1 * n_2$ rules. On the other hand, we may use two tables in a pipeline, where the effects of the actions are accumulated and executed at the end of the pipeline, to implement the combined function with $n_1 + n_2$ rules. This improvement can make it feasible to implement certain functions in OpenFlow which are currently not possible given the current switch table rule capacities.

9.3.3 Switch-Specific Optimizations

Although all OpenFlow switches adhere to the OpenFlow protocol, the performance of a particular rule set varies dramatically across implementations. For example, some switches can not provide hardware-accelerated packet processing if both L2 and L3 fields are matched in a single rule. In addition, switches differ in terms of the cost of control plane operations. For example, in the Open VSwitch virtual switches, inserting a single flow table rule interrupts packet processing and causes lookup data structures used by the forwarding process to be flushed. On the other hand, hardware switches may be able to process updates and lookups in parallel. By applying switch-specific optimizations, the compiler/RTS will achieve substantial network performance improvements. This approach would allow users to write truly portable SDN programs that can run unchanged on a variety of switches and achieve predictable performance.

9.3.4 Network Function Virtualization (NFV)

Telecoms and Internet service providers rely on a number of specialized, proprietary physical devices to implement required services. For example, BRAS devices implement authentication and QoS policies for DSL subscribers and Session Border

Controllers implement control in Voice over Internet Protocol networks. Network Function Virtualization (NFV) aims to virtualize these diverse functions into software that can run on standardized computational platforms, thereby reducing the deployment constraints and reducing capital and operational expenses. This goal is substantially similar to the goals of SDN. In fact, NFV can be seen as an extension of SDN from basic switching, routing, access control and QoS, to more complex functionality.

Maple could be used to support NFV in multiple ways. On the one hand, Maple in its current form complements NFV by providing a convenient method for flexibly composing and orchestrating NFV platforms, which may consist of both specialized hardware as well standard server hardware running virtualized appliances. Such deployments may require complex network control to cope with issues such as middlebox failures (which should not cause network failures) and balancing of load across multiple middlebox instances to handle large traffic loads cost effectively.

On the other hand, the Maple programming model and runtime system could be extended to directly implement these complex functions. For example, it may be possible to extend Maple with new commands, such as incrementing and reading counters and recording packets and alert messages on particular kinds of packets. Such extensions would allow SDN programmers to directly express complex network functions in a unified and familiar general-purpose programming language. To support this, the Maple compiler and runtime system would require extensions to identify how to compile programs performing more general packet processing, perhaps by offloading to specialized hardware when available, and otherwise relying on conventional servers to implement network functions.

9.4 Final Remarks

This dissertation has demonstrated that it is practical and efficient to use arguably the simplest SDN programming model, in which a programmer defines the overall network behavior by writing a packet processing function in a familiar, general-purpose programming language. The packet processing function is conceptually applied to every packet entering the network and has access to logically centralized state.

We believe that the algorithmic policy programming model and Maple compiler and runtime open the door to a number of extensions and refinements. In particular, the programming model can be extended naturally to handle a number of more complex packet processing functions, including the ability to send new packets into the network in response to packets and to manipulate packet counters. Similarly, the runtime and compilation techniques can be extended to observe further information about algorithmic policies and to compile to more refined forwarding models, such as multi-table pipelines.

Bibliography

[1] Intel Xeon Processor E7- 8800/4800/2800 Product Families, Volumn 2. 2011.

[2] acme-http. `http://hackage.haskell.org/package/acme-http`.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[5] Andrey Alexeev. nginx. In *The Architecture of Open Source Applications, Datasheet Volume 2*. 2012.

[6] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. 2007.

[7] Ivo Balbaert. *The Way to Go: A Thorough Introduction to the Go Programming Language*. 2012.

[8] Beacon. Beacon, 2013. [Online; accessed 19-January-2014].

[9] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[10] Boost c++ libraries. `http://www.boost.org`.

[11] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. Maestro: Balancing Fairness, Latency and Throughput in the OpenFlow Control Plane. Technical report, Rice University, 2011.

[12] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols*

for computer communications, SIGCOMM '07, pages 1–12, New York, NY, USA, 2007. ACM.

[13] Cbench. Cbench, 2012. [Online; accessed 10-April-2012].

[14] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, March 2010.

[15] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, 2011.

[16] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. Towards an Elastic Distributed SDN Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 7–12, New York, NY, USA, 2013. ACM.

[17] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 118–129, 2011.

[18] David Erickson. The Beacon OpenFlow Controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[19] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[20] Floodlight. `http://www.projectfloodlight.org/floodlight/`.

[21] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[22] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[23] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[24] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Maziéres, and Nick McKeown. Where is the Debugger for My Software-defined Network? In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 55–60, New York, NY, USA, 2012. ACM.

[25] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical Declarative Network Management. In *WREN '09: Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 1–10, New York, NY, USA, 2009. ACM.

[26] i7z. `http://code.google.com/p/i7z`.

[27] Internet2. Internet2, 2014. [Online; accessed 29-January-2014].

[28] Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 81–92, 2009.

[29] Teemu Koponen, Martin 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.

[30] 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*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[31] Jonathan Lemon. Kqueue - A Generic and Scalable Event Notification Facility. In *the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 141–153, 2001.

[32] Davide Libenzi. *Improving (network) I/O performance*, 2001.

[33] S. Marlow et al. *Haskell 2010 Language Report*, 2010.

[34] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th international symposium on Memory management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM.

[35] Simon Marlow and Simon Peyton Jones. Multicore garbage collection with local heaps. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 21–32, New York, NY, USA, 2011. ACM.

[36] Simon Marlow and Simon Peyton Jones. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume 2*. 2012.

[37] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *14th ACM International Conference on Functional Programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[38] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 65–78, 2009.

[39] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Extending the Haskell Foreign Function Interface with Concurrency. In *Proceedings of the Haskell Workshop*, 2004.

[40] B.D. Marsh, M.L. Scott, T.J. LeBlanc, and E.P. Markatos. First-class user-level threads. *ACM SIGOPS Operating Systems Review*, 25(5):110–121, 1991.

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

[42] Jeffrey C. Mogul et al. DevoFlow: cost-effective flow management for high performance enterprise networks. In *9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets '10, pages 1:1–1:6, New York, NY, USA, 2010. ACM.

[43] Stephen P. Molloy and Chuck Lever. Accept() scalability on Linux. In *ATEC '00 Proceedings of the annual conference on USENIX Annual Technical Conference*, 2000.

[44] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. A compiler and run-time system for network programming languages. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 217–230, New York, NY, USA, 2012. ACM.

[45] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-defined Networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[46] David Mosberger and Tai Jin. httperf a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26(3):31–37, December 1998.

[47] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: Dynamic Access Control for Enterprise Networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 11–18, New York, NY, USA, 2009. ACM.

[48] network transport. network-transport, 2014. [Online; accessed 19-January-2014].

[49] Neutron. Neutron, 2013. [Online; accessed 19-January-2014].

[50] Opendaylight. `http://www.opendaylight.org`.

[51] OpenStack. Openstack, 2014. [Online; accessed 19-January-2014].

[52] Bryan O'Sullivan and Johan Tibell. Scalable I/O Event Handling for GHC. In *Proceedings of the 2010 ACM SIGPLAN Haskell Symposium (Haskell'10)*, pages 103–108, 2010.

[53] R. Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley professional computing series. Addison Wesley, 2000.

[54] Simon Peyton Jones, A Gordon, and S Finne. Concurrent Haskell. In *Proceedings of 23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, 1996.

[55] Pox. `http://http://www.noxrepo.org/pox/about-pox`.

[56] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1654 (Proposed Standard), July 1994. Obsoleted by RFC 1771.

[57] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. In *Proceedings of the 13th International Conference on Passive and Active Measurement*, PAM'12, pages 85–95, Berlin, Heidelberg, 2012. Springer-Verlag.

[58] Ryu. `http://http://osrg.github.io/ryu/`.

[59] Devavrat Shah and Pankaj Gupta. Fast Updating Algorithms for TCAMs. *IEEE Micro*, 21(1):36–47, January 2001.

[60] S.Hares. Analysis of Comparisons between OpenFlow and ForCES. Internet-Draft draft-hares-forces-vs-openflow-00.txt, IETF Secretariat, 2012.

[61] Simple Network Access Control (SNAC). `http://www.openflow.org/wp/snac/`.

[62] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming*. Addison-Wesley Professional, 2004.

[63] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. In *HotNets*. ACM SIGCOMM, 2009.

[64] David E. Taylor and Jonathan S. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, June 2007.

[65] threadscope. `http://www.haskell.org/haskellwiki/ThreadScope`.

[66] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. In *Hot-ICE*, 2012.

[67] Andreas Voellmy, Bryan Ford, Paul Hudak, and Y. Richard Yang. Scaling Software-Defined Network Controllers on Multicore Servers. Technical report, YaleCS1468, 2012.

[68] Andreas Voellmy and Paul Hudak. Nettle: Taking the Sting Out of Programming Network Routers. In *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, pages 235–249, Berlin, Heidelberg, 2011. Springer-Verlag.

[69] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 87–98, New York, NY, USA, 2013. ACM.

[70] Rob von Behren, Jeremy Condit, and Eric Brewer. Why Events Are A Bad Idea (for High-Concurrency Servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, 2003.

[71] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based Server Load Balancing Gone Wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[72] Z. Wang, T. Tsou, J. Huang, X. Shi, and X. Yin. Analysis of Comparisons between OpenFlow and ForCES. Internet-Draft draft-wang-forces-compare-openflow-forces-01.txt, IETF Secretariat, 2012.

[73] weighttp. `http://redmine.lighttpd.net/projects/weighttp/wiki`.

[74] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, October 2001.

[75] Kazu Yamamoto. Mighttpd - a High Performance Web Server in Haskell. In *The Monad.Reader Issue 19*. 2011.

[76] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. Allocation wall: a limiting factor of Java applications on emerging multi-core platforms. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 361–376, 2009.